A Graph Parsing Algorithm

Technical Report CSE-90-04

Carolyn L. McCreary

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

July 31, 1990

A Graph Parsing Algorithm

Carolyn L. McCreary
Auburn University

A Graph Parsing Algorithm

Carolyn McCreary
Department of Computer Science and Engineering
109 Dunstan Hall
Auburn University, AL 36849

A Graph Parsing Algorithm

The theory of graph-grammars is a generalization of formal languages and the theory of string grammars. The objects that are created through productions are graphs rather than strings and the production rules must include instructions for connecting the produced graph to the existing graph (embedding rules). The lack of clear and concise statements of the production rules and the absence of a parsing algorithm to recapture a set of productions are two problems that have prevented much use of graph-grammars. Most graph rewriting schemes have a complex set of rules for determining which production can be applied and for specifying the embedding of the replacement graph in the host graph.

This paper presents a parsing algorithm for decomposing two large classes of graphs, dependency graphs and Hasse graphs. A dependency graph is the transitive closure of a directed acyclic graph (dag) and a Hasse graph is the transitive reduction of a dag. A dag can be uniquely associated with each of these graphs. The graph rewriting scheme is simple and the resulting parse is unique. This parsing algorithm is the first known polynomial time, $O(n^3)$ parsing algorithm for a general class of graphs.

List of Symbols

$\in$	is an element of
$\notin$	is not an element of
$\subseteq$	is a subset of
$\varnothing$	null set
$\cap$	set intersection
$\cup$	set union
$<_G$	partial ordering defined in section 3.2
$\neq$	not equal
Σ	sum of
$H^*(S)$	$H(S) \cup S$
$F/F_1 \ldots F_n$	Quotient graph defined in section 4.1
$\mathcal{S}$	partition of graph nodes into sibling sets
$\mathcal{M}$	partition of graph nodes into mate sets

A Graph Parsing Algorithm

1 Introduction

The theory of graph-grammars is a generalization of formal languages and the theory of string grammars. The objects that are created through productions are graphs rather than strings and the production rules must include instructions for connecting the produced graph to the existing graph (embedding rules). The lack of clear and concise statements of the production rules and the absence of a parsing algorithm to recapture a set of productions are two problems that have prevented much use of graph-grammars. Most graph rewriting schemes have a complex set of rules for determining which production can be applied and for specifying the embedding of the replacement graph in the host graph.

This paper presents a parsing algorithm for decomposing two large classes of graphs, dependency graphs and Hasse graphs. A dependency graph is the transitive closure of a directed acyclic graph (dag) and a Hasse graph is the transitive reduction of a dag. A dag can be uniquely associated with each of these graphs. The graph rewriting scheme is simple and the resulting parse is unique. This parsing algorithm is the first known polynomial time, $O(n^3)$ parsing algorithm for a general class of graphs.

2 Definitions and Concepts

2.1 Graph definitions and notations

A *directed graph* $G = (V,E)$ consists of a set V of vertices and a set $E \subseteq VxV$ of directed edges such that each edge $e \in E$ is associated with a unique pair, (v,w) of vertices. The set of edges is written E(G) and the set of vertices is V(G) or simply G. For $(v,w) \in E(G)$, v is called a *parent* of w and w is a *child* of v. A *path* from v_0 to v_n in directed graph G is a sequence of edges $\{(v_0,v_1),(v_1,v_2),...,(v_{n-1},v_n)\}$ in E(G). If there is a path from v to w in directed graph G, then v is said to be an *ancestor* of w and w is a *descendant* of v.

For any directed graph G, the *source* of G is the set of nodes S where for each $s \in S$ there is no incoming edge. $S = \{s \in G \mid (v,s) \notin E(G) \text{ for any } v \in G\}$. The *sink* of G is the set of nodes M such that for each $m \in M$, there is no outgoing edge.
$M = \{m \mid (m,v) \notin E(G) \text{ for any } v \in G\}$.

2.2 Dependency graphs and Hasse graphs

A *dependency graph* is a directed acyclic graph having the transitive property. The *transitive property* for graph G means that if (a,b) and (b,c) are edges of G, then (a,c) is also an edge of G. For any dag, the corresponding dependency graph is found by augmenting the set of edges with edges that satisfy the transitive property, i.e.,finding the transitive closure of the edges. The transitive property imposes a proliferation of graph edges. For example, the simplest connected dag, as shown in Figure 1, is a linear sequence of k nodes, $n_1, n_2, ..., n_k$ containing k-1 edges, (n_i, n_{i+1}) $1 \le i < k$. Its corresponding dependency graph contains (k-1)(k-2)/2 edges (n_i, n_j) where $1 \le i < j \le k$. For a given dag G, there is a path from node u to node v, if and only if the corresponding dependency graph, D(G), contains edge (u,v). D(G) is maximal in the sense that no edges can be added to D(G) and still maintain this correspondence between edges in D(G) and paths in G.

The inverse property of transitive closure is transitive reduction. Just as the dependency graph of dag G has a maximal number of edges corresponding to paths in G, the transitive reduction of G or the Hasse graph of G contains the minimal number of edges required to maintain paths in G. A *transitive reduction* of directed graph G is a graph H(G) where

(i) there is a directed path from u to v in H(G) if and only if there is a directed path from u to v in G, and

(ii) there is no graph with fewer edges than H(G) that satisfies condition (i).

Figure 2 shows a dag G and its transitive reduction G'. Intuitively, we can describe a Hasse graph as a graph with no short-cuts. If there is a path from u to v containing at least one node different from u or v, then no (u,v) edge exists in the Hasse graph.

The Hasse graph of an directed acyclic graph is unique [1] and the transitive closure of the Hasse graph of a dependency graph is the dependency graph itself. Because we can generate a unique Hasse graph from any dependency graph and a dependency graph from its Hasse graph, we will simplify many of our processes on G by considering only H(G).

2.3 Clans

The central focus of our decomposition algorithm is the clan. Let G be a dependency graph. A subset $X \subseteq G$ is a clan iff for all $x, y \in X$ and all $z \in G - X$,

(a) z is an ancestor of x iff z is an ancestor of y, or

(b) z is a descendant of x iff z is a descendant of y.

An alternate description of a clan depicts it as a subset of nodes where every element not in the subset is related in the same way (i.e. ancestor, descendant or neither) to each member in the subset. Trivial clans include singleton sets and the entire graph.

In Figure 3, sets {2,3,4}, {2,3,4,5}, {1,2,3,4}, and {3,4} are the nontrivial clans. C={2,3,4} is a clan since node 1 is an ancestor of each element of C and 5 is a descendant of each element of C. The set {2,5} is not a clan since 3 and 4 are ancestors of 5 but not ancestors of 2.

We can classify a graph as (i) *primitive* if the only clans in G are ∅, G and singleton sets; (ii) *independent* if every subset is a clan; or (iii) *linear* if for every pair of nodes x and y in the graph, x is an ancestor or descendant of y.

Independent graphs are sets of isolated nodes. Figure 4 is an example of a primitive graph. Clans in a linear graph are sequences of one or more nodes $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$ where for k<i, v_i is an ancestor of v_k. (See Figure 1).

2.4 Graph-grammars

String grammars are a special case of graph-grammars. A string is isomorphic to a linear graph, and in a production of the string grammar, the replacement string is connected to the host string in the same way as the replaced string. For example, if the production rule ab → cde, is applied to the string xabc, the string xcdec is produced. Graphically, Figure 5 shows the original and resulting strings.

In a sequential graph rewriting system or graph grammar, graphs are generated from some initial graph by productions where a subgraph of the host graph, the *mother* graph, is replaced by another subgraph, the *daughter* graph. The main problem of graph grammars is specifying how to embed the daughter graph in the host graph, that is, specifying the edges that should be added to connect the daughter graph to the host graph and determining how edges incident to the mother graph should be modified in the derived graph.

For the graph-grammar we define here, the reconnection rule or embedding rule is *heredity*. An embedding is called *hereditary* if the mother graph consists of a single node and each node in the daughter graph of a production is connected to the host graph in exactly the same way as its mother. More formally, let the mother graph be node u. For each vertex v in the daughter graph, (w,v) is an edge in the resultant graph whenever (w,u) is an edge in the host graph, and (v,w) is an edge in the resultant graph whenever (u,w) is an edge in the host graph. For example if the host graph is the Hasse graph H in Figure 6 and the production maps node x into the graph D, the resulting graph is H'. H" is the Hasse graph of H'.

A *dependency graph grammar* is a system G = (z, P, H) where z is a node called the axiom or start node. P is a set of pairs (v,D) where v is a graph vertex and D is a primitive, independent or linear dependency graph. P represents the set of productions and H is the hereditary rule of reconnection. Let us call a production of this system an *h-production*. Applications of h-productions preserve the properties of dependency graphs. Furthermore, the daughter graph becomes a clan in the resultant graph.

In a dependency graph grammar, all graphs resulting from the productions are also dependency graphs. Development of a similar system for Hasse graphs is desirable. The goal for a Hasse or reduced production system is to define a system where the result of a production is a Hasse graph when G and D are Hasse graphs. Clearly we can find the Hasse graph of a graph derived from a dependency graph grammar, but it is a simplification process to find the reduced graph directly. Define a *reduced hereditary* rule of embedding to be a reconnection rule where each source s of the daughter graph is connected to all parents of the mother node and each sink m of the daughter is connected to all children of the mother node. In other words, for a production (d,D), where S = {sources of D} and M = {sinks of D}, if $(v,d) \in E(G)$, then $(v,s) \in E(G')$ for all $s \in S$ and if $(d,v) \in E(G)$, then $(m,v) \in E(G')$ for all $m \in M$. An *rh-production* is a production where (i) the host graph is a Hasse graph H, (ii) the mother is a vertex of H, (iii) the daughter D is a primitive, linear or independent Hasse graph, and (iv) the embedding rule is reduced heredity.

In an rh-production, the inserted (daughter) graph D becomes a clan in the host graph and the resulting graph H' is a Hasse graph. In Figure 6, H'' is formed directly by an rh-production since edge (a,x) is replaced by edges (a,d) and (a,e) only, and edge (x,c) is replaced by (f,c) and (g,c).

3 Graph Decomposition Algorithm

Our goal is to write an algorithm to parse or equivalently, find a derivation tree for a given dependency graph. This involves finding the right hand sides or daughter graphs of the h-productions. Since each daughter D is a clan in the derived graph, we will look for clans.

Ehrenfeucht and Rosenberg [2,3,4] prove that every dependency graph can be decomposed into (built up from) the three types of graphs: independent, linear, and primitive. They further describe a canonical derivation for a given graph that is unique. A derivation is called *canonical* if an h-production with linear (independent) daughter graph D is never followed directly by another production where D is linear (independent). This restriction in no way affects the class of graphs that can be derived, for any two consecutive productions of the same form can be combined into a single production. Any dependency graph yields a unique decomposition when the decomposition is canonical. The purpose of this paper is to show an algorithm for finding clans of the canonical derivation which labels each clan as primitive, linear or independent.

3.1 Sets of siblings and sets of mates

Because of the identification of a unique Hasse graph with each dependency graph G, we can simplify our discussion by considering only H(G). **From now on all graphs are assumed to be Hasse graphs.**

If a primitive or independent clan were to replace a node using an rh-production, each source of the clan would then have the same parents as the mother node, and each sink of the clan would then have the same children as the mother node. This leads us to look for nodes with the same parents that will be the sources of the clans and nodes with the same children that will be the sinks of the clans. The entire clan will then include the sources, the sinks, and all the nodes "in between." An edge entering a potential clan at a non-source node, or an edge leaving a potential clan at a non-sink node violates the clan definition. In summary we need to find subgraphs of G with the properties:

A. any ancestor of one source of the subgraph is an ancestor of all elements of the subgraph.

B. any descendent of one sink of the subgraph is a descendant of all elements of the subgraph.

C. all children of non-sink elements of the subgraph must be contained in the subgraph and all parents of non-source elements of the subgraph must be contained in the subgraph.

For any set of nodes S, we will denote the sets of parents, children, ancestors, and descendants of S by P(S), C(S), A(S), and D(S), respectively. To simplify many of the equations needed, let F*(S) denote the set of nodes $F(S) \cup S$.

These properties indicate that we should search for groups of nodes with the same parents and groups of nodes with the same children. The first group of nodes we will call siblings and the second group we will call mates. Since any set of siblings is a potential set of sources for a clan, we partition all nodes into sets S where $x,y \in S$ if and only if the set of parents of x is the same as the set of parents of y (x and y are siblings). Note that $\mathcal{S} = \{S\}$ is a partition of the nodes of the graph. Similarly we partition all nodes into sets M where $x,y \in M$ if and only if the set of children of x is the same as the set of children of y (x and y are mates). $\mathcal{M} = \{M\}$ is a partition of the nodes of the graph. By considering only subgraphs with sources from some $S \in \mathcal{S}$ and sinks from some
$M \in \mathcal{M}$ and such that conditions A, B, and C hold, we will find clans. The following theorem gives an alternate characterization of a clan.

Theorem 3.1: Let H be a Hasse graph with $\mathcal{S}$ and $\mathcal{M}$ partitions of the nodes into sets of siblings and sets of mates, respectively. A subgraph of nodes in set G is a clan if and only if

(i) $G = D^*(S) \cap A^*(M)$ for some $S \subseteq S' \in \mathcal{S}$ and $M \subseteq M' \in \mathcal{M}$
(ii) $D^*(S) - (D^*(M) \cup A^*(M)) = \varnothing$
and (iii) $A^*(M) - (D^*(S) \cup A^*(S)) = \varnothing$.

Part (i) corresponds to properties A and B and parts (ii) and (iii) grant the fulfillment of property C. If (ii) is violated, it is said that an *illegal exit* from the clan has occurred. If (iii) is violated, an *illegal entry* into the clan has occurred.

3.2 A Partial Ordering

Clans will be found where the sources of each clan are siblings, and the sinks of each clan are mates. All other nodes in the clan will be descendants of the sources and ancestors of the sinks. We could find all clans by constructing all subsets which use a set $S \in \mathcal{S}$ as source and which use a set $M \in \mathcal{M}$ as sink. This would pair as many as n^2 sets and many of the pairs could not produce clans. For example, in any clan, sinks must be descendants of the sources. If all S's and M's were paired, many would not have this property. To limit the number of matches we must check, we will define a partial ordering on the sets of nodes S and M of G. The partial ordering will then be used to match S's and M's that have potential for being sources and sinks of clans.

Define a binary relation $<_G$ on $\mathcal{S} \cup \mathcal{M}$ by $X <_G Y$ if
(A) $X,Y \in \mathcal{S}$ and $y \in D(X)$ for any node $y \in Y$, or
(B) $X,Y \in \mathcal{M}$ and $x \in A(Y)$ for any node $x \in X$, or
(C) $X \in \mathcal{S}$ and $Y \in \mathcal{M}$ and $D^*(X) \cap A^*(Y) \neq \varnothing$, or
(D) $X \in \mathcal{M}$ and $Y \in \mathcal{S}$ and $y \in D(x)$ for any node $y \in Y$ and any node $x \in X$.

The relation $<_G$ is non-reflexive, antisymmetric and transitive, and the four cases represent all possibilities for X and Y on $\mathcal{S} \cup \mathcal{M}$. In cases (A) and (B), where both X and Y are in $\mathcal{S}$ or both are in $\mathcal{M}$, $X \cap Y = \varnothing$ since $\mathcal{S}$ and $\mathcal{M}$ partition the nodes. In (A) all nodes of Y have the same ancestors and if one $y \in Y$ is a descendant of any element of X all elements of Y are descendants of X. The case is well-defined since it is impossible for $y \in D(X)$ and $x \in D(Y)$ in an acyclic graph (otherwise all elements of Y are descendants of X and all elements of X are descendants of Y. Case (B) is the dual of case (A) where $\mathcal{S}$ is replaced by $\mathcal{M}$ and descendant by ancestor.

In case (D), if one element of Y is a descendant of one element of X, all elements of Y are descendants of all elements of X since all elements of Y have the same set of parents (and hence ancestors) and all elements of X have the same children (and hence descendants).

Case (C) is defined in a different manner since w e do not want to treat S and M in perfectly symmetrical ways. Clans may exist where some elements are both sources and sinks. The independent clan is such an example. In these cases, $S \subset M$ or $M \subset S$ or $S = M$. when $S \cap M \neq \emptyset$ we want to insure that $S <_G M$ so that S and M will be paired for finding this clan.

3.3 Graph of the partial order

As with any partial ordering, $<_G$ may be represented by a directed graph which we will call the order graph, R. The nodes of R are the elements of S and M. For $Z_i \in S \cup M$, if $Z_i <_G Z_j$ then there is an edge from Z_i to Z_j. Note that since the ordering $<_G$ is antisymmetric, R is acylic, and since the ordering $<_G$ is transitive, R is transitive. Hence R is itself a dependency graph and has a corresponding Hasse graph, H(R). If a node is denoted by S, it is an element of $\mathcal{S}$, and if it is denoted by M, it is an element of $\mathcal{M}$.

H(R) itself is an interesting structure. The source nodes of G form a sibling set S with no ancestors and this set is a single source for H(R). The dual set of mates from M is the set of sinks of G and gives H(R) its single sink. It is always the case that if $S_i <_G S_j$ then there is some element M so that $S_i <_G M <_G S_j$ (see Lemma 3.6 in [5]) and the reduced order graph is a bipartite graph where all edges connect elements of $\mathcal{S}$ to elements of $\mathcal{M}$. By the construction of H(R), when there is a path from S_i to M_j, then $S_i <_G M_j$ and it is possible that there is a clan with source from S_i and sink from M_j. since paths in H(R) represent sets of nodes that are related by the ancestral relationship, paths in H(R) will be used to select the elements of $\mathcal{S}$ and $\mathcal{M}$ that yield the clans of G.

Let $Y_1, Y_2, \ldots Y_n$ and $Z_1, Z_2, \ldots Z_m$ be two paths in H(R). We say the paths are *disjoint* if $[\cup_i (Y_i, Y_{i+1})] \cap [\cup_j (Z_j, Z_{j+1})] = \emptyset$. In other words, paths are disjoint if their edges are unique.

The parsing algorithm builds H(R) and uses elements along its paths to form the clans. any (S,M) pair on a path from S to M is the potential source, sink pair for a clan. To insure that every pair (S,M) that yields a clan is inspected, find all disjoint paths in H(R) of maximal length. Short disjoint paths may not include all possible (S,M) pairs. An algorithm that selects the proper paths is a combination of depth-first and breadth-first searching algorithms. Starting at the source S, find a path by following adjacent edges in a depth-first manner until M is on the path. Return to S; choose another edge emanating from S as in a breadth-first algorithm and define another path by following adjacent edges until there are no adjacent edges no yet placed on a path. Repeat until all edges emanating from S have been placed on a path. Iterate the process starting at the children of S.

Consider the example graph given in Figure 7. The notation used to define sets of $\mathcal{S}$ and $\mathcal{M}$ will be strings where the first character is S or M signifying that the set

is in $\mathcal{S}$ or $\mathcal{M}$. The remaining characters give the set contents. For example Sabc is a sibling set containing nodes a,b, and c. In Figure 7 $\mathcal{S}$ = {Sabc, Sde, Sf, Sg, Shi, Sjk} and $\mathcal{M}$ = {Mabc, Md, Me, Mfg, Mh, Mijk}. The order graph is shown in Figure 9 and one possible choice of paths is:

Path 1 = Sabc Mabc Sde Md Sf Mfg Shi Mh Sjk Mijk

Path 2 = Sde Me Sg Mfg

Path 3 = Me Sf

4 Parsing A Dependency Graph

4.1 Algorithm Input and Output

Input to the algorithm is a connected dependency graph G. The output is a subset of all clans from which the canonical derivation can be found. A subset of clans is sought since the total number of clans in a dependency graph may be exponential. For example, if G is the independent graph on n nodes, (i.e. the set of n isolated nodes), then every subset of nodes is a clan and there are 2^n clans of G. If G is a linear graph on n nodes, every sequence of k nodes where $k < n$ is a graph, and the total number of nodes is $\Sigma(n-k+1) = O(n^2)$. Since the canonical derivation involves only O(n) clans, an appropriate subset of all clans must be found.

The clans we find will be classified as primitive, independent or linear according to their "quotient" graph. Let F be a clan of G and $\{F_1, F_2, ..., F_n\}$ be a partition of F where each F_i is a clan of G. Then the *quotient graph* of F, denoted by $F/F_1...F_n$ is the graph where the nodes are $F_1...F_n$ and the edges are (F_i,F_j) whenever there exist $x \in F_i$ and $y \in F_j$ such that $(x,y) \in E(F)$. The algorithm finds clans F containing at least 2 nodes for which one of the following holds:

1. F/F_1F_2 is the linear graph of two nodes.
2. $F/F_1...F_n$ is a maximal independent graph.
3. $F/F_1...F_n$ is a primitive graph.

In the graph of figure 8a for example, F_1 = {a,b,c} and F_2 = {d,e} are two clans that partition the entire graph F. The quotient graph is the linear connection of F_1 and F_2 as shown in Figure 8b.

4.2 Algorithm Description

The algorithm consists of two distinct parts: the preprocessing section and the decomposition section. The preprocessing section finds relevant features of the graph that are used by the decomposition section. It first computes the Hasse graph of G, H(G), and then uses H(G) in the rest of its processing. It partitions the nodes of H(G) into sets of siblings $\mathcal{S}$ and sets of mates $\mathcal{M}$. It then uses the ordering $<_G$ on $\mathcal{S}$ and $\mathcal{M}$ to

compute the order graph. After finding the Hasse graph of the order graph, it finds the disjoint paths of the graph.

The decomposition section uses two distinct ways of finding the clans and each way scans a different list of the sets of $\mathcal{S}$ and $\mathcal{M}$. The first finds independent and primitive clans and scans the paths of the order graph. The second finds the linear clans by scanning a list called the completed list created by the first process.

To find the independent and primitive clans, we search each path in the order graph for two sets S and M which are adjacent and where $S <_G M$. Inspect $F=D^*(S) \cap A^*(M)$. If there are no edges leaving F from non-sink nodes nor entering F to non-source nodes, F is a clan. If F is not connected, it is independent and each of its connected components is a primitive or linear clan. If F is connected, it is either a primitive or linear clan.

If F is not a clan, it may be a subgraph of a clan. This larger clan may be found by using source $S_k \leq_G S$ and sink $M_i \geq_G M$. To find this comparison, exchange S and M in the path and again scan the path for adjacent pairs $S_h < M_j$. If the graph contains nested clans, testing adjacent path elements first will insure that innermost clans for each path will be found first.

In the process of finding independent and primitive clans, we prepare a list to be used by the linear clan checker to test for linear clans. A linear connection is a set of edges from the sink of one clan, C, to the source of the next clan, C', or a single edge from one node n to the next m where m is the only child of n and n is the only parent of m. In either case, the source of C' must be the same as the children of the sink of C and the sink of C must be the same as the parents of the source of C'. To check for these connections, we form a list of clans and nodes that are potentially linearly connected and then check for linearity. The list contains singleton sets of $\mathcal{S}$ and $\mathcal{M}$ and sets S and M where S is the source of some clan F and M is the sink of some clan F'.

The potential sources and sinks (including singletons) of linear clans are placed on the "completed list" (CL) in the same order in which they were on the order list. Then this list is scanned for adjacent sets S and M where $M <_G S$, $C(M) = S$, and $P(S)=M$.

4.3 The Algorithm

Input: A connected dependency graph G

Output: Each clan of G and an associated identification of primitive, independent, or linear.

Preprocessing

i. Compute the Hasse graph H(G)

ii. For each node $x \in H(G)$ find: the sets of parents P(x), children C(x),ancestorsA(x),and descendants D(x).

iii. Partition nodes n_i into sets $S_1, S_2, ..., S_n$ where $n_i, n_j \in S_k$ if and only if $P(n_j) = P(n_i)$. $\mathcal{S} = \{S_1, S_2, ..., S_n\}$

iv. Partition nodes n_i into sets $M_1, M_2, ..., M_m$ where $n_i, n_j \in M_k$ if and only if $C(n_i) = C(n_j)$. $\mathcal{M} = \{M_1, M_2, ..., M_m\}$

v. For all S_i and M_j in partitions $\mathcal{S}$ and $\mathcal{M}$, sort by the rules

a. $S_i <_G M_j$ if $S_i \cap A(M_j) \neq \emptyset$

b. $M_j <_G S_i$ if $s_i \in D(M_j)$ for any s_i in S_i.

vi. Define the order graph R=(V,E) of the relation $<_G$ by $V = \{ Z_i \mid Z_i \in \mathcal{S} \text{ or } Z_i \in \mathcal{M}\}$, $E = \{(Z_i, Z_j\} \mid Z_i <_G Z_j\}$.

vii. Find the Hasse graph for R defined in step vi. Call this graph the order graph.

viii. Partition the edges of the order graph into distinct maximal paths P_i.

Find factors

For each path $Z_1 < ... < Z_k$ in the order graph do

1. For each pair Z_i, Z_{i+1} where $Z_i \in \mathcal{S}$ and $Z_{i+1} \in \mathcal{M}$ do

1.1 If Z_i (Z_{i+1}) is a singleton, remove it from the path and place it in location i (i+1) on the completed list. Consider the next Z_i,Z_{i+1} pair.

1.2 Otherwise $S \Leftarrow Z_i$, $M \Leftarrow Z_{i+1}$, $F = D^*(S) \cap A^*(M)$.

1.2.1 For each connected component F_k of F with source S_k and sink M_k check for illegal entry or illegal exit using formulas (ii) and (iii) of section 3.1. Return as a primitive clan each component with legal entry and legal exit. Return the union of the primitive clans as an independent clan, F. If F is connected with legal exit and entry, F is a primitive clan.

1.2.2 If there are no legal components in step 1.3, exchange Z_i and Z_{i+1} in the path and consider next Z_i, Z_{i+1} pair.

1.2.3 If the source of F is Z_i, identify Z_i with F, remove Z_i from the path, and place it in the proper location on the completed list. If the sink of F is Z_{i+1}, identify Z_{i+1} with F, remove it from the path and place it in the proper location on the completed list. Otherwise exchange Z_i and Z_{i+1} on the path.

2. Check the completed list for linear clans. For each adjacent pair Z_i, Z_{i+1} on the list where $Z_i \in \mathcal{M}$ and $Z_{i+1} \in \mathcal{S}$, if $C(Z_i) = Z_{i+1}$ and $P(Z_{i+1}) = Z_i$, then clans or singletons represented by Z_i and Z_{i+1} are linearly connected.

Example:

When we apply the algorithm to the graph of Figure 7, the resulting clans and their classification are

C1 = {a,b,c}I; C2= {j,k}I; C3 = {d,e,f,g}P;
C4 = {i,h,j,k}I; C5 = {C1, C3}L;
C6 = {C3, C4}L and C7 = {h,C2}L

The parse tree is shown in Figure 10.

Both the preprocessing and the clan finding portions of the algorithm require a time of $O(n^3)$. The time bound is met by implementing ordered sets and by using the properties of dependency graphs to find connected components.

5 Significance

The work of the paper exhibits a good (polynomial) parsing algorithm for a very general graph-grammar. Although graph-grammars have existed since the early 1970's, they have not been widely used primarily because no good parsing algorithms have existed for them. Just as the importance of string grammars has come through the discovery of LR parsers, an algorithm such as the one presented here may lead to important uses of graph-grammars.

An application of this work in the area of serial to parallel code translation is being investigated [6] and [7]. In this application, a dag represents the data flow graph of a program. Nodes represent executions and edges represent data dependencies. Independent clans show where nodes can be executed in parallel and linear clans show where serialization is necessary. The parse tree represents the degree to which a computation can be made parallel. The leaves represent the highest degree of parallelism or finest grains. Internal nodes represent the aggregation of their children into coarser grains. Each level from tree bottom to top represents larger grains and lower communication costs. A bottom-up tree traversal will lead to a proper balance between a high degree of parallel execution and low communication costs.

Bibliography

1. A. V. Aho, R. Garey, and J. D. Ullman, Transitive Reduction of a Directed Graph, *SIAM J. Comput.* **1** (1972), 131-137.

2. A. Ehrenfeucht and G. Rozenberg, Theory of 2-Structures, Part I: Clans, Basic Subclasses, and Morphisms, *Theoretical Computer Science 70* (1990), 277-303.

3. A. Ehrenfeucht and G. Rozenberg, Theory of 2-Structures, Part II: Representation Through Labeled Tree Families, *Theoretical Computer Science 70* (1990), 305 -342.

4. A. Ehrenfeucht and G. Rozenberg, Primitivity is Hereditary for 2-Structures, *Theoretical Computer Science 70* (1990), 343-358.

5. C. Mccreary, "An Algorithm for Parsing a Graph-Grammar," Ph.D. Dissertation, University of Colorado, Boulder, 1987.

6. C. McCreary and H. Gill, Automatic Determination of Grain Size for Efficient Parallel Processing, *Communications of the ACM*, **32** (1989).

7. C. McCreary and H. Gill, Efficient Exploitation of Concurrency Using Graph Decomposition, *Proceedings of the International Conference on Parallel Processing* , (1990).

8. D. M. Moyles and G. L. Thompson, Finding a Minimum Equivalent of a Digraph, *Journal of the ACM*, **16** (1969), 455-460.

Figure 1: A linear graph and its dependency graph

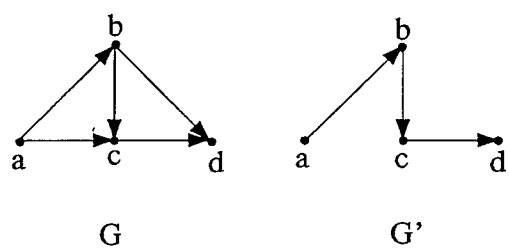

Figure 2: A dag and its Hasse graph

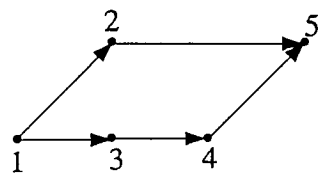

Figure 3: Clans

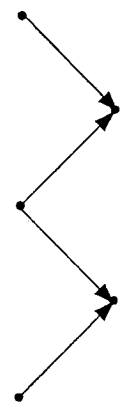

Figure 4: Primitive Graph

Figure 5: String Production

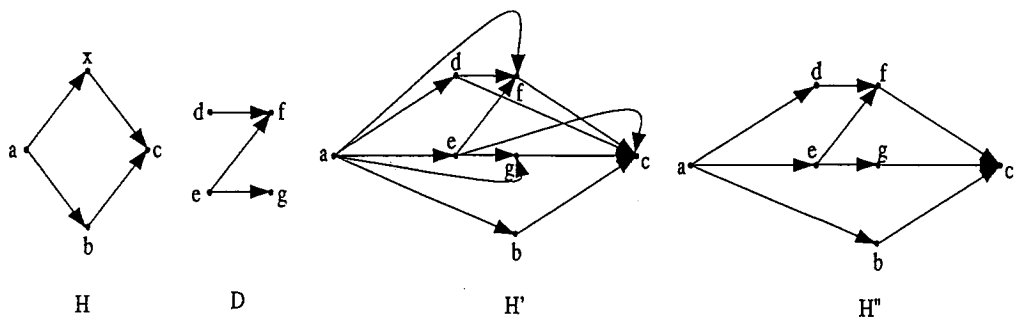

Figure 6: h-production and reduction

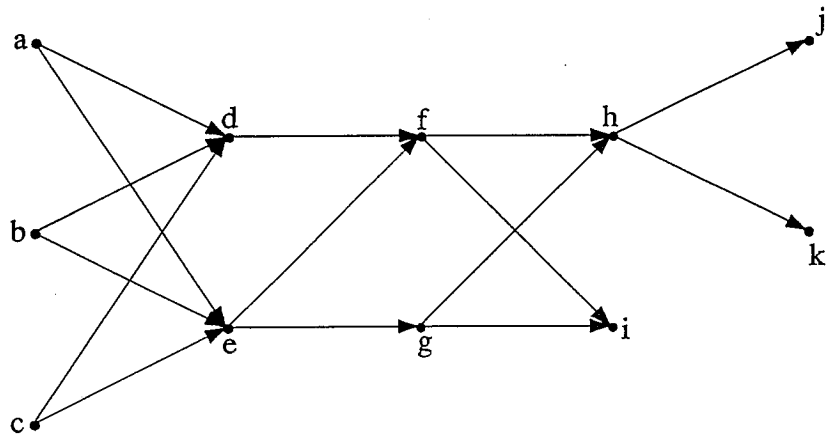

Figure 7: Example Graph

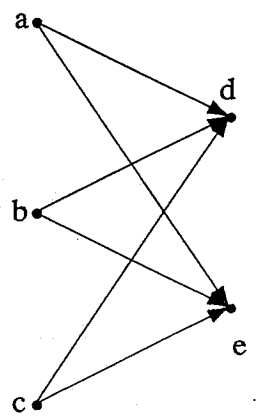

Figure 8a: Graph

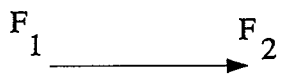

Figure 8b: Quotient Graph

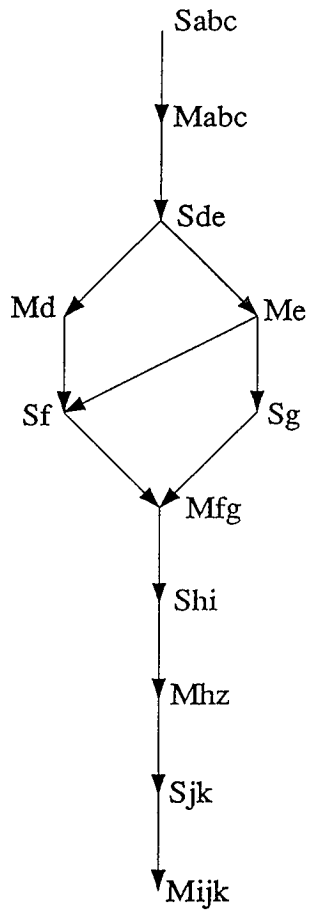

Figure 9: Order Graph

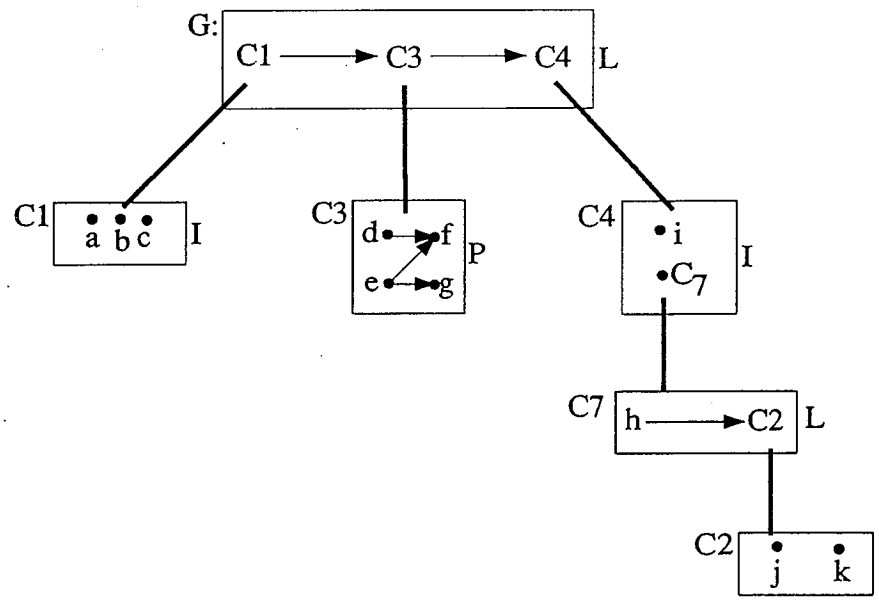

Figure 10: Parse Tree