McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Generating Irregular Partitionable Data Structures

Prakash Panangaden
Clark Verbrugge

ACAPS Technical Memo 100
March 15, 1996

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

A fundamental problem in parallel computing is partitioning data structures in such a way as to minimize communication between processes while keeping the loads balanced. In the present paper we propose a linguistic mechanism, based on ideas from graph grammars, that allows one to generate graphs that can be easily partitioned. We give a precise definition of what it means to be easily partitionable and show that all the graphs that we generate can indeed be easily partitioned. We illustrate the expressiveness of our scheme by examples and establish limits to the scheme by showing that all the graphs that we generate have bounded tree-width.

Contents

List of Figures

1 Introduction

A fundamental problem in parallel computing is partitioning data structures in such a way as to minimize communication between processes while keeping the loads balanced. The problem is particularly acute when the underlying data structures are irregular, pointer-based structures. The vast majority of the research has focussed on algorithmic aspects of partitioning [6, 26, 36, 43] or on heuristics [16, 25, 52]. In the present paper we propose a linguistic mechanism, based on graph grammars, that allows one to generate graphs which can be easily partitioned *automatically*; i.e. once the family of graphs is specified in our formalism the partitioning can be done without further effort by the user. We give a precise definition of what it means to be easily partitionable and show that all the graphs that we generate can indeed be easily partitioned. We illustrate the expressiveness of our scheme by describing examples of families of graphs that we can generate and we also describe the limitations of our method by proving that we only generate graphs that have bounded tree-width.

We have identified a general form of graph grammar which is suitable for developing "partitionable" graphs. Using any grammar in this class, a family of graphs can be specified all of which have attractive properties for graph partitioning. These bounds are established by relating each generated graphs to its (modified) derivation tree from the grammar; by partitioning the tree we partition the graph, and the communication and balancing bounds then follow from the grammar specification.

Algorithms on unstructured, dynamic data are not easy to parallelize; the unknown size and "shape" of the data structure makes dividing the work equally and efficiently among processors quite difficult. A great deal of research into parallelism has therefore concentrated on finding "good" data partitionings. For the most part these efforts have been directed toward data that is regular [1, 7, 53]. Most such work involves using information extracted from the program to deduce constraints on array indices and use various techniques (e.g. linear programming) to solve these constraints. Such approaches, while very interesting and powerful, will shed no light on the problem of partitioning dynamic data structures.

Our approach is to define a type of graph-grammar that is used to *generate* structures which can be partitioned automatically. Naturally, if data structures are restricted to being amenable to partitioning, then not all data structures will be expressible. The grammars we define, though, are general enough to express a wide variety of common data structures, trees of course, threaded trees, trees where the leaves have sibling pointers, structured compiler control flow graphs, and other examples described below. The fundamental point is that though the graphs that we generate are more general than trees they have *bounded tree width* [47]; thus they are "tree-like" at some deeper level.

In designing a data structure one usually thinks about how the data structure will be updated. This usually translates into grammar rules for generating the structure. In our experience, there was little overheard in developing a grammar for data structures of interest.

In the following section we formalize the notion of "partitionability." Section 3 develops the grammars in which we are interested. In section 4 we prove that weighted k-ary trees can be partitioned with guaranteed bounds on load-balancing. This result is used in section 5, where we relate the derivation tree of any graph produced by our grammars to the actual graph. By partitioning this weighted tree we also partition the graph, and the bounds on cost and balance of the graph partitioning follow from the tree partitioning; this is our main result. Section 6 illustrates the expressibility of our formalism; we show several different grammars defining several different graphs commonly used in computer science applications. Contrasting this are the results in section 7; all of the graphs we generate have bounded "tree-width."

2 What is a Good Partition?

It is a platitude to say that a "good" partition should not cut too many links. We need a quantitative notion of what this means. The paradigmatic example of an easily decomposed structure is a tree and an easily partitioned structure should be, roughly speaking, as easy to partition as a tree. Thus, we define a *strong* partitionability through the following series of definitions.

Definition 2.1 *A* p-**partitioning** *of a graph* $D = (V, E)$ *is an equivalence relation* $\cong$ *on the vertices of* D *such that there are exactly* p *equivalence classes.*

A p-partitioning induces a communication cost from a partition to the rest of the graph (and of course to any other partition). For a given partitioning P let V_i be the set of nodes associated with the i^{th} piece.

Definition 2.2 *The communication cost of* V_i *is defined as:*

$$Cost(V_i) \;=\; |\{(v, v') \in E |\; v \in V_i \;\wedge\; v' \notin V_i\}|$$

In parallel computing, processors are not usually viewed as a resource fixed at compile-time. Accordingly, partitionability should be a property which provides bounds on communications costs no matter how many partitions are envisaged.

Definition 2.3 *Given some function* f *of* n*, an* f-**partitionable** *graph of* n *vertices is a graph that can be partitioned into* p *pieces of size* $(n/p) \pm c$ *for any* $1 \le p \le n$ *and some constant* c*, such that the communication cost of any piece is no more than* $f(n)$*.*

Arbitrary, undirected graphs without loops or multiple edges are trivially n^2-partitionable, since each node in a partition of $n/p \pm c$ nodes can connect to no more than n other nodes. Graphs with a bound k on the degree of each node are kn-partitionable.

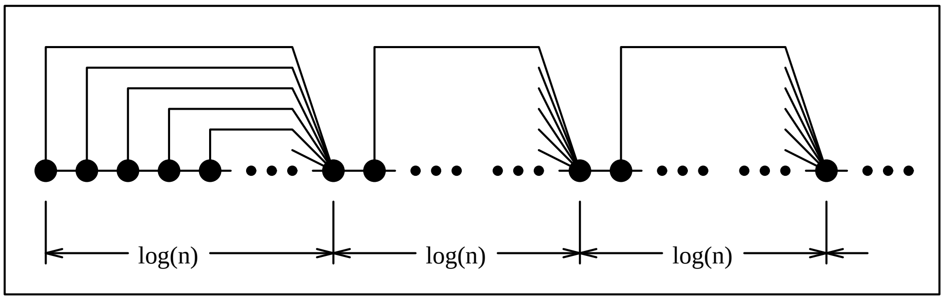

Figure 1: A reasonably partitionable graph.

$O(1)$-partitionable graphs are clearly ideal. Unfortunately, this category only includes lists and small variations; for instance, the class of trees with bounded fanout k has a lower bound on communication cost of $\Omega(k \log(n)/\log(k))$ (see the discussion of theorem 3.2 in [9]). Since trees are certainly a data structure we would like to represent, any general partitioning strategy will have a similar lower bound.

Square grids of $\sqrt{n} \times \sqrt{n}$ vertices form another interesting class of graphs, ones which have a lower bound on partitionability of $\sqrt{n}$. Dense structures such as these, though, are often more efficiently represented by arrays than by pointer-based structures. Moreover, the relatively high lower bound on partitionability makes it difficult to include them in any generalized approach without overly diluting the efficacy of the partitioning algorithm/approach when applied to more partitionable data structures.

If we are to quantitatively evaluate partitioning it is necessary to commit ourselves to some hard distinction as to what is reasonable and what is not. Certainly trees are necessary, and with simple extensions can be made to encompass the bulk of computer science data structures. Similarly, grids are certainly better dealt with using array-based methods. The fundamental dichotomy is therefore embodied in the following definition:

Definition 2.4 *Let G be an f-partitionable graph. Then G is* **reasonably partitionable** *if $f \in O(\log(n))$.*

Remarks: Almost all data structures fall into this category, other than direct representations of densely-connected data (such as grids or triangulations). Obviously, this also excludes any graph with a node having degree in $\omega(\log(n))$; for example, a tree with each leaf connected to the root is not $k \log(n)$-partitionable for any constant k, since some partition piece must include the root (of degree n). An example of a reasonably-partitionable graph is in figure 1; here a linked list of nodes is divided into $n/\log(n)$ pieces each of length $\log(n)$, where the head of each piece is connected to every node in its piece.

Partitionability is relatively simple to define. It is considerably harder to detect whether a graph is reasonably partitionable, or to find a method for ensuring the intended graphs of an application are reasonably partitionable. This, however, is exactly what we do; the grammars we develop in the rest of the paper have the property that every graph they generate must be reasonably-partitionable.

3 Dangling Graph Grammars

Graph grammars in general are rewrite systems. Given a graph, a graph grammar specifies how to locally change the graph into another graph, based on the existence of a certain subgraph. The rules which govern this transformation are termed *productions,* and the graph to which the productions are (initially) applied is the *axiom.* This process is usually iterated, generating a sequence of graphs, which collectively constitute the *language* of the grammar.

3.1 Dangling Graphs

The usual definition of labelled graphs involves sets of nodes, edges, labels and functions associating edges with nodes, nodes with labels, and edges with labels. The nature of graph partitioning, which requires "splitting" edges to form partitions, makes it more convenient to use so-called *dangling graphs.* The essential idea is to form the graph from nodes and *half-edges,* or edges associated with just a single node:

Definition 3.1 *A* **dangling graph**, *D is an 8-tuple $(V, E, \nu, \phi, \psi, \Sigma_V, \Sigma_E, C)$, where:*

V *is a set of* vertices *(or* nodes*),*

E *is a set of 1/2-edges,*

$\nu : E \rightarrow V$ *is an injective function returning the vertex associated with a given 1/2-edge.*

Σ_V *is a finite set of node labels,*

Σ_E *is a finite set of 1/2-edge labels,*

$\phi : V \rightarrow \Sigma_V$ *is a node labelling function,*

$\psi : E \rightarrow \Sigma_E$ *is a 1/2-edge labelling function, with the property that no two 1/2-edges connected to the same vertex have the same label:*

$$\forall\ e, e' \in E,\ \nu(e) = \nu(e') \Rightarrow \psi(e) \neq \psi(e')$$

$C \subseteq E \times E$ *is a connection relation between 1/2-edges, such that:*

$$\forall\ (e, e') \in C,\ \neg\exists\ (e, e'') \in C \ for\ e'' \neq e',\ and\ (e', e) \in C$$

In other words, C describes the connected pairs of 1/2-edges, and is symmetric.

Remarks: Like most definitions of graph, a dangling graph is based on nodes, and connections between them. In the above definition, the connections are managed by a connection relation; each connection between two nodes is formed from two "1/2-edges," where each such 1/2-edge is individually associated with a node through the ν function. The connection is then actually established by pairing 1/2-edges in the connection relation C. Any 1/2-edge not involved in a connection relation is considered a *dangling edge* (hence the moniker). More formally, the set of dangling edges of a dangling graph D (described as above) is given by a function Ξ, where:

$$\Xi(D) = \{e \in E \mid \neg\exists\, e' \in E, (e, e') \in C\}$$

We are concerned with *node and 1/2-edge*-labelled dangling graphs. Thus, there is an alphabet for both (Σ_V and Σ_E), and functions to map each node or 1/2-edge to a node or half-edge label, ϕ and ψ respectively. Note that each *1/2-edge* has a label, including dangling ones, and thus each connection between nodes will have two labels, one for each 1/2-edge forming the connection.

The *degree* of a vertex n in a dangling graph D is defined in the same way as for regular graphs; $\mathrm{Degree}(n) = |\{e|\ \nu(e) = n\}|$. If there exists a natural number k such that

$$\forall\ n \in V, degree(n) \leq k$$

then D is called a *k-bounded dangling graph.* It should be noted that since the set of 1/2-edge labels, Σ_E is finite, and no two 1/2-edges attached to the same vertex have the same label, all dangling graphs as described above are already $|\Sigma_E|$-bounded.

Bounded-degree dangling graphs are meant to model doubly-connected data structures, as they might be found in a procedural language like C. Each vertex corresponds to a data structure with a bounded number of pointers, and is attached to other vertices by a two-way connection, corresponding to two data records/nodes having individually-named pointers directed at each other. Dangling edges, 1/2-edges not involved in a connection relation, can then be viewed as nil-pointers. To convert a dangling graph to a "regular" graph we merely dispose of the dangling edges, a process known as *trimming.*

Definition 3.2 *A* trimmed *dangling graph is a dangling graph with the dangling edges removed: if D is a dangling graph then the trimmed version is given by the function ξ, where* $\xi(D) = D[E_D - \Xi(D)/E_D]$.

A graph grammar that operates on the domain of node and edge-labelled dangling graphs is termed a *dangling graph grammar*. Such grammars form the basis of our method of generating partitionable graphs.

3.2 Productions

Productions are rules which define ways in which a given dangling graph can be changed.

Definition 3.3 *A* **production** *is a pair of dangling graphs, a (connected)* **source** *and a* **target,** *along with a partial mapping between dangling edges. If S and T are dangling graphs, then (S,T,δ) is a production if both $\delta : \Xi(S) \rightarrow \Xi(T)$ and $\delta^{-1} : \Xi(T) \rightarrow \Xi(S)$ are partial functions.*

Intuitively, a production is pattern-matched with the graph according to its source. When a matching subgraph is found, that subgraph is excised from the graph and the target is inserted in its stead. How the target is connected to the graph is specified by the partial mapping δ. This sequence of steps can be described formally using the following definitions:

Definition 3.4 *A dangling graph S is a* subgraph *of another dangling graph D if:*

$$\begin{array}{llll} V_S \subseteq V_D & E_S \subseteq E_D|_{V_S} & \nu_S = \nu_D|_{E_S} & \Sigma_{S,V} \subseteq \Sigma_{D,V} \\ \Sigma_{S,E} \subseteq \Sigma_{D,E} & \phi_S = \phi_D|_{V_S} & \psi_S = \psi_D|_{E_S} & C_S \subseteq C_D|_{E_S} \end{array}$$

Remarks: The subgraph relation is as one might expect; one defines a subset of the nodes, 1/2-edges and connection relations, and restricts the various functions to these subsets. A more constrained form of subgraph is one where one must include *all* 1/2-edges of each node included in the subgraph:

Definition 3.5 *A* complete subgraph *of a dangling graph D is a subgraph D of D', such that:*

$$\forall v \in V',\ \exists e \in E.\ \nu(e) = v \ \Rightarrow e \in E'$$

A complete subgraph D' is a complete strict subgraph *if $D' \neq D$. In symbols, $D' \subseteq_c D$ and $D' \subset_c D'$ respectively.*

Any subgraph or complete subgraph is a partition of the containing graph, and the number of connections from the subgraph to the rest of the graph is the cost associated with that partition.

Definition 3.6 *Let G_1 and G_2 be disjoint subgraphs of some dangling graph G. Then the* connectivity *of G_1 and G_2 is the number of connection relations linking vertices in G_1 with vertices in G_2. If:*

$$CSet(V_1, V_2) = \{(e, e') \in C |\ (\nu(e) \in V_1 \ \wedge\ \nu(e') \in V_2)\ \vee (\nu(e) \in V_2 \ \wedge\ \nu(e') \in V_1)\}$$

then the connectivity of G_1 and G_2 is given by: $Con(V_1, V_2) = |CSet(V_1, V_2)|/2$. We will sometimes express this as $CSet(G_1, G_2)$, or $Con(G_1, G_2)$ respectively.

There is a natural order on dangling graphs, similar to the usual (subgraph) ordering on regular graphs:

Definition 3.7 *If D_1, D_2 are two dangling graphs, then $D_1 \sqsubseteq D_2$ iff there exist two label-preserving injections, $\alpha : V_1 \longrightarrow V_2$ and $\beta : E_1 \longrightarrow E_2$, such that:*

$$\begin{array}{rcl} \forall e \in E_1,\ \nu_1(e) = v & \Rightarrow & \nu_2(\beta(e)) = \alpha(v) \\ (e, e') \in C_1 & \Rightarrow & (\beta(e), \beta(e')) \in C_2 \end{array}$$

And if $D_1 \sqsubseteq D_2$ and $D_2 \sqsubseteq D_1$, then $D_1 \equiv D_2$. In this latter situation, α and β would be bijections.

This ordering on graphs and the complete subgraph relation can be combined to formalize what it means for a given graph D' to be "in" another graph D, even if D' is not actually a subgraph of D:

Definition 3.8 *A dangling graph D'* occurs *in another dangling graph D if there exists $D'' \subseteq_c D$, such that $D'' \equiv D'$. The graph D'' is then the* occurrence *of D' in D. The set of all such occurrences is given by the function $Occurs(D', D)$.*

Finally, we can now define how productions are used to rewrite the given graph:

Definition 3.9 *The* application *of a production $\rho = (S, T, \delta)$ to a dangling graph G involves locating an occurrence, S' of S within G, and replacing S' with (a copy[1] of) T. The function δ describes how to modify the connection relation so T is embedded in $G - S'$, utilizing only the connections in $CSet(S', G - S')$. A production ρ then* derives *a dangling graph H from a dangling graph G if ρ can be applied to G, and H is the result once dangling edges are suitably replaced.*

Assuming a production $\rho = (S, T, \delta)$, a dangling graph G to which ρ applies, an image, S' of S in G, and that the 1/2-edges and vertices of T are disjoint from G, the derived graph H can be defined as follows:

$$\begin{array}{lll} V_H = (V_G - V_{S'}) \cup V_T & E_H = (E_G - E_{S'}) \cup E_T & \nu_H = \nu_G|_{E_G - E_{S'}} \cup \nu_T \\ \Sigma_{H,V} = (\Sigma_{G,V} \cup \Sigma_{T,V}) & \Sigma_{H,E} = (\Sigma_{G,E} \cup \Sigma_{T,E}) & \phi_H = \phi_G - \phi_{S'} \cup \phi_T \\ \psi_H = \psi_G - \psi_{S'} \cup \psi_T & & \end{array}$$

The connection relation is somewhat more complicated; if

$$R = \{(e, e') | \; \exists e''. \; (e, e'') \in CSet(V_{S'}, V_G - V_{S'}) \wedge \delta(e'') = e'\}$$

and $\widehat{R}$ is the symmetric closure of R, then

$$C_H = C_G - C_{S'} - CSet(V_{S'}, V_G - V_{S'}) \cup C_T \cup \widehat{R}$$

Generally, the derivation of H from G will be designated by a single arrow subscripted by the production used: $G \rightarrow_\rho H$, and an n-step derivation using a set of productions Υ by $G \xrightarrow{n}_\Upsilon H$. The transitive closure is of course $G \xrightarrow{*}_\Upsilon H$.

Remarks: An application involves locating an occurrence matching the source of the production, removing the occurrence, and attaching a distinct copy of the target by reassigning connection relations involving dangling edges of the occurrence to dangling edges of the (copy of

[1]In order to simplify concepts and notation, where safe we ignore the distinction between the "template" T and the copy of T actually embedded into G.

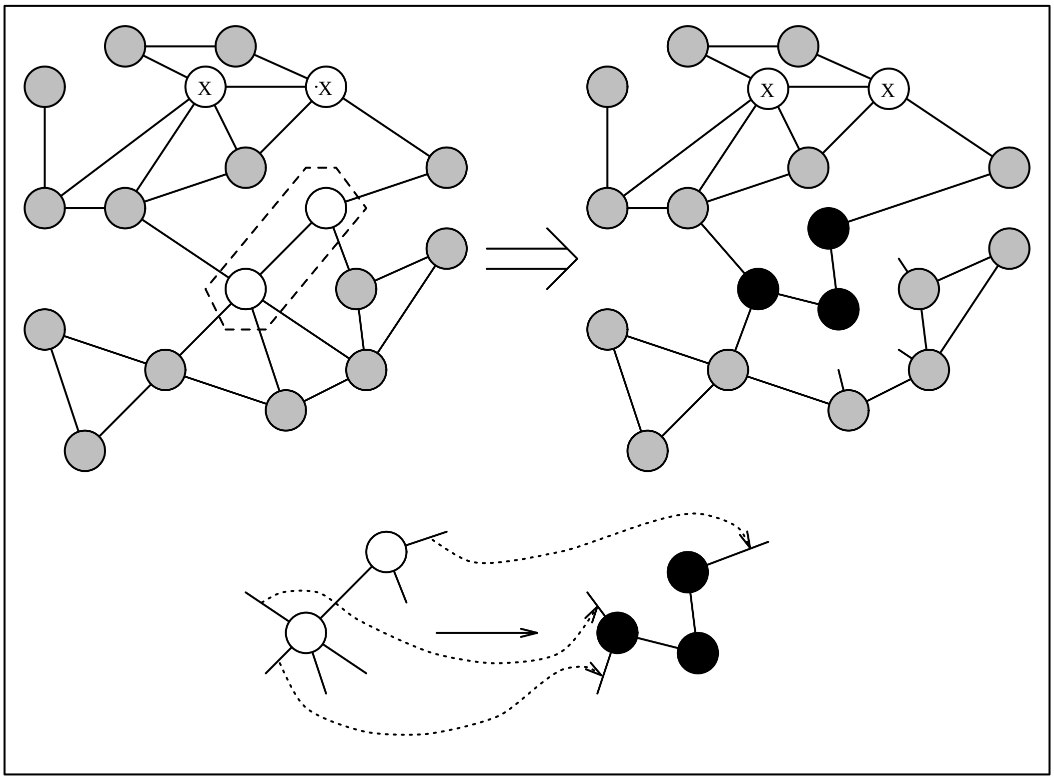

Figure 2: A production (bottom) is applied to a graph.

the) target. There are restrictions on the occurrence—the pattern matching of the source graph must result in a label and structure-preserving bijection h between the nodes and 1/2-edges in the source graph and the nodes and 1/2-edges in its *occurrence* in the graph. As well, if a node in the graph is included in the occurrence, then there must be corresponding matches in the source for *every* 1/2-edge attached to that node.

An example of a production being applied is shown in figure 2; the input graph (axiom) is on the top left, the output is on the top right, and the production is shown on the bottom. Dotted arrows indicate the δ mapping for the production, and the the region enclosed on the input graph is the occurrence being rewritten. Node labels are illustrated by colour (shade), but 1/2-edge labels are not shown. Note that the other two white nodes (marked with x's) cannot be rewritten by this production; even if all labels matched, they do not form an exact image of the source graph (both x-marked nodes have degree 4, whereas the source requires one node with degree 4 and one with degree 3).

Once the occurrence is located, the nodes and 1/2-edges of the occurrence are removed and a distinct copy of the target graph is inserted. If within the graph a dangling edge e of the occurrence is paired with some other dangling edge e' to form a connection $c = e \times e'$, then the 1/2-edge designated by δ of the corresponding 1/2-edge of the source graph, $\delta(h(e))$, is substituted into c in place of e. If $\delta(h(e))$ is undefined for e, the connection relation c is discarded. In the example in figure 2, three connection relations are transferred from the source to the target graph (indicated by dotted arrows), and any connection involving the other three 1/2-edges are deleted by the rewrite.

Within a single derivation, because of the restrictions on how the target is imbedded, the number of connection relations linking the imbedded target to the rest of the graph can be no more than the number of connection relations linking the occurrence to the rest of the graph. This property will prove critical to partitionability:

Proposition 3.1 *Given a production $\rho = (S, T, \delta)$ and a graph G to which ρ applies, $G \rightarrow_\rho H$, with S' the image (occurrence) of S in G as above, the number of connection relations linking the imbedded target to the rest of H is no more than the number of connection relations linking S' to G.*

3.3 Grammars

A collection of productions acting on a given dangling graph constitutes a *dangling graph grammar.* Such a system consists of a pair of objects: a collection of productions, Υ, and an initial graph, the *axiom*. All the graphs that can be derived from this axiom using only the given productions collectively form the *language* generated by the grammar:

Definition 3.10 *The* language *generated by a graph grammar $G = (A, \Upsilon)$ is the set of all dangling graphs which can be derived from A using productions in Υ:*

$$L(G) \;=\; \{B| \; A \xrightarrow{*}_\Upsilon B\}$$

3.4 Grammar Properties

Our ability to partition the graphs generated by our grammars will depend on the grammars having two properties, both based on a concept of *overlap* between dangling graphs. While both of our properties are restrictions on grammars, the combination has the benefit of also being sufficient to sensibly extend our grammars to *parallel* grammars—ones wherein more than one production can be applied concurrently. Thus, the type of grammar we develop will be such that any number of productions can be applied concurrently.

Definition 3.11 *Two dangling graphs D and S* **overlap** *if there exist complete subgraphs of each, D' and S' respectively, a non-empty dangling graph W such that $W \equiv D'$ and $W \equiv S'$, and such that every dangling edge of W is mapped by the $\equiv$ relation to either a dangling edge of D or a dangling edge of S (or both). The set of all such maximal (in number of nodes and connections) such W form the actual overlap of S and S'.*

In a parallel model of application, we may have more than one production applying at once. If two productions are applied at the same time, however, and their occurrences are not completely disjoint, the two form a critical pair—conflicting behaviour might be specified for nodes in the intersection of the two occurrences.

Fortunately, it easy to restrict our class of grammars to ones admitting concurrent application while still being deterministic. If all occurrences *must* be disjoint, then the rewrite of each node and 1/2-edge is determined by only one production, and there can be no conflict in specification. This is precisely the no overlap property between all production source graphs:

Definition 3.12 *If $G = (A, \Upsilon)$ is a dangling graph grammar, and for all $\rho_1, \rho_2 \in \Upsilon$, $\rho_1 = (S_1, T_1, \delta_1)$ and $\rho_2 = (S_2, T_2, \delta_2)$ it is the case that $Overlap(S_1, S_2) = \emptyset$ or $\rho_1 = \rho_2$, then G is* **SS-overlap free.**

Proposition 3.2 *If (A, Υ) is SS-overlap free, then the grammar is deterministic even if some productions are applied simultaneously.*

Proof: Let $G = (A, \Upsilon)$ be a non-deterministic grammar. Then for some dangling graph D there must exist some node n included in each of the simultaneous occurrences O and O' of two productions ρ and ρ'. Let s and s' be the images of n in S and S' (the source graphs of ρ and ρ') respectively; it must be that the complete subgraph consisting just of n and its 1/2-edges is isomorphic to s and to s'. Let W be the largest complete subgraph of D including n which has an isomorphic image in S and S'.

Let e be a dangling edge of W, and suppose e is not mapped by the isomorphism to any dangling edge of S or S'. Let d and d' be the 1/2-edges in S and S' to which e is isomorphic, and let r and r' be the nodes attached to the other 1/2-edges involved in the connection relation with d and d'. Both r and r' must be included in their occurrences, but the connection to them is not included in W; either W is not maximal, or the occurrence of one of S or S' does not include a match for r or r' (and so one of S or S' does not in fact occur), either of which is a contradiction. $\square$

Lack of overlap between source graphs, though, is not enough to ensure partitionability. To guarantee that the tree-based method we will develop below applies, it is necessary that the overlap between source and target graphs also be restricted.

Definition 3.13 *Let $G = (A, \Upsilon)$ be a dangling graph grammar, and let $\mathcal{T} = \{T \mid (S, T, \delta) \in \Upsilon\}$. G is* **ST-overlap free** *if for all $(S, T, \delta) \in \Upsilon$, we have:*

$$\forall \tau \in \mathcal{T},\ \forall O \in Overlap(S, \tau),\ (O \equiv \emptyset\ \vee\ O \equiv S)$$

Remarks: The ST-overlap free property merely specifies that given any combination of production source S and target τ, either S actually occurs in τ, or S and τ do not overlap.

This simple property will prove critical when we describe the partitioning method. Note that this definition implies that if every production in a graph grammar has a source consisting of just one node ("context-free" in the literature), then the grammar is trivially ST-overlap free (the overlap of a single-node graph and any other can only be an identical single-node graph, or empty).

3.5 Contexts

The development of many dynamic data structures depends on the nature of the graph locally surrounding the update site. The process of changing the data structure requires rewriting only a small area, but the decision to do so may depend on the surrounding neighbourhood; a binary tree in which right-child leaves are to be expanded into subtrees only after left-child leaves have already been rewritten into subtrees, for example, requires this sort of local information. This can be modelled with our grammars, but it would require rewriting the entire *context* for the rewrite—the update site, and its neighbourhood. Doing so, however, often introduces undesired overlap between productions that depend on the same sort of neighbourhood.

This problem can be alleviated by including *contexts* along with the source of each production. A context is just a dangling graph which includes the source within it; the entire context must occur in order for the production to be applied, but only the source is actually rewritten. In this way the application of a production can be restricted to a given graph configuration. We therefore define grammars with contexts as one of the possible variations we will be considering with respect to partitionability.

Definition 3.14 *A production* **with context** *is one $\rho = (S, T, \delta)$ with a context I as defined above, with the property that each occurrence of S must be included in an occurrence of I. If a grammar G includes a production with context then the grammar is with context.*

4 Partitioning Trees

Our method for generating partitionable structures relies on being able to efficiently partition trees. Here we prove that *weighted* trees, trees with a non-negative weight w_i assigned to each node, with a total weight of W and a bound b on the fanout of each node are $O(log(W))$-partitionable.

Lemma 4.1 *Given a natural number n, and a set N of any other m natural numbers which sum to n, it must be that if n_i is the i^{th} largest number in N then $n_i \leq n/i$.*

Proof: By contradiction; assume n_i is strictly larger than n/i for some n, N and i. Since n_i is the i^{th} largest, there are $i - 1 \geq 0$ other numbers in N, each of which is at least as large as n_i. These i numbers then necessarily sum to a value strictly greater than n. □

We will use lemma 4.1 to prove a cost bound on a certain kind of partitioning of trees. First, we define some essential terminology.

Definition 4.1 *Let $T = (N, E)$ be a tree with nodes N and edges $E \subset N \times N$. Then Subtree(n) for $n \in N$ is the set of all nodes in N which are in the subtree rooted at n, including n itself, and Fanout(n) is the number of children of a given node n.*

Definition 4.2 *A* postorder *tree traversal is a total ordering of the vertices of a tree such that if v_i represents the i^{th} vertex in the ordering, then v_j is a vertex in the subtree rooted at v_i only if $j < i$.*

A postorder search of a tree is most often discussed in the context of recursion, where it corresponds to a recursive search of a tree, examining each child node before examining the parent node. In such a non-backtracking procedure, each stage of the enumeration implies a separation of the vertices of the tree into two groups: those which have been enumerated, and those which have not, with movement always from the latter group to the former. If this grouping serves as a basis for partitioning, the communication cost can be bounded for bounded-degree trees. Let $\cong_s$ represent the equivalence class based on the enumerated/not-enumerated division, when s vertices have been enumerated.

Lemma 4.2 *Let T be a tree of n nodes with maximum fanout b, with a positive integer weight w_i assigned to each node v_i, such that $\sum_i w_i = W \geq 1$. Let W_i be the total weight of all nodes in the subtree rooted at v_i; we also require that W_i is at least 1 for all subtrees. If a postorder search is performed where the child nodes are examining in decreasing order of total subtree weight, ordering the vertices as $v_1, \ldots, v_n$, then for any partition of T into two parts, $\cong_i = \{\{v_1, \ldots, v_i\}, \{v_{i+1}, \ldots, v_n\}\}$, it must be that Cost($\cong_i$) $\leq (b-1)\log_2(W) + b$.*

Proof: By induction on n, the number of nodes in the tree. The base case, with just a single node is trivially true. Since there are no edges, cost is 0. Assume true for all $n' < n$, and let T be an n-node tree, each node having maximum fanout b, and with $t_1, \ldots, t_{b'}$ as the $b' \leq b$ child trees, ordered by decreasing total subtree weight.

If the root of T is enumerated in a postorder search, then the entire tree T has been enumerated, and partition cost is 0. Assume, then, that the root of T has not been enumerated.

The cost of the partitioning $\cong_i$ will be at most one for each of the children that have been fully enumerated (to account for the edge connecting the child to the root), plus the cost of the partial enumeration of any single child. When no children are partially enumerated, total cost cannot be more than $b' \leq b$. So, assume at least one child tree is only partially visited, and that it is the j^{th} largest in terms of total weight. Let w_r be the weight of the root node.

If $j = 1$, then no other subtrees have been enumerated, and so the total cost is just the cost of the partial enumeration of t_1, which is by inductive assumption bounded by $(b-1)\log_2(W - w_r) + b$.

Assume, then, that $j > 1$. By definition of the search strategy, the $j-1 > 0$ subtrees with larger weights have already been enumerated, and so the cost must include the $j-1$ links to the parent. By lemma 4.1, the j^{th} subtree can have weight no more than $(W - w_r)/j$. Hence, using the inductive hypothesis, partially enumerating t_j can cost no more than $(b-1)\log_2((W-w_r)/j) + b$, or equivalently $(b-1)\log_2(W-w_r) - (b-1)\log_2(j) + b$.

Adding in the cost of severing the $j-1$ links to the parent for the fully-enumerated subtrees, the entire cost can be no more than $(b-1)\log_2(W-w_r) - (b-1)\log_2(j) + j - 1 + b$. By assumption $1 < j \leq b$, and hence $\log_2(j) \geq 1$ and $j - 1 \leq b - 1$. Thus, the term $j - 1 - (b-1)\log_2(j) \leq 0$. Since w_r is non-negative, the total cost is then upper bounded by $(b-1)\log_2(W) + b$. □

This lemma establishes an upper bound on the cost of partitioning. However, bounds on load-balancing do not directly follow; for load-balancing we need to assume bounds on the sizes of the weights associated with each tree node.

Corollary 4.1 *If T is a tree with total weight W as described in lemma 4.2 with the extra condition that for all weights w_i, $w_i \leq m$, then for any $0 \leq \omega \leq W$, there exist two partitionings, $\cong_s$ and $\cong'_s$, of T into two parts T_1, T_2 and T'_1, T'_2 respectively such that T_1 has total weight $\omega - m'$ for some $0 \leq m' \leq m$, T'_1 has total weight $\omega + m''$ for some $0 \leq m'' \leq m$, and and the total cost of either partitioning is no more than $(b-1)\log_2(W) + b$.*

Proof: Let $v_1, \ldots, v_n$ be the n vertices of T ordered as per a post-order search examining child trees in order of decreasing total weight. Since no vertex has weight larger than m, there must exist some i such that $w_1 + \cdots + w_i = \omega - m'$ for some $0 \leq m' \leq m$. Similarly, there must exist some j such that $w_1 + \cdots + w_j = \omega + m''$ for some $0 \leq m'' \leq m$. By lemma 4.2, both partitionings have cost no more than $(b-1)\log_2(W) + b$. □

Lemma 4.2 and corollary 4.1 establish an upper bound on the cost of a partitioning and the maximum difference between partition sizes, respectively. Partitioning, however, is into p pieces where p can be anywhere between 1 and W, the total weight of the tree. We would like, then, bounds on the cost and size of partitions when dividing the tree into p pieces for any $1 \leq p \leq W$. Given a tree T as in lemma 4.2, and a post-order search of its weighted vertices, we can consider the problem of producing such a weight-balanced p-partitioning T to be equivalent to the problem of p-partitioning an ordered sequence of non-negative integers summing to W, each of which is no more than m.

Lemma 4.3 *Given an ordered list of n integers, $N = w_1, \ldots, w_n$ such that $0 \leq w_i \leq m$ and $W = \sum w_i$, N can be partitioned into $1 \leq p \leq W$ disjoint, contiguous and covering sets, such that each partition has sum $W/p \pm m$.*

Proof: The total weight, W, can be rewritten as $Wx/p \pm m$, for $x = p$. Under this syntax, N should be partitioned into contiguous and covering pieces totalling $W/p \pm m$.

We perform an induction on x. Assume N is contiguous and has sum $W' = Wx/p \pm m$, for some positive p and positive $x \leq p$, and that we wish to split N into x pieces, each of sum $W/p \pm m$.

The base case, $x = 1$, is trivially true; the lone partition is all of N, and has by assumption a total of $W/p \pm m$.

Assume true then for $x - 1$, and let $x > 1$. Let the actual weight of N be $W' = Wx/p + m'$, for some $0 \leq m' \leq m$ (the other case, $W' = Wx/p - m'$, is symmetric). By corollary 4.1, N can be split either at $\omega + m_1$ or $\omega - m_2$, for any given $0 \leq \omega \leq W'$ and some $0 \leq m_1, m_2, \leq m$, so remove from the front a contiguous partition N_1 of size $W/p + m_1$. The remaining partition, N_2 is also contiguous and has weight $W' - W/p - m_1 = W(x-1)/p \pm m$, so by inductive hypothesis N_2 can be partitioned into $x - 1$ pieces, each with weight $W/p \pm m$.

Since no partitions overlap, and the base case consumes the entire remaining list, the partitions must be covering. Each partition is also a contiguous portion of a contiguous list, and so the partitioning satisfies the given criteria. □

REMARKS: Although the above lemma proves that N can be partitioned into p pieces for any $1 \leq p \leq W$, if $p \geq W/m$ then some partitions may exist which contain no vertices at all. Still, these partitions fall within the $\pm m$ bounds on partition size.

Corollary 4.2 *If T is a b-ary tree, as per lemma 4.2 with an upper bound m on the weight associated with each vertex, then T can be partitioned into $1 \leq p \leq W$ pieces, each of which has total weight $W/p \pm m$, and total cost no more than $2(b-1)\log_2(W) + 2b$.*

Proof: By lemma 4.3, T can be partitioned into p pieces such that each partition has weight $W/p \pm m$. Each such partition is completely separated from the rest of the tree by no more than two cuts, each of which can be seen as a split of T into two pieces. Hence, by corollary 4.1, each partition can have cost no more than twice $(b-1)\log_2(W) + b$. □

Thus, it is possible to partition b-ary weighted trees with an $O(\log(W))$ bound on partition cost, and the load-balance of the partitions will be a function of the bound on the weight assigned to each tree node.

5 Graph Partitioning

The two overlap properties, SS-overlap free and ST-overlap free, are together sufficient to give the history of production applications a general "tree-like" shape, which can be exploited for partitioning the graphs generated. The nature of the graph embedding, combined with these properties, ensures that this tree-like aspect remains tree-like throughout the derivation of each graph in the grammar language. This property is robust in the face of parallel or sequential execution, and with or without contexts.

5.1 Tree Partition Schemes

The partition strategy we will evince for graphs will be based on a method for partitioning trees, and a mapping from the nodes of the graph to the nodes of the tree and from the connection relations of the graph to the edges of the tree. For any tree let the relation $a \leq b$ applied to nodes a and b indicate that a is contained in the subtree rooted at b. Then,

Definition 5.1 *A* **tree partition scheme** *for a dangling graph D with nodes V, 1/2-edges E and connection relations $C \subset E \times E$ is a tree T with nodes N and directed links L, together with a function $\nu : V \rightarrow N$ and a relation[2] $\tau \subseteq C \times L$ such that:*

1. *$\forall v \in V$, if $\nu(v) = n$ and $\ell : n' \rightarrow n$ and $v' \in V \setminus \bigcup_{\overline{n} \leq n} \nu^{-1}(\overline{n})$ and $e = (v, v')$ then $e \tau \ell$.*
2. *$\forall n \in N$, $|\bigcup_{\overline{n} \leq n} \nu^{-1}(\overline{n})| > 0$.*
3. *$\forall v, v' \in V$, if $\nu(v) = \nu(v')$ and $e = (v, v')$ then $\tau(e) = \emptyset$.*

Remarks: Several connection relations may be associated with a given link. If all these connection relations are cut, then the set of graph vertices corresponding to the nodes of the detached subtree become disconnected from the rest of the graph. Note that the relationships between tree links and graph connection relations do not reflect connectivity in any simple way; i.e.,one does not in general have a homomorphism.

An example of a graph embedded in a tree partition scheme is shown in figure 3. Dashed ovals indicate the graph nodes mapped to each tree node, and all the edges between two ovals are mapped to the corresponding tree link. Cutting all the edges mapped to a given link is guaranteed to disconnect all graph nodes mapped into the subtree from the rest of the graph.

The nature of the mapping and the tree will of course be critical to the success of the method. A tree consisting of just one node to which every graph node is mapped satisfies the above requirements, but clearly does not further the task of partitioning.

Definition 5.2 *A tree partitioning scheme (T, ν, τ) of a dangling graph D is said to be* **bounded** *if there exist three positive integers, β, μ, λ, where:*

[2] We will often use the functional notation, i.e.,$\tau(e) = \{\ell \in L |\ e \tau \ell\}$, with the converse $\tau^c(\ell) = \{e \in C |\ e \tau \ell\}$.

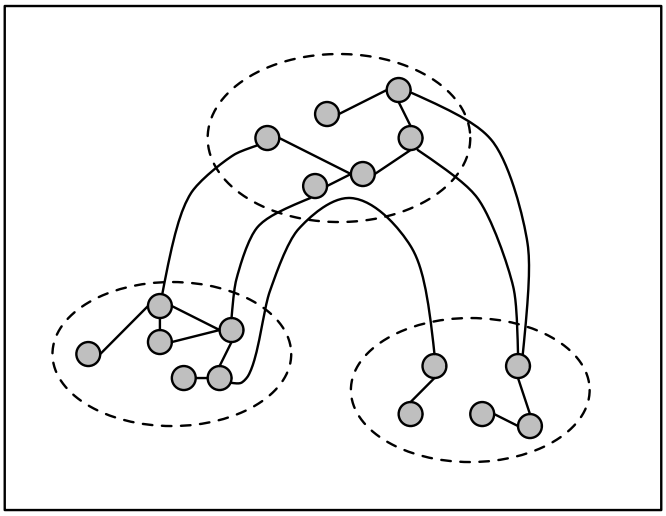

Figure 3: A graph imbedded into a tree partition scheme.

1. β *is a bound on the branching factor of* T.

2. μ *is a bound on the size of* $\nu^{-1}(n),\ \forall n \in N$.

3. λ *is a bound on the size of* $\tau^c(\ell),\ \forall \ell \in L$.

Bounded tree partitioning schemes permit the graph to be partitioned with cost and size bounds determined by the three numbers β, μ, and λ.

Lemma 5.1 *Let* $T(\nu, \tau, \beta, \mu, \lambda)$ *be a bounded tree partitioning scheme of* n *nodes for some dangling graph* D *of* $|V|$ *nodes, as detailed above. Then* D *can be partitioned into* p *pieces each of size* $|V|/p \pm \mu$ *with maximum cost* $2\lambda(\beta - 1)\log_2(|V|) + 2\lambda\beta$.

Proof: This follows directly from corollary 4.2. □

In order to describe how these tree partitioning methods and structures apply to dangling graph grammars, it is first necessary to define bounds which depend on the grammar specification itself.

Definition 5.3 *The* **bounds** *of a dangling graph grammar* $G = (A, \Upsilon)$ *are three positive integers,* m, g, *and* k *such that* $|A| \leq m$, $\forall n \in V_A$, *Degree*$(n) \leq k$, *and* $\forall\ (S, T, \delta) \in \Upsilon$ *we have* $|T| \leq m$, $|S| \leq g$, *and Degree*$(v) \leq k$ *for all vertices* v *in* T.

We can associate a bounded tree partition scheme with each graph generated by the grammar. Inductively, each time the grammar is iterated generating a new graph from an old, a new bounded tree partition scheme is also created from the old scheme. The two overlap properties ensure that the tree partition scheme remains a tree after every set of concurrent rewrites.

Lemma 5.2 *Let $G = (A, \Upsilon)$ be an SS-overlap free and ST-overlap free dangling graph grammar with bounds (m, g, k). Then for any non-empty dangling graph D where $A \xrightarrow{s}_{\Upsilon} D$ for some s, there exists a bounded tree partition scheme $T(\nu, \tau, \beta = m, \mu = m, \lambda = gk)$ such that $\forall n \in N$, $Fanout(n) + w_n \leq m$, where w_n is the weight of node n. Moreover, let O be an occurrence of a production in Υ in D; then if v, v' are graph vertices in O, $\nu(v) = \nu(v')$.*

Proof: By induction on the size of s. Let $T = (N, L)$, where N is the set of tree nodes and L is the set of tree edges (or links). In all cases we will let $w_n = |\nu^{-1}(n)|$, and total weight W will be the number of graph vertices.

The base case is trivial; when $s = 0$, $D = A$, and T can be a single node tree, $T = (\{n_1\}, \{\})$, with ν defined as the constant function with $\forall v \in V_G$, $\nu(v) = n_1$ and τ undefined everywhere. Two of the three required integers are trivial, β and λ certainly exist at the indicated levels, since there are no tree edges, and since $|A| \leq m$, $|\nu^{-1}(n_1)| \leq m$, giving the third required bound. Since there is only one node in T and it corresponds to the axiom, necessarily each graph vertex is mapped to n_1, and so the vertices v, and v' of any occurrence must be both mapped to n_1.

Assume true for any D' such that $A \xrightarrow{s-1}_{\Upsilon} D'$, and let D be such that $A \xrightarrow{s-1}_{\Upsilon} D' \xrightarrow{1}_{\Upsilon} D$. By inductive hypothesis, there exists a bounded tree partition scheme $T'(\nu', \tau', \beta', \mu', \lambda')$ for D' with the above properties; we will show how to extend T' to a bounded tree partition scheme T for D.

The graph D is the rewrite of D' by the productions in Υ. Hence, there is a set $\mathcal{O}$ of all occurrences that transformed D' to D. As well, and because the grammar is deterministic, a function exists $\kappa : \mathcal{O} \rightarrow \{Z | Z \subseteq_c D\}$, which returns the imbedded target of a given occurrence.

By inductive assumption, each occurrence $O \in \mathcal{O}$ in D' must rewrite only vertices mapped to the same tree node, and so a function $\sigma : \mathcal{O} \rightarrow N'$ exists associating occurrences with the tree node containing the vertices forming the occurrence.

We define ν and τ to be the same as ν' and τ' for all nodes and connections not changed by the rewrite. We now construct T from T' with the following changes:

Add new nodes For each $O \in \mathcal{O}$ create a new node n_O in T, and for each such O extend ν to map the image of every graph vertex v in $\kappa(O)$ to n_O. Note that each $\kappa(O)$ thereby has a corresponding target node, n_O in T, and so a function exists $\zeta : \mathcal{O} \rightarrow N$. By assumption, $|\kappa(O)| \leq m$, and since each n_O has fanout 0, it is still true that $\text{Fanout}(n) + w_n \leq m$.

Connect new nodes For each $\zeta(O)$ created in the above step which is not already connected to the rest of T, add an edge in T from $\sigma(O)$ to $\zeta(O)$, and delete all nodes in $\mathcal{O}$ from the function ν. Since each production must rewrite at least one graph node, and the same graph node can never be rewritten by more than one production, if T' had the property that each tree node n is such that $\text{Fanout}(n) + w_n \leq m$, then this will surely be the case in T' after adding these edges and deleting these nodes from the node-mapping function.

Include new edges in τ Let $C_O = \{(e, e') \in C_D | \ (e, e') \in \mathrm{CSet}(\kappa(O), D - \kappa(O))\}$. Increase the relation τ to map each connection in C_O to the tree edge $(\sigma(O), \zeta(O))$. Each $\kappa(O)$ is linked to the rest of D only by modifications to the original connection set between O and D' (see proposition 3.1). Since each occurrence consists of at most g graph nodes, of degree at most k (by assumption), there can be at most gk distinct[3] connection relations between $\kappa(O)$ and $D - \kappa(O)$. Hence, τ maps no more than gk graph edges to the tree edge $(\sigma(O), \zeta(O))$. Note that all other differences between τ' and τ result from the *deletion* of connections (due to rewrites), and so the number of connections mapped to an existing edge in the tree can only decrease.

Fixup τ for existing edges Consider the set of all connection relations (e, e') in C such that there exists (f, f') in C' with either $(e = f, e' = \delta_2(f'))$, $(e = \delta_1(f), e' = f')$, or $(e = \delta_1(f), e' = \delta_2(f'))$, for some δ_1 and/or δ_2 (of two productions ρ_1 and ρ_2). These are all the connection relations which are have been altered by a substitution using some δ operator(s). Increase τ to map (e, e') onto $\tau'(f, f')$. This process does not alter the mappings of any newly created tree edge, and only replaces a former mapping ((f, f') will not exist in D) with a corresponding new one, so any bounds on the size or claims about connectivity for T' will continue to hold in T.

It remains to verify that the generated tree T is indeed a bounded tree partition scheme with the desired integers and properties as described in the statement of the lemma.

As detailed in the above steps, the constructions of ν from ν' and τ from τ' are such that $|\nu^{-1}(v)| \leq m$, and $|\tau^c(\ell)| \leq gk$. Also by construction, severing the edges mapped to any $(\sigma(O), \zeta(0))$ disconnects $\kappa(0)$ from the rest of the graph, so τ certainly possesses the desired disconnection property for all $\kappa(0)$. To see that τ retains this property for the rest of the tree, we can simply note that in the last step if (e, e') is a connection relation in D' which is altered by the rewrite, then the rewritten connection will replace the previous connection, and all connections untouched by the rewrite are retained.

Let $e = (v, v')$ be a connection relation in D. If both v and v' existed in D', then if $\nu(v) = \nu(v')$ by inductive assumption $\tau(e) = \emptyset$. If v existed in D' and v' did not, then by construction it cannot be that $\nu(v) = \nu(v')$, and if neither v nor v' existed in D then also by construction if $\nu(v) = \nu(v')$ then both v and v' are mapped by ν to the same newly-introduced node, and so $\tau(e)$ will not be defined on e.

The third integer bound for a bounded tree partition scheme is trivial to verify. Because of the invariant $\mathrm{Fanout}(n) + w_n \leq m$, for all tree nodes n, a bound m exists on the branching factor of T.

It is necessary to ensure that any future occurrences of these productions will have all their graph vertices mapped by ν to the same tree node. Consider an occurrence

[3] In fact, there are at most $2gk$ such connection relations, but since connection relations are symmetric we need only be concerned with distinct pairs.

O of some production $\rho = (S_\rho, T_\rho, \delta_\rho) \in \Upsilon$ in D. Trivially, if all vertices in O are mapped by ν to the same tree node $n \in N$, then the property is satisfied. If O includes vertices only mapped by ν to tree nodes in N', then the inductive hypothesis ensures the desired property—vertices are never added to existing tree nodes, so if an image of S_ρ exists in D using just vertices from D', then S_ρ also occurred in D'.

Assume, then, that O includes some vertices mapped to a node in N which is not in N'; let v be such a vertex, $\nu(v) = n \notin N'$, and let v' be another vertex in O such that $\nu(v') \neq \nu(v)$. Because n is a tree node we just inserted, the vertices in $\nu^{-1}(n)$ are the embedded copies of some production target graph τ. Let O_v be a maximal (strictly) complete subgraph of O including v with every vertex in O_v mapped by ν to $\nu(v)$. It must be that $O_v \in \text{Overlap}(S_\rho, \tau)$; every 1/2-edge of O_v is either a 1/2-edge of $O \equiv S_\rho$, or it matches a 1/2-edge of τ—a 1/2-edge e of O_v which is not in $\Xi(O)$ is connected to some other 1/2-edge e' and vertex v' in O. If there is a corresponding match for e' and v' in τ, then O_v is not maximal; if there is not and the image of e is not dangling in τ, then it cannot be that O is an occurrence. Thus, there is a member of $\text{Overlap}(S_\rho, \tau)$ which is neither $\emptyset$, nor the same as S_ρ (O_v includes v but not v'), and the grammar cannot be ST-overlap free.

The only remaining property to check is the assertion that no subtree of T exists with total weight 0. The above construction generates tree nodes for each imbedded target, even if the target is the empty graph, and so after the indicated steps some branches of T might exist which have 0 weight. However, such "dead branches" can be removed without altering any of the desired properties. Numerical bounds on tree branching, the maximum number of connection relations mapped to tree edge, or the maximum number of vertices mapped to nodes are trivially preserved. Since there are no vertices mapped to any node in such a dead branch, all conditions specified for tree partitioning schemes, and the extra conditions in the lemma statement too, continue to apply after removing all dead branches. □

This lemma leads directly to our main result:

Theorem 5.1 *Let $G = (A, \Upsilon)$ be an SS-overlap free and ST-overlap free dangling graph grammar with constant bounds (m, g, k). For any dangling graph D such that $A \xrightarrow{*}_\Upsilon D$, it must that D is $(2gk(m-1)\log_2(|V|) + 2gkm)$-partitionable.*

Proof: This follows trivially from lemma 5.2 and lemma 5.1. By the former, for each dangling graph generated by G there is a corresponding bounded tree partition scheme $T(\nu, \tau, m, m, gk)$, and by the latter such a tree can be partitioned into pieces of size $|V|/p \pm m$ with maximum cost $2gk(m-1)\log_2(|V|) + 2gkm$, for any $1 \leq p \leq |V|$. □

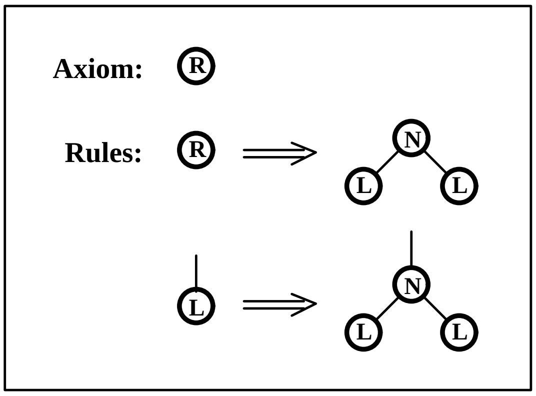

Figure 4: A grammar generating trees.

6 Expressibility

There is no guarantee that the class of graphs constructed using the above methods will be at all interesting. We can, however, demonstrate the expressibility of our scheme by showing how to generate a variety of computer science data structures.

6.1 k-ary Trees

The class of k-ary trees is trivial to generate. The axiom consists of just a single node, labelled `root`. Two rules then suffice to expand either the root or a leaf into an internal node and k child leaves. In the case of a leaf being rewritten, the 1/2-edge connecting the rewritten leaf to its parent is associated with the 1/2-edge extending from the internal node (see figure[4] 4).

6.2 Threaded k-ary Trees

By adding two 1/2-edges to each node, for left and right threaded neighbours, threading can be maintained as the leaves are expanded (see figure 5). If just the leaves are to be threaded, the process is similar; expanded leaves generate a leaf-threaded subtree, and the original threaded neighbours are transferred to the new leaf children (figure 6). Our examples illustrate binary trees and inorder threading, but clearly it is possible to accomodate any recursive threading policy in the same way. In fact, the same underlying method illustrated here we have used to express the quasi-balanced quadtree data structure used in a very non-trivial finite element method algorithm [51].

[4] Edge labels and the δ function are not illustrated; they should be obvious from the geometric positioning of the 1/2-edges.

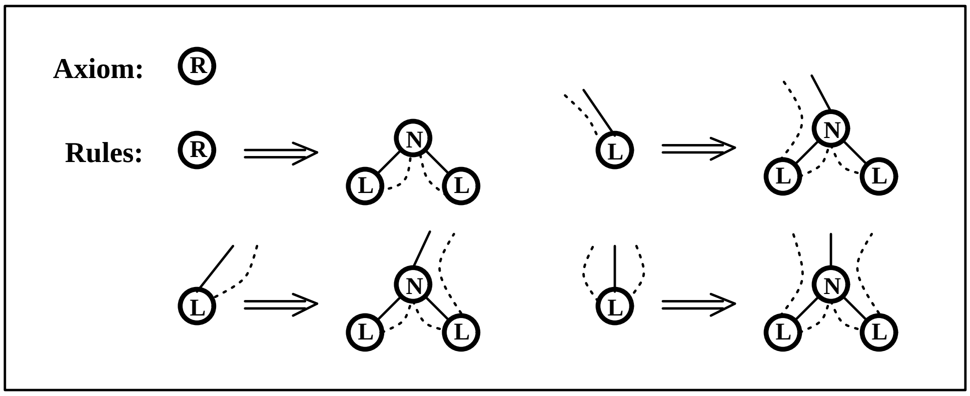

Figure 5: A grammar generating inorder threaded (binary) trees.

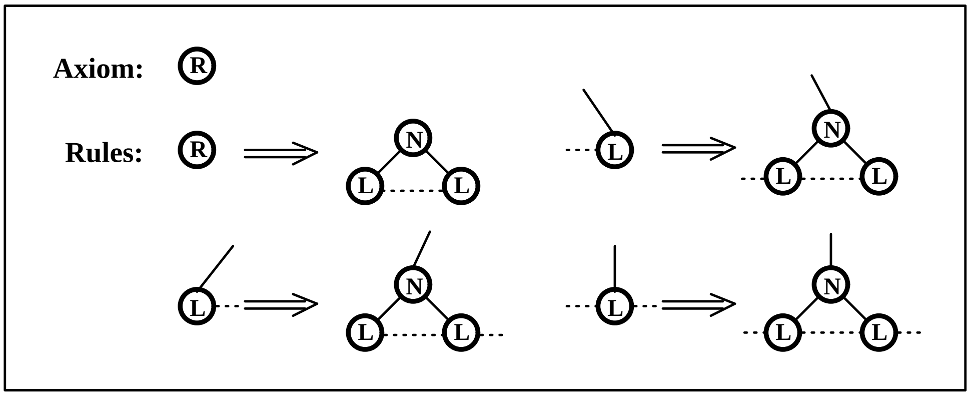

Figure 6: A grammar generating leaf-threaded trees.

6.3 Linked Lists

Normal linked lists can be easily generated by marking the tail and/or head of the list distinctly, and then generating new entries by expanding the tail or head into an internal node and a new tail or head (figure 7). If the list is intended to be circular, then an extra connection between the head and tail is maintained through the rewrites (figure 8). Different orders of application for the rules then correspond to the different variations on lists—stacks, queues, double-ended queues, *etc.*

6.4 Compiler Control-Flow Graphs

Structured procedural languages can be modelled by graphs, with linked lists of nodes representing sequences of statements, and cycles representing loops and conditionals. The usual directedness of these graphs is simply reflected in the choice of edge labels.

The axiom thus consists of a single statement node bracketted by a begin and end marker. Rules exist to expand a statement node into a pair (or more) of statement nodes, into a loop

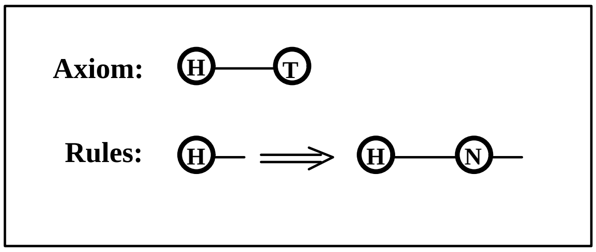

Figure 7: A grammar generating linked lists.

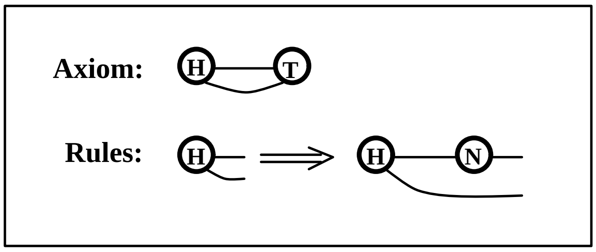

Figure 8: A grammar generating circular linked lists.

statement consisting of a cycle including a statement node, or into a conditional, consisting of a cycle with true and false branches, with the conditional exit continuing control flow out of the conditional (figure 9).

6.5 Contexts

The above examples are all of grammars with the productions having just single nodes in their source graphs. Data structures which are built based on the local nature of the surrounding graph require contexts or larger source graphs to distinguish which vertices are to be expanded. A tree, for example, where right-child leaves are expanded only if the corresponding leaf-child leaves have already been expanded would need this sort of local information (see figure 10).

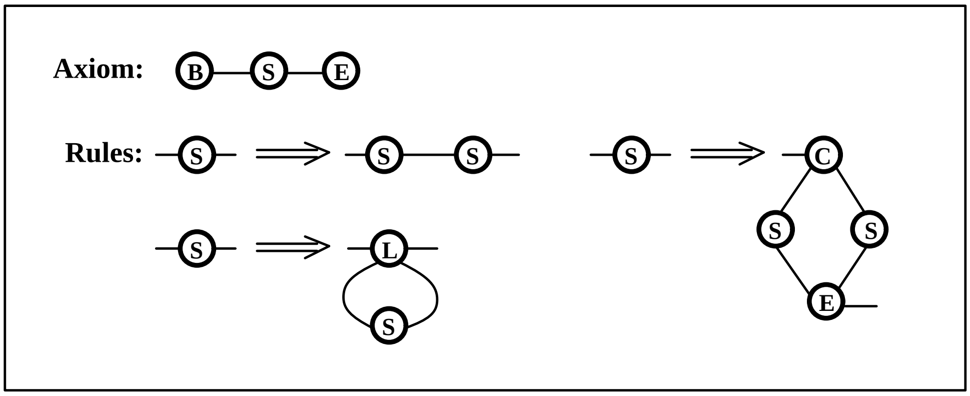

Figure 9: A grammar generating compiler control flow graphs.

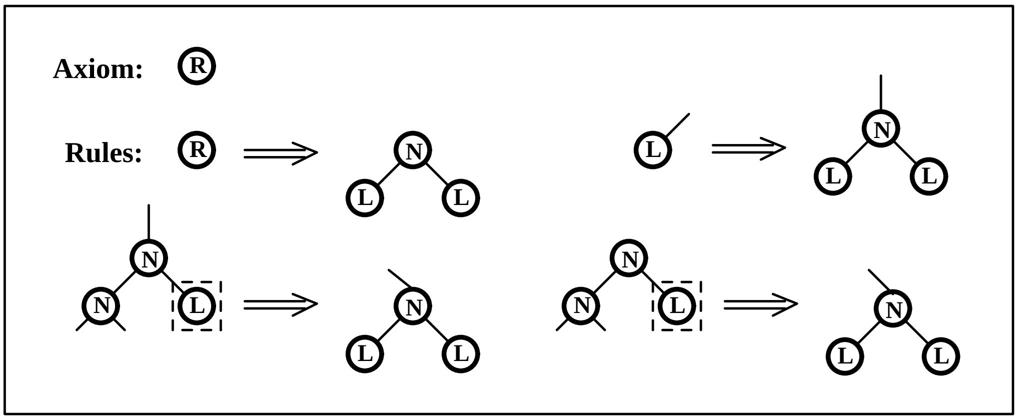

Figure 10: A grammar requiring contexts; right leaves are expanding only if the left sibling is not a leaf.

7 Tree Width

Tree-width is a concept in graph theory meant to model how "close" a given graph is to a tree. More importantly, a bounded tree-width specifies a large class of graphs for which polynomial-time (and often linear-time) algorithms exist for a variety of problems in NP [3, 4, 8, 28, 49]. If we can find a tree partition scheme for a given graph, however, it is possible to adapt the tree partition scheme into a tree decomposition, and the tree-width is then necessarily bounded. First, however, we must define tree-width:

Definition 7.1 *Let $T = (N, L)$ be a tree with nodes N and edges L. Then $\pi : N \times N \rightarrow \mathcal{P}(N)$ is a function returning the set of vertices forming the loop-free path connecting the two input vertices.*

Definition 7.2 *Let $G = (V, E)$ be a graph. A* **tree-decomposition** *of G is a tree $T = (N, L)$ and a function $\sigma : N \rightarrow \mathcal{P}(V)$ such that:*

1. $\forall (v, v') \in E,\ \exists n \in N.\ v, v' \in \sigma(n)$.
2. $\forall n, n' \in N,\ v \in \sigma(n) \bigcap \sigma(n') \ \Rightarrow\ \forall \widehat{n} \in \pi(n, n')\ v \in \sigma(\widehat{n})$
3. $V = \bigcup_{n \in N} \sigma(n)$.

Let m_T be the maximum cardinality of any $\sigma(n)$ in T; i.e., $m_T = \max|\sigma(n)|$ over all n in N. Then the **tree-width** *of D is defined as one less than the minimum m_T over all tree-decompositions T.*

Theorem 7.1 *Let G be an SS-overlap free and ST-overlap free dangling graph grammar with constant bounds (m, g, k). Let $D = (V, E, \nu, \phi, \psi, \Sigma_V, \Sigma_E, C)$ be a dangling graph such that $A \xrightarrow{*}_\Upsilon D$, necessarily D has tree-width smaller than $2mgk + m$.*

Proof: Let $T'(\nu', \tau', \beta', \mu, \lambda') = (N', L')$ be the corresponding bounded tree partition scheme, as generated in lemma 5.2. Let $T = (N, L)$ be a tree identical to T' by a homomorphism $h : T \rightarrow T'$, and let $\ell(n)$ be the set of outgoing links attached to a node n in T'. Define σ as:

$$\begin{aligned}\sigma(n) \quad = \quad & \{v \in V \mid \nu'(v) = h(n) \ \vee \\ & (\exists (e, e') \in C.\ (\nu(e) = v \ \wedge \ \exists l \in \ell(h(n)).\ (e, e')\tau' l))\}\end{aligned}$$

The function σ maps each tree node n in T to the set of vertices which are mapped by ν' to $h(n)$, or which are included in a connection relation which is mapped by τ' to an outgoing edge attached to $h(n)$. By construction of T', every vertex in D is already uniquely mapped by ν' to some node in T', so certainly σ covers the vertices of D. Every connection relation c in D is either between two vertices both mapped to the same tree node in T', in which case a node in T must exist containing both endpoints, or c links two nodes which are not mapped by ν' to the same tree node. In the latter case, let v, v' be the two endpoints. At least one of the subtrees rooted at $\nu'(v)$ or $\nu'(v')$ must not contain the other, and so c is a connection relation which must be broken in order to partition a subtree from the rest of the tree. By definition, then, τ' must map c to some tree link, and thus σ maps both endpoints to at least one node in T.

The remaining property to show is that whenever a graph vertex v is contained in $\sigma(n) \cap \sigma(n')$ for two tree nodes n, n', then for any $\widehat{n} \in \pi(n, n')$, $v \in \sigma(\widehat{n})$. Let v, x, y be a vertex and two nodes in such a situation, and let z be a node in $\pi(x, y)$. Since vertices are uniquely mapped by ν' to nodes in T', a subset of V can be identified with each subtree Subtree(t) in T, corresponding to the set of nodes contained in Subtree(t). Let $\epsilon : N' \rightarrow \mathcal{P}(V)$ be a function returning the subset V corresponding to a given node (subtree) in T'. Note that if and only if $n \in$ Subtree(n') for two tree nodes n, n', then $\epsilon(n) \subseteq \epsilon(n')$. We will also need the following lemma:

Lemma 7.1 *Let v be a vertex in D, and let $\mathcal{N}(v)$ be the set of vertices which are adjacent to v by no more than one connection relation (note that $v \in \mathcal{N}(v)$). Let n be a node in T such that $v \in \sigma(n)$, then $\exists v' \in \mathcal{N}(v).\ v' \in \epsilon(h(n))$.*

Proof: The set $\sigma(n)$ includes v only if v is mapped by ν' to $h(n)$, or if a connection involving v is required to be broken to partition off $\epsilon(h(n))$. The former is trivial; by definition, $v \in \mathcal{N}(v)$. The latter is not much harder; if a connection to v is to be broken to separate $\epsilon(h(n))$, then clearly $\epsilon(h(n))$ must include either v or an immediate neighbour of v. □

The common ancestor of x and y can be assumed to be neither x nor y, or it can be x itself (or y, but without loss of generality we can assume x). Assume the former, and let z be the common ancestor of x and y along $\pi(x, y)$ such that $z \neq x \wedge z \neq y$.

We first show that if v is a vertex mapped by σ to both x and y, then v is necessarily mapped to z.

Let z' be the child of z leading to x, and let z'' be the child of z leading to y. By lemma 7.1, $\exists v', v'' \in \mathcal{N}(v).\ v' \in \epsilon(h(x)) \ \wedge\ v'' \in \epsilon(h(y))$. Thus, $v' \in \epsilon(h(z'))$ and $v'' \in \epsilon(h(z''))$. The graph vertex v cannot be contained in both $\epsilon(h(z'))$ and $\epsilon(h(z''))$, so in order to partition off at least one of these subgraphs a connection to v will need to be broken. Thus, a connection relation involving v must be related by τ' to some link l from $h(z)$ to some child of $h(z)$, and thereby $v \in \sigma(z)$.

The common ancestor of x and y must therefore include all common vertices of $\sigma(x)$ and $\sigma(y)$ (of course if the common ancestor is not distinct from x and y, this is trivially true). Assume, then, that z is a node along $\pi(x, y)$ distinct from x and y where $x \in \mathrm{Subtree}(y)$.

Assume $v \notin \sigma(z)$. Then for each child z' of z, $\epsilon(h(z'))$ can be separated from the rest of D without cutting any connections involving v. By lemma 7.1, $\epsilon(h(x))$ includes either v or some immediate neighbour of v. Thus, if z' is the child of z leading to x along $\pi(x, y)$, then $\epsilon(h(z'))$ must include v and all immediate neighbours of v. However, $v \in \sigma(y)$, so either v is mapped by ν' to $h(y)$, in which case $v \notin \epsilon(h(z'))$ (a contradiction), or a connection involving v must be severed to separate $\epsilon(h(y))$. But v and all its neighbours are contained in $\epsilon(h(z'))$, and $\epsilon(h(z')) \subseteq \epsilon(h(y))$, so a contradiction again results.

This establishes that the tree T represents a valid representation from which one can derive (an upper bound on) tree-width. Since the number of outgoing links from any node in T' is bounded by m, the number of connection relations mapped to a given tree link is bounded by gk, and m is also a bound on the number of vertices mapped by ν' to any node, there will never be more than $2mgk + m$ vertices of D mapped by σ to any single tree node in T. $\square$

8 Related Work

Lipton and Tarjan [36] show how to partition planar graphs using $O(\sqrt{n})$ separator vertices; an extension of this classical result to graphs of fixed genus is available [50] and other variations have been explored [6, 9] from the viewpoint of graph embedding. These results give partitions of where the number of edges cut can reach $\sqrt{n}$ whereas we are interested in partitions such that the number of edges cut is $O(\log n)$.

Efforts have also been made to characterize data structures linguistically, using several approaches. Klarlund and Schwartzbach [27] have developed an interesting extension to data types by appending *routing expressions* to recursive type definitions. This allows the expression of recursive data structures which do not strictly form trees, and without the generality of arbitrary and explicit pointers. Their formalism is directed toward facilitating automated reasoning about pointer structures, though and not partitionability.

The *Abstract Description of Data Structures (ADDS)* formalism of Hummel, Hendren and Nicolau [18] is another interesting approach in the same vein. Recursive data structure definitions are augmented by a set of keywords defining the general shape (*via* interacting *dimensions*), and the intended traversals as well. The emphasis here is on increasing the compiler's ability to perform automatic error detection, optimizations, and fine-grain parellelization.

Graph grammars themselves have been extensively explored, though primarily with the aim of developing an expressiveness heirarchy [12, 20, 23, 41] analogous to work on string-based language theory. Nevertheless, graph grammars have been used for a variety of purposes. Okada and Hayashi develop a confluent graph grammar which they use for analysing network reliability [42]. The language *PROGRESS* is a graph rewriting formalism also intended to model data structures [48], though the embedding relation they use links target graphs using general path specifications, and so is unsuitable for partitioning.

Our grammars do not dramatically differ from the established formalisms for graph grammars. If we restrict the size of source graphs within production to just a single node, our grammars closely resemble the family of "eNCE" graph grammars [14, 15, 22]. The particular embedding operation we use for the target graphs of productions is quite similar to that used by Slisenko, in a work describing a polynomial-time solution to the Hamiltonian Circuit problem for certain graphs [49]. The concept of non-rewritten local contexts for productions is well-established in the theory of L-Systems, a parallel form of string-rewriting grammar [34, 44].

9 Conclusions

Our grammars cannot generate all graphs. A formalism which does generate all reasonably-partitionably graphs, and no more, is unlikely to exist given the plethora of NP problems in the area of graph partitioning. Nevertheless, our formalism is expressive enough to include a large variety of graphs and algorithms commonly used in computer science applications.

Graphs with bounded tree-width have been recognized as constituting a class of graphs about which many difficult problems, some in NP, can be solved efficiently [3, 4, 8, 30, 31, 32, 47, 49]. We have taken the opposite approach, starting with a difficult problem and showing that a certain efficient solution implies bounded tree-width. Nevertheless, it is interesting to find our solution converging to the same class of graphs. Bounded tree-width is clearly a fundamental property in graphs, and the connections with complexity theory add credence to our initial assumptions.

A major theoretical question is whether the bounded tree-width of the graphs generated by our grammars is a requirement for being reasonably-partitionable. It is straightforward to find examples of the converse—a tree consisting of a root with $n-1$ children has tree-width 1, the same as any other tree, but has a lower bound on partitionability of $n-1$. But if a graph is "as partitionable as a tree," is it necessarily tree-like? Of course, an affirmative answer would still not make partitionability easy to recognize.

The grammars we have defined form an interesting basis for synchronous parallel algorithms on irregular graphs. Any graph, and any algorithm which can be expressed in the grammar formalism, is necessarily reasonably-partitionable, and hence amenable to parallelization through automatic partitioning. The implicit load-balancing and guaranteed $O(\log(n))$ communication cost mean that as long as computation is reasonably uniform throughout the data structure, the computation will be efficiently parallel. Moreover, because the partitioning is determined incrementally by the derivation, the efficiency often can be maintained as the graph is grown and modified.

Acknowledgements

This research was funded by grants from NSERC of Canada and FCAR of Quebec.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] Noga Alon, Paul Seymour, and Robin Thomas. Planar separators. *SIAM Journal on Discrete Mathematics*, 7(2):184–193, May 1994.

[3] Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT*, 25(1):2–23, 1985.

[4] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Problems easy for tree-decomposable graphs. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, pages 38–51, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[5] Franz J. Brandenburg. The computational complexity of certain graph grammars. In A. B. Cremers and H. P. Kriegel, editors, *Theoretical Computer Science: 6th GI-Conference*, number 145 in Lecture Notes in Computer Science, pages 91–99, Dortmund, West Germany, January 1983. Springer-Verlag.

[6] Thang Nguyen Bui and Andrew Peck. Partitioning planar graphs. *SIAM Journal on Computing*, 21(2):203–215, April 1992.

[7] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. *SIGPLAN Notices,* 21(7), July 1986.

[8] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 195–241. Elsevier, Amsterdam, 1990.

[9] Krzystof Diks, Hristo N. Djidjev, Ondrej Sýkora, and Imrich Vrťo. Edge separators of planar and outerplanar graphs with applications. *Journal of Algorithms*, 14:258–279, 1993.

[10] W. E. Donat and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, September 1973.

[11] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Proceedings of the 4th International Workshop on Graph Grammars and their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[12] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rozenberg, editors, *Proceedings of the 3rdInternational Workshop on Graph-Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 3–14. Springer-Verlag, December 1987.

[13] Hartmut Ehrig, Paul Boehm, Udo Hummert, and Michael Löwe. Distributed parallelism of graph transformations. In H. Gottler and H. J. Schneider, editors, *Proceedings of the 13th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '87)*, number 314 in Lecture Notes in Computer Science, pages 1–19. Springer-Verlag, July 1988.

[14] Joost Engelfriet. Context-free NCE graph grammars. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT '89)*, number 380 in Lecture Notes in Computer Science, pages 148–161, Szeged, Hungary, August 1989. Springer-Verlag.

[15] Joost Engelfriet. A characterization of context-free NCE graph languages by monadic second-order logic on trees. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 311–327, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[16] Thomas A. Feo and Mallek Khellaf. A class of bounded approximation algorithms for graph partitioning. *Networks*, 20:181–195, 1990.

[17] Greg N. Frederickson. Optimal algorithms for tree partitioning. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 168–177, San Francisco, California, January 1991.

[18] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[19] Manfred Jackel. ADA concurrency specified by graph grammars. In Gottfried Tinhofer and Gunther Schmidt, editors, *Proceedings of the 12th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '86)*, number 246 in Lecture Notes in Computer Science, pages 41–57, Bernried, West Germany, June 17–19, 1986. Springer-Verlag.

[20] D. Janssens and G. Rozenberg. Restrictions, extensions and variations of NLC grammars. *Information Sciences*, 20:217–244, 1980.

[21] D. Janssens and G. Rozenberg. Hypergraph systems generating graph languages. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, pages 172–185. Springer-Verlag, October 1983.

[22] D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars. *Computer Graphics and Image Processing*, 18:279–304, 1982.

[23] D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars, II. *Computer Vision, Graphics, and Image Processing*, 23:295–312, 1983.

[24] Richard Kennaway. On "On graph rewritings". *Theoretical Computer Science*, 52:37–58, 1987.

[25] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[26] Brian W. Kernighan. Optimal sequential partitions of graphs. *Journal of the ACM*, 18(1):34–40, January 1971.

[27] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, January 10–13, 1993.

[28] Tom Kloks. *Treewidth: Computations and Approximations.* Number 842 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[29] Sukhamay Kundu and Jayadev Misra. A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1):151–154, March 1977.

[30] Clemens Lautemann. Decomposition trees: Structured graph representation and efficient algorithms. In M. Dauchet and N. Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, number 299 in Lecture Notes in Computer Science, pages 28–39. Springer-Verlag, March 1988.

[31] Clemens Lautemann. Efficient algorithms on context-free graph languages. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, pages 362–378, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[32] Clemens Lautemann. Tree automata, tree decomposition and hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 520–537, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[33] Timo Lepistö and Arto Salomaa, editors. *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[34] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[35] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, February 1992.

[36] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[37] Igor Litovsky, Yves Métivier, and Wieslaw Zielonka. The power and the limitations of local computations on graphs. In E. W. Mayr, editor, *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '92)*, number 657 in Lecture Notes in Computer Science, pages 333–345, Wiesbaden-Naurod, Germany, June 18–20, 1992. Springer-Verlag.

[38] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, May 1974.

[39] Jayadev Misra and R. Endre Tarjan. Optimal chain partitioning of trees. *Information Processing Letters*, 4(1):24–26, September 1975.

[40] Ugo G. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control*, 16:243–267, 1970.

[41] Manfred Nagl. On the relation between graph grammars and graph l-systems. In Marek Karpinski, editor, *Fundamentals of Computation Theory: Proceedings of the 1977 International FCT-Conference*, number 56 in Lecture Notes in Computer Science, pages 142–151, Poznan-Kornik, Poland, September 1977. Springer-Verlag.

[42] Yasuyoshi Okada and Masahiro Hayashi. Graph rewriting systems and their application to network reliability analysis. In G. Schmidt and R. Berghammer, editors, *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science*

(WG '91), number 570 in Lecture Notes in Computer Science, pages 36–47, Fischbachau, Germany, June 17–19, 1991. Springer-Verlag.

[43] Yehoshua Perl and Marc Snir. Circuit partitioning with size and connection constraints. *Networks*, 13:365–375, 1983.

[44] Przemyslaw Prusinkiewicz and James Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, pages 193–211. Springer-Verlag, Berlin, 1992.

[45] Jean Claude Raoult. On graph rewritings. *Theoretical Computer Science*, 32:1–24, 1984.

[46] Rüdiger Reischuk. Graph theoretical methods for the design of parallel algorithms. In L. Budach, editor, *Proceedings of the 8th International Conference on Fundamentals of Computation Theory (FCT '91)*, number 529 in Lecture Notes in Computer Science, pages 61–67, Gosen, Germany, September 1991. Springer-Verlag.

[47] N. Robertson and P. D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Journal of Algorithms*, 7:309–322, 1986.

[48] Andy Schürr. Introduction to PROGRESS, an attribute graph grammar based specification language. In M. Nagl, editor, *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '89)*, number 411 in Lecture Notes in Computer Science, pages 151–165, Castle Rolduc, The Netherlands, June 1989. Springer-Verlag.

[49] A. O. Slisenko. Context-free grammars as a tool for describing polynomial-time subclasses of hard problems. *Information Processing Letters*, 14(2):52–56, April 1982.

[50] Ondrej Sýkora and Imrich Vrťo. Edge separators for graphs of bounded genus with applications. *Theoretical Computer Science*, 112:419–429, 1993.

[51] C. Verbrugge, P. Panangaden, and R. Pucella. Locally-adaptive grid generation using quadtrees. Technical Report TR-SOCS-95.3, School of Computer Science, McGill University, Montréal, Québec, June 1995.

[52] C. Walshaw, M. Cross, S. Johnson, and M. Everett. A parallelisable alorithm for partitioning unstructured meshes. In *Proceedings of Irregular '94: Parallel Algorithms for Irregularly Structured Problems*, 1994.

[53] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, January 1988.