Theoretical Computer Science 238 (2000) 31–80

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fundamental Study

Generating irregular partitionable data structures

Prakash Panangaden[a], Clark Verbrugge[b,*,1]

[a] *McGill University, 3480 University St., Montréal, Québec, Canada H3A 2A7*
[b] *IBM Toronto Lab, 1150 Eglinton Ave., Toronto, Ont., Canada M3C 1H7*

Received August 1996; received in revised form March 1998
Communicated by G. Rozenberg

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. The problem is particularly acute when the underlying data structures are irregular, pointer-based structures. Here we present a methodology for partitioning a general class of dynamic data structures with guaranteed bounds on load-balancing and communication costs. Our method is based on a form of *graph grammar*, which specifies only families of graphs for which a "good" partitioning must exist. By modeling the construction and changes in a data structure using our formalism, one can quickly derive a good partitioning for a wide variety of common data structures. Moreover, expressing the structure updates in our grammars is generally a trivial operation with little overhead; this makes our approach particularly well-suited to dynamic situations. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Graph grammars; Irregular data structures; Partitionability; Parallelism; Dynamic data structures

Contents

* Corresponding author.
E-mail address: clarkv@ca.ibm.com (C. Verbrugge).
[1] This work was done while at McGill University.

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. Here we present a methodology for partitioning a general class of dynamic data structures with guaranteed bounds on load-balancing and communication costs. Our method is based on a form of *graph grammar*, which specifies only families of graphs for which a "good" partitioning must exist. By modeling the construction and changes in a data structure using our formalism, one can quickly derive a good partitioning for a wide variety of common data structures. Our approach is also particularly well-suited to dynamic situations – structures can be both constructed and updated by the same sort of grammar rules. The method is illustrated by first giving a precise definition of "good" partitionability as it pertains to parallelism, then proving all the graphs we generate do indeed have good partitions, and finally by showing how a wide variety of useful graphs can be produced using our formalism.

Our partitioning technique has a number of advantages over existing methods, which can be slow and/or require significant programmer interaction. The grammar is usually trivial to produce given a particular data structure and associated update methods, and we have observed little overhead in grammar specification. Our method is also very fast: once the structure has been constructed, the cost of partitioning for an arbitrary number of processors is at most linear in the number of graph nodes. Moreover, the partitionings so produced have a guaranteed quality, with specific bounds on the resultant communication cost and load balance. This makes our method quite viable, combining the speed and cost-guarantees of a problem-specific approach with the generality of heuristics, while maintaining a simple and straightforward specification and implementation.

Naturally, if graphs or data structures are restricted to being amenable to partitioning, then not all data structures will be expressible, and this is intended. The grammars we define, though, are general enough to express many common data structures: trees of course, threaded trees, trees where the leaves have sibling pointers, structured compiler control flow graphs, and with some extension, rectangular grids and other less "tree-like" structures. In fact, while the graphs we generate are more general than trees, the grammar specification defines an upper limit on *tree-width* [90]; thus they are all "tree-like" in a mathematical sense, and in a manner correlated with the grammar definition.

In the following section we formalize the notion of "partitionability" and describe the criteria we will use for measuring quality. Section 3 develops the basis for 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 extends this result to a larger class of graphs, showing how we can generate denser graphs with a corresponding sacrifice in partitionability. Section 7 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 8, where we establish limits on the "tree-width" of all graphs we generate. Finally, Section 9 contextualizes our method; we describe other approaches to managing irregular data structures for parallelism, and provide an overview of related work on graph grammars.

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 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), which is simply the number of edges "cut" to isolate any partition. For a given partitioning P let V_i be the set of nodes associated with the ith piece.

Definition 2. The *communication cost* of V_i is defined as

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

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 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 $1 \leqslant p \leqslant n$ 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.

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 [21]). 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. Nevertheless, and despite the relatively high lower bound on partitionability, it is sometimes desirable to generate such graphs explicitly.

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 often better dealt with using array-based methods. The fundamental dichotomy is therefore embodied in the following definition:

Definition 4. Let G be an f-partitionable graph. Then G is *reasonably partitionable* if $f \in \mathrm{O}(\log(n))$.

Remark. 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 Fig. 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

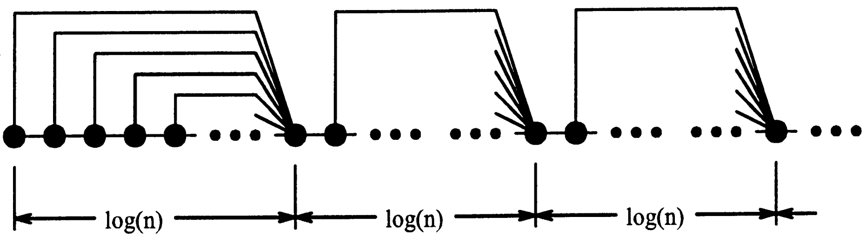

Fig. 1. A reasonably partitionable graph.

in its piece. This example is noteworthy for demonstrating that a reasonably partitionable graph can include an unbounded number of vertices with degree $\log(n)$.

The definition of partitionability is relatively straightforward; it is a considerably more complex task to algorithmically detect or specify reasonably partitionable graphs. In the subsequent sections, however, we develop a class of non-trivial graph grammars which *do* only express reasonably partitionable graphs.

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 5. A *dangling graph D* is an 8-tuple $(V, E, v, \phi, \psi, \Sigma_V, \Sigma_E, C)$, where:
- V is a set of *vertices* (or *nodes*),
- E is a set of $\frac{1}{2}$-edges,
- $v : E \to V$ is an injective function returning the vertex associated with a given $\frac{1}{2}$-edge.
- Σ_V is a finite set of node labels,

- Σ_E is a finite set of $\frac{1}{2}$-edge labels,
- $\phi : V \rightarrow \Sigma_V$ is a node labelling function,
- $\psi : E \rightarrow \Sigma_E$ is a $\frac{1}{2}$-edge labelling function, with the property that no two $\frac{1}{2}$-edges connected to the same vertex have the same label:

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

- $C \subseteq E \times E$ is a connection relation between $\frac{1}{2}$-edges, such that

$$\forall (e,e') \in C,\ \neg\exists (e,e'') \in C \text{ for } e'' \neq e',\ \text{and } (e',e) \in C.$$

In other words, C describes the connected pairs of $\frac{1}{2}$-edges, and is symmetric.

Remark. Like most definitions of graphs, 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 "$\frac{1}{2}$-edges", where each such $\frac{1}{2}$-edge is individually associated with a node through the v function. The connection is then actually established by pairing $\frac{1}{2}$-edges in the connection relation C. Any $\frac{1}{2}$-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* $\frac{1}{2}$*-edge*-labelled dangling graphs. Thus, there is an alphabet for both (Σ_V and Σ_E), and functions to map each node or $\frac{1}{2}$-edge to a node or half-edge label, ϕ and ψ respectively. Note that each $\frac{1}{2}$*-edge* has a label, including dangling ones, and thus each connection between nodes will have two labels, one for each $\frac{1}{2}$-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; $\text{Degree}(n) = |\{e \mid v(e) = n\}|$. If there exists a natural number k such that

$$\forall n \in V, \quad degree(n) \leqslant k,$$

then D is called a *k-bounded dangling graph.* It should be noted that since the set of $\frac{1}{2}$-edge labels, Σ_E is finite, and no two $\frac{1}{2}$-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, $\frac{1}{2}$-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 6. 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 a mapping between two dangling graphs, and thereby define possible ways of modifying any other graph. By locating an image of the first graph within a given graph, and replacing that image with a copy of the second graph the given graph can be changed. This can be formalized.

Definition 7. 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 $\delta : \Xi(S) \to \Xi(T)$ and $\delta^{-1} : \Xi(T) \to \Xi(S)$ 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 *embedding relation*, a partial mapping δ. This sequence of steps can be described formally using the following definitions:

Definition 8. A dangling graph S is a *subgraph* of another dangling graph D if

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

Remark. The subgraph relation is as one might expect; one defines a subset of the nodes, $\frac{1}{2}$-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* $\frac{1}{2}$-edges of each node included in the subgraph:

Definition 9. An *induced subgraph* of a dangling graph D is a subgraph D of D', such that

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

An *induced* subgraph D' is an *induced strict subgraph* if $D' \neq D$. In symbols, $D' \subseteq_i D$ and $D' \subset_i D'$, respectively.

Any subgraph or *induced* 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 10. 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

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

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 11. 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

$$\forall e \in E_1,\ v_1(e) = v \Rightarrow v_2(\beta(e)) = \alpha(v) \quad (e, e') \in C_1 \Rightarrow (\beta(e), \beta(e')) \in C_2.$$

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 *induced* 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 12. A dangling graph D' *occurs* in another dangling graph D if there exists $D'' \subseteq_i D$, 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 13. 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[2] 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 $\frac{1}{2}$-edges and vertices of T are disjoint from G, the

[2] 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.

derived graph H can be defined as follows:

$$\begin{aligned}
&V_H = (V_G - V_{S'}) \cup V_T, \quad && E_H = (E_G - E_{S'}) \cup E_T,\\
&v_H = v_G|_{E_G - E_{S'}} \cup v_T, && \Sigma_{H,V} = (\Sigma_{G,V} \cup \Sigma_{T,V}),\\
&\Sigma_{H,E} = (\Sigma_{G,E} \cup \Sigma_{T,E}), && \phi_H = \phi_G - \phi_{S'} \cup \phi_T,\\
&\psi_H = \psi_G - \psi_{S'} \cup \psi_T.
\end{aligned}$$

The connection relation is somewhat more complicated; if

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

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 $G \xrightarrow{*}_\Upsilon H$.

Remark. 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 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 $\frac{1}{2}$-edges in the source graph and the nodes and $\frac{1}{2}$-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* $\frac{1}{2}$-edge attached to that node.

An example of a production being applied is shown in Fig. 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 region enclosed on the input graph is the occurrence being rewritten. Node labels are illustrated by colour (shade), but $\frac{1}{2}$-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 $\frac{1}{2}$-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 $\frac{1}{2}$-edge designated by δ of the corresponding $\frac{1}{2}$-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 Fig. 2, three connection relations are transferred from the source to the target graph (indicated by dotted arrows), and any connections involving the other three $\frac{1}{2}$-edges are deleted by the rewrite.

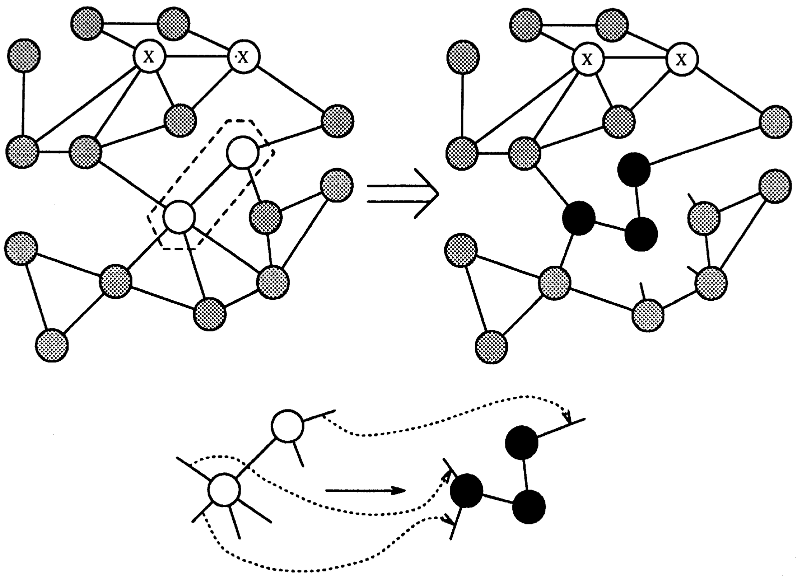

Fig. 2. A production (bottom) is applied to a graph.

Within a single derivation, because of the restrictions on how the target is embedded, the number of connection relations linking the embedded 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 14. *Given a production $\rho = (S, T, \delta)$ and a graph G to which ρ applies, $G \to_\rho H$, with S' the image (occurrence) of S in G as above, the number of connection relations linking the embedded 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 15. 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:

$$L(G) = \{B \mid A \xrightarrow{*}_{\Upsilon} B\}.$$

3.4. Grammar properties

Our ability to partition the graphs generated by our grammars will depend on the grammars having a property based on a concept of *overlap* between dangling graphs. This same concept, applied in a different manner, is often used to ensure concurrent rule applications can be done independently, and without conflict. While both overlap properties are restrictions on grammars, the combination has the benefit of being sufficient to sensibly extend our grammars to *parallel* grammars – ones wherein more than one production can be applied concurrently.

Definition 16. Two dangling graphs D and S *overlap* if there exist *induced* 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) 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 is easy to restrict a 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 $\frac{1}{2}$-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 17. 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 $\rho_1=\rho_2$ and $Overlap(S_1,S_2)$ is just the trivial overlap, then G is *SS-overlap free.*

Proposition 18. *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 $\frac{1}{2}$-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 $\frac{1}{2}$-edges in S and S' to which e is mapped, and let r and r' be the nodes attached to the other $\frac{1}{2}$-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.

If there is no such e then $W \equiv S_1 \equiv S_2$, and either there is certainly overlap, or $\rho = \rho'$ and W is the trivial overlap, in which case there is no non-determinism. □

Lack of overlap between source graphs is useful for parallelism, but it does not 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 (rather than between source and source) be restricted.

Definition 19. Let $G = (A, \Upsilon)$ be a dangling graph grammar, and let $\mathscr{T} = \{T \mid (S, T, \delta) \in \Upsilon\}$. G is *ST-overlap free* if for all $(S, T, \delta) \in \Upsilon$, we have

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

Remark. The ST-overlap free property 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, 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 20. 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 21. *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 ith largest number in N then $n_i \leqslant n/i$.*

Proof. By contradiction; assume n_i is strictly larger than n/i for some n, N and i. Since n_i is the ith largest, there are $i-1 \geqslant 0$ 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 21 to prove a cost bound on a certain kind of partitioning of trees. First, we define some essential terminology.

Definition 22. 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 23. A *postorder* tree traversal is a total ordering of the vertices of a tree such that if v_i represents the ith 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 24. *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 $\sum_i w_i = W \geqslant 1$. 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* $\mathit{Cost}(\cong_i) \leqslant (b-1)\log_2(W) + b$.

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 $b' \leqslant b$ 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 $b' \leqslant b$. So, assume at least one child tree is only partially visited, and that it is the jth 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 21, the jth 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 $1 < j \leqslant b$, and hence $\log_2(j) \geqslant 1$ and $j - 1 \leqslant b - 1$. Thus, the term $j - 1 - (b-1)\log_2(j) \leqslant 0$. 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 25. *If* T *is a tree with total weight* W *as described in Lemma 24 with the extra condition that for all weights* w_i, $w_i \leqslant m$ *for some* $m > 0$, *then for any* $0 \leqslant \omega \leqslant W$, *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* $0 \leqslant m' \leqslant m$, T_1' *has total weight* $\omega + m''$ *for some* $0 \leqslant m'' \leqslant m$, *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 $0 \leqslant m' \leqslant m$. Similarly, there must exist some j such that $w_1 + \cdots + w_j = \omega + m''$ for some $0 \leqslant m'' \leqslant m$. By Lemma 24, both partitionings have cost no more than $(b-1)\log_2(W) + b$. □

Lemma 24 and Corollary 25 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 $1 \leqslant p \leqslant W$. Given a tree T as in Lemma 24, 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 26. *Given an ordered list of n integers, $N = w_1, \ldots, w_n$, such that $0 \leqslant w_i \leqslant m$ and $W = \sum w_i$, N can be partitioned into $1 \leqslant p \leqslant W$ 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 $x \leqslant p$, 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 $0 \leqslant m' \leqslant m$ (the other case, $W' = Wx/p - m'$, is symmetric). By Corollary 25, N can be split either at $\omega + m_1$ or $\omega - m_2$, for any given $0 \leqslant \omega \leqslant W'$ and some $0 \leqslant m_1, m_2, \leqslant m$, 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. □

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

Corollary 27. *If T is a b-ary tree, as per Lemma* 24 *with an upper bound m on the weight associated with each vertex, then T can be partitioned into $1 \leqslant p \leqslant W$ 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 26, 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 25, 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 ST-overlap property is 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. Since contexts merely restrict the application of a rule, this property remains true even if we include contexts, and if we also include SS-overlap, then we find we can partition the graphs even if rule application proceeds in parallel.

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 $a \leqslant b$ applied to nodes a and b indicate that a is contained in the subtree rooted at b. Then,

Definition 28. A *tree partition scheme* for a dangling graph D with nodes V, $\frac{1}{2}$-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 $v : V \to N$ and a relation[3] $\tau \subseteq C \times L$ such that
(1) $\forall v \in V$, if $v(v) = n$ and $\ell : n' \to n$ and $v' \in V \setminus \bigcup_{\bar{n} \leqslant n} v^{-1}(\bar{n})$ and $e = (v, v')$ then $e\tau\ell$.
(2) $\forall n \in N$, $|\bigcup_{\bar{n} \leqslant n} v^{-1}(\bar{n})| > 0$.
(3) $\forall v, v' \in V$, if $v(v) = v(v')$ and $e = (v, v')$ then $\tau(e) = \emptyset$.

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

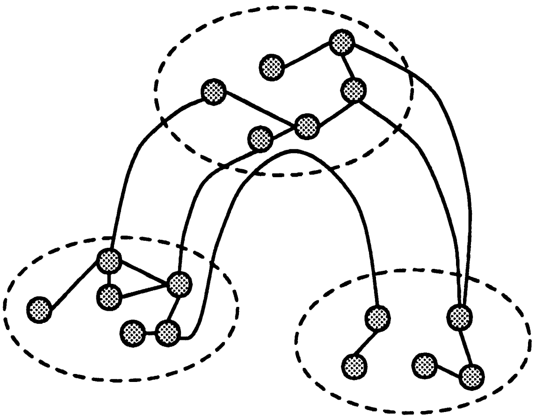

Fig. 3. A graph embedded into a tree partition scheme.

Remark. 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 Fig. 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 29. A tree partitioning scheme (T, v, τ) of a dangling graph D is said to be *bounded* if there exist three positive integers, β, μ, λ, where:
(1) β is a bound on the branching factor of T.
(2) μ is a bound on the size of $v^{-1}(n)$, $\forall n \in N$.
(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 30. *Let $T(v, \tau, \beta, \mu, \lambda)$ 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 27. $\square$

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 31. The *bounds* of a dangling graph grammar $G=(A, \Upsilon)$ are three positive integers, m, g, and k such that $|A| \leqslant m$, $\forall n \in V_A$, $degree(n) \leqslant k$, and $\forall (S, T, \delta) \in \Upsilon$ we have $|T| \leqslant m$, $|S| \leqslant g$, and $degree(v) \leqslant k$ 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 ST-overlap properties ensure that the tree partition scheme remains a tree after every set of concurrent rewrites.

Lemma 32. *Let $G=(A, \Upsilon)$ be an ST-overlap free dangling graph grammar, with no node rewritten by more than one production at once, and 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 $\forall n \in N$, $Fanout(n) + w_n \leqslant m$, 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 $|A| \leqslant m$, $|\nu^{-1}(n_1)| \leqslant m$, 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 no node is rewritten by more than one production, a function exists $\kappa : \mathcal{O} \to \{Z \mid Z \subseteq_i D\}$, which returns the embedded 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 $\sigma : \mathcal{O} \to N'$ exists associating occurrences with the tree node containing the vertices forming the occurrence.

We define v and τ to be the same as v' 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 v 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 $\zeta : \mathcal{O} \to N$. By assumption, $|\kappa(O)| \leqslant m$, and since each n_O has fanout 0, it is still true that $\mathit{Fanout}(n) + w_n \leqslant m$.
- *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 v. 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 $\mathit{Fanout}(n) + w_n \leqslant m$, 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 $C_O = \{(e,e') \in C_D \mid (e,e') \in \mathit{CSet}(\kappa(O), D - \kappa(O))\}$. 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 14). Since each occurrence consists of at most g graph nodes, of degree at most k (by assumption), there can be at most gk distinct[4] 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.
- *Fix-up 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 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 v from v' and τ from τ' are such that $|v^{-1}(v)| \leqslant m$, and $|\tau^{c}(\ell)| \leqslant gk$. 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

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

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 $\mathit{Fanout}(n) + w_n \leqslant m$, 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 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) induced subgraph of O including v with every vertex in O_v mapped by ν to $\nu(v)$. It must be that $O_v \in \mathrm{Overlap}(S_\rho, \tau)$; every $\frac{1}{2}$-edge of O_v is either a $\frac{1}{2}$-edge of $O \equiv S_\rho$, or it matches a $\frac{1}{2}$-edge of τ – a $\frac{1}{2}$-edge e of O_v which is not in $\Xi(O)$ and is connected to some other $\frac{1}{2}$-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 $\mathrm{Overlap}(S_\rho, \tau)$ 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 embedded 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 33. *Let $G=(A,\Upsilon)$ be an 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 be that D is $(2gk(m-1)\log_2(|V|)+2gkm)$-partitionable.*

Proof. This follows trivially from Lemmas 32 and 30. By the former, for each dangling graph generated by G there is a corresponding bounded tree partition scheme $T(v,\tau,m,m,gk)$, 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 $1 \leqslant p \leqslant |V|$. □

6. Denser graphs

There are often situations where one wants a schematic rewrite rule; that is to say, an infinite family of rewrite rules which exhibit a regular or repetitive pattern. For instance, we may wish to generate the family of rectangular grids (see Fig. 4).

If we build it row-by-row, then in order to ensure all the connections in the next row can be made (without overlapping rules) we would need an infinite family of rules, one for each of the possible number of a's. We would need one rule as in Fig. 5, one as in Fig. 6, and so on. It would certainly be easier to write one rule just indicating the pattern, as in Fig. 7.

Thus, instead of specifying source and target graphs precisely, we would like to specify source and target patterns. Patterns allow the generation of a larger class of

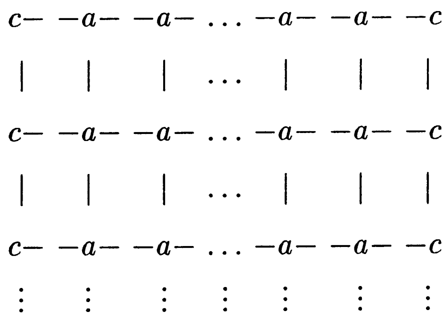
c— —a— —a— ... —a— —a— —c
c— —a— —a— ... —a— —a— —c
c— —a— —a— ... —a— —a— —c

Fig. 4. A schematic rectangular grid.

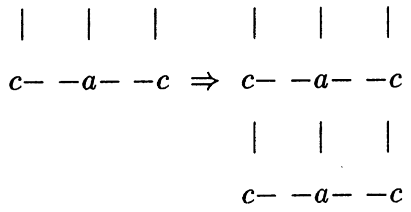

Fig. 5. One of an infinite family of rules.

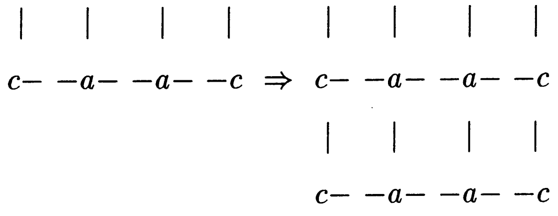

Fig. 6. Another of an infinite family of rules.

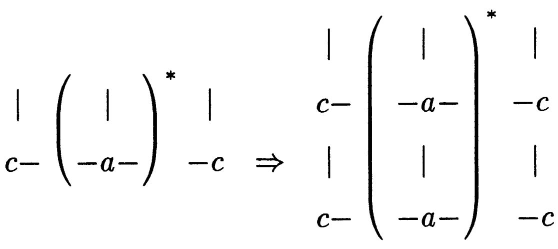

Fig. 7. A schematic rule, representing an infinite family of rules.

graphs; including, for example, the class of rectangular grids shown in Fig. 4. This class of graphs cannot be expressed using any bounded number of rules all of which have fully specified source and target graphs without introducing overlap. Naturally there is a tradeoff; the use of schematic rules implies an increase in the bound on partitioning cost – square grids are $\Omega(\sqrt{n})$-partitionable, a bound much higher than our previous $O(\log(n))$ limit. The following formalism for schematic graphs, called *path expressions* is designed to permit the increase in cost to be easily calculable.

6.1. Path expressions

A formalism for specifying the schematic representation of a family of graphs must be such that occurrences and the various forms of overlap between productions are still recognizable. For this reason, *path expressions* are based on an algorithmic model, similar to regular expressions on strings.

Path expressions are built up inductively from graphs and operators representing connection, choice and repetition. Each inductive operation indicates how one or two families of graphs with a given set of available unconnected $\frac{1}{2}$-edges (or "free edges") can be combined to generate another family of graphs. Note that this means that the operators must not only specify the appropriate action – connection, choice, repetition – but exactly which $\frac{1}{2}$-edges are to be connected to which others to actually form the desired structure. In the definition below, this function is provided by the partial bijection λ.

Definition 34. A *path expression* is defined inductively as follows:

(1) A dangling graph G of one node is a path expression. All $\frac{1}{2}$-edges are considered free.
(2) If G and H are path expressions with free edges $E=\{e_1,\dots,e_n\}$ and $F=\{f_1,\dots,f_n\}$, and $\lambda : E \leftrightarrow F$ is a partial bijection, then $(G \circ_\lambda H)$ is a path expression with free edges $\{x \mid (x \in E \wedge \nexists y \in F.\lambda(x)=y) \vee (x \in F \wedge \nexists y \in E.\lambda(y)=x)\}$.
(3) If G and H are path expressions with free edges $E=\{e_1,\dots,e_n\}$ and $F=\{f_1,\dots,f_n\}$, and $\lambda : E \leftrightarrow F$ is a partial bijection, then $(G|_\lambda H)$ is a path expression with free edges: $\{(e|f) \mid \lambda(e)=f\}$.
(4) If G is a path expression with free edges $E=\{e_1,\dots,e_n\}$, and $\lambda : E \leftrightarrow E$ is a partial bijection, then $(G+_\lambda)$ is a path expression with free edges E. Note that in this iterated graph there will actually be as many copies of each $\frac{1}{2}$-edge not involved in λ as there are replications of G, but that there will be only one copy of each $\frac{1}{2}$-edge which is involved in λ. For instance, if we have an expression like

$$\left(\begin{array}{c} |_p \\ n \\ {}_l/\backslash_r \end{array}\right)^{+_{r\to p}}$$

(which indicates a sequence of 1 or more nodes labelled n, connected r to p), then in any such sequence there is exactly one $\frac{1}{2}$-edge labelled p, one labelled r, and as many labelled l as there are nodes in the sequence.

The free set will be used below to establish bounds on the partitionability of graphs indicated by this method. For this reason it is essential that an unbounded number of $\frac{1}{2}$-edges does not get included in the definition. Thus, the free set for an iterated expression is defined to only include the (single) copies of each $\frac{1}{2}$-edge involved in λ, and the very first copy of any $\frac{1}{2}$-edge not involved in λ of the sequence.

The set of graphs indicated by a given path expression P forms the *language* of P, and is designated by $\mathscr{L}(P)$.

Note that we have not defined the usual "?" (match 0 or 1 instance of a graph) and "*" (0 or more repetitions) operators. Except for the ability to match the empty graph, this does not alter the expressiveness of the scheme. We have also not included "." (match any singleton); this could be included, but is simple syntactic sugar for the collection of all singleton graphs cascading |-ed together.

Example 35. Consider the following path expression.

$$\left(\left(\begin{array}{c} |_p \\ n \\ {}_l/\backslash_r \end{array}\right)\Bigg|_{\substack{p\to p\\ l\to r}}\left(\begin{array}{c} |_p \\ n \\ {}_l/\backslash_r \end{array}\right)\right)^{+_{(l|r)\to p}}$$

This expression generates a list of nodes, connected either l to p or r to p – the set of all paths in a binary tree from the root to any node.

The base case of a path expression is just a single dangling node. However, a path expression can also be thought of as composed from the three operators applied to fully-defined graphs, which are themselves constructed only from nodes and the $\circ$-operator. The next two definitions formalize this concept:

Definition 36. A path expression P is *concrete* if P consists entirely of dangling nodes and the '$\circ$' operator.

Definition 37. The *skeleton* of a path expression P is a function formed according to the syntactic expression of P with all concrete subexpressions removed. The skeleton of P, designated by "∂P", takes concrete path expressions as input, substituting them for the concrete expressions extracted from P. In order to ensure ∂P is unique, it must be that if a minimum of c concrete expressions must be removed from P so there are no more concrete expressions in P, then ∂P is a c-ary function, or of *order* c. The (ordered) list of c concrete expressions extracted from P is given by $[P]$, such that $\partial P([P]) = P$, with the i^{th} element in $[P]$ addressable by $[P]_i$.

Example 38. As an example, consider the following path expression[5] and its associated skeleton:

$$P = (a \circ b)|(((d \circ e)+)|((f)+)) \circ ((g)+)$$

$$\partial P(\#1, \#2, \#3, \#4) = (\#1)|(((\#2)+)|((\#3)+)) \circ ((\#4)+)$$

Hence, $[P] = (a \circ b, d \circ e, f, g)$ where $[P]_1 = a \circ b$, $[P]_2 = d \circ e$ and so on, and ∂P is of order four.

Some properties of path expressions should be immediately clear. For instance, any path expression normally written down by a human will have some constant bound on the size of the free set dictated by the "length" of the path expression. The length of a path expression is simply the number of nodes in the parse tree corresponding to the inductive definition; it can also be defined directly:

Definition 39. Given a path expression P, the *length* of P, given by $|P|$ is defined inductively:
(1) If P is a one node dangling graph, then $|P| = 1$.
(2) If $P = (G \circ_\lambda H)$, then $|P| = |G| + |H|$.
(3) If $P = (G|_\lambda H)$, then $|P| = \max(|G|, |H|) + 1$.
(4) If $P = (G+_\lambda)$, then $|P| = |G| + 1$.

[5] $\frac{1}{2}$-edges and $\frac{1}{2}$-edge labels are not shown.

Path expressions are adequate for describing simple linear structures, with limited branching. For instance, a path expression cannot be used to describe the class of binary trees. In fact, path expressions are all O(1)-partitionable; this is established using the following series of results.

Proposition 40. *Let P be a path expression of length ℓ over nodes with bounded degree k. Then the free set F of P is such that $|F| \leqslant k\ell$.*

Proof. By induction on $|P| = \ell$. If $\ell = 1$, then P matches only a single node of bounded degree k, and hence the free set is of size k.

Assume then that the hypothesis holds for all path expressions of length no more than $\ell - 1 \geqslant 1$, and let P be a path expression of length ℓ.

If P is of the form $(P_1 \circ_\lambda P_2)$, then $\ell_1 = |P_1|$ and $\ell_2 = |P_2|$ where $\ell_1 + \ell_2 = \ell$. By inductive assumption then, P_1 and P_2 have free sets of size $k\ell_1$ and $k\ell_2$ respectively, and by definition of '$\circ$' the free set of P is no more than the combination of the free sets of P_1 and P_2, which is of size $k\ell_1 + k\ell_1 = k\ell$.

If P is of the form $(P_1 |_\lambda P_2)$, then by definition of '|' the free set of P can be no larger than the smaller free set between P_1 and P_2; both of which are by inductive assumption of size no more than $k(\ell - 1)$.

Finally, if P is of the form $(P_1 +_\lambda)$, then the free set of P is identical in size to the free set of P_1, which by inductive assumption is no more than $k(\ell - 1)$. □

Lemma 41. *Let P be a path expression of length ℓ over nodes with bounded degree k. Then any graph $G \in \mathscr{L}(P)$ is $k\ell^2$-partitionable.*

Proof. By induction on ℓ.

If $\ell = 1$, then P is a single dangling node n, and partitioning is trivial. Assume then that the inductive hypothesis is true for all path expressions of length $\leqslant \ell - 1$, and let P be a path expression of length $\ell > 1$.

If P is of the form $(P_1 \circ_\lambda P_2)$, then the length of P_1 and P_2 will be ℓ_1 and ℓ_2, respectively, where $1 \leqslant \ell_1, \ell_2 \leqslant \ell - 1$. By inductive assumption then, any graph specified by P_1 or P_2 is $k(\ell - 1)^2$-partitionable. Moreover, by Proposition 40, there are no more than $k(\ell - 1)$ free edges emanating from (any graph specified by) P_1 to connect to P_2, and vice versa. To partition any graph specified by P it is sufficient to partition P_1 and then P_2; this can cost no more than the cost of partitioning the graphs specified by P_1 and P_2 plus the cost of severing the $k(\ell - 1)$ free edges between the two subgraphs at each partition. This is $k(\ell - 1)^2 + k(\ell - 1)$, which reduces to $k(\ell^2 - \ell)$, which is certainly no more than $k\ell^2$.

If P is of the form $(P_1 |_\lambda P_2)$, then to partition any graph specified by P it is sufficient to partition either P_1 or P_2. The bounds therefore follow trivially from the inductive assumption.

If P is of the form $(P_1 +_\lambda)$, then by inductive assumption P_1 can be partitioned with cost no more than $k(\ell-1)^2$. Since each copy of a graph specified by P_1 is connected to the next copy (if one exists) in the sequence by no more than $k(\ell-1)$ connections, and to the previous copy (if one exists) in the sequence by no more than $k(\ell-1)$ connections, any subsequence of images of P_1 can be disconnected from the rest of the sequence with cost no more than $2k(\ell-1)$. To disconnect any portion of an image of P_1 from the rest of its image can cost no more than $k(\ell-1)^2$, so disconnecting any portion of the graph has a maximum cost of $k(\ell-1)^2 + 2k(\ell-1)$, which reduces to $k(\ell^2-1)$, which is certainly no more than $k\ell^2$. □

Theorem 42. *Let P be a path expression with length and maximum degree bounded by a constant. Then any graph $G \in \mathscr{L}(P)$ is* O(1)*-partitionable.*

Proof. This follows immediately from Lemma 41. □

6.2. Path-expressions in productions

As with a normal graph specification, path expressions can be included as the source and target of productions. However, some structure is required if such a specification is to be sensible. It is not meaningful, for instance, for there to be a rule like

$$(a \circ b) \rightarrow (e|f).$$

In this case it is certainly not clear what the rule is telling us to do – should we replace the $a \circ b$ graph with an e node or an f node? Similarly, a rule such as

$$(a \circ b) \rightarrow (d \circ e)^+$$

does not provide enough information – how many iterations of $(d \circ e)$ should $(a \circ b)$ be replaced with?

Such problematic interpretations can be avoided by restricting the structure of the target path expression to be related to the path expression of the source. As long as the structure of the target is essentially the "same" as the source structure, modulo the specification of actual graphs, the transformation can be unambiguously based on the actual graph matched by the source.

Suppose we restrict the free sets at each inductive level of a path expression so only the $\frac{1}{2}$-edges actually used by an enclosing $\circ$, $|$, or $+$-operator are contained in the free sets. This way the free set at each level only includes "used" edges; any other $\frac{1}{2}$-edge not included in a free set is then certain to be dangling.

Productions using path expressions will then be formed from a collection of mappings between corresponding concrete subexpressions of the source and target expressions. The $\frac{1}{2}$-edges not found in the free set around each concrete expression (and which are therefore dangling) are used by the δ function in the same way as normal productions would. By splitting up the δ function among the individual mappings an effect similar to an interconnected collection of productions can be achieved, though it is also necessary to ensure these δ mappings do not conflict. The following definition formalizes these concepts:

Definition 43. Let S and T be path expressions, such that $\partial S = \partial T$. Let $[S] = (C_1, \dots, C_c)$ and $[T] = (D_1, \dots, D_c)$, for some c, with $F_1, \dots, F_c$ and $G_1, \dots, G_c$ their corresponding free sets. Let $\delta_1, \dots, \delta_c$ be a sequence of c partial bijections, such that $\delta_i : \Xi(C_i) \leftrightarrow \Xi(D_i)$, where $\forall i,\ \nexists e.\ (\delta_i(e) \in G_i) \vee (\delta_i^{-1}(e) \in F_i)$. Then if $\forall G \in \mathscr{L}(S)$ G is connected, $(S, T, \delta_1, \dots, \delta_c)$ form a *path-extended production.*

Remark. A skeleton, such as ∂S, specifies an algorithm. When S is actually matched with a graph, the choices made (such as which side of an $|$ to use, or how many iterations of a $+$-expression are needed) can be used to guide the actions of the ∂T algorithm, since $\partial S = \partial T$. This establishes the correspondence between concrete expressions in the source and in the target. One can view a path-extended production, then, as an interconnected series of regular productions between corresponding concrete subexpressions of the source and target.

6.3. Determinism and overlap

In order to ensure no two productions are attempting to rewrite the same node at the same time, the sources of any two productions must not overlap. Fortunately, a conservative answer is easily determined – although there is a concomitant reduction in expressibility.

A path-extended production can be viewed as a collection of regular productions between corresponding concrete subexpressions of the source and target. If for each path-extended production P we build such a set of regular productions $\widehat{P}$, then no two distinct productions in our original set of path-extended productions will rewrite the same node if all of $\widehat{P}$ is SS-overlap free. In other words, we have to extend the concept of "SS-overlap free" to path-extended productions.

Definition 44. Let $\mathscr{P}$ be a set of (path-extended) productions. Then $\mathscr{P}$ is *SS-overlap free* if the set

$$\widehat{\mathscr{P}} = \{([S]_i, [T]_i, \delta_i) \mid 1 \leqslant \imath \leqslant |[S]| \wedge \exists (S, T, \delta_1, \dots, \delta_{|[S]|}) \in \mathscr{P}\}$$

is SS-overlap free.

Ensuring the SS-overlap free property for path-extended productions means that no two different productions will attempt to rewrite the same graph node. However, this is still insufficient for actually ensuring *determinism*; the use of the iteration operator has not yet been fully-defined. For instance, the following path-extended production for a linear chain of a-nodes can match just one a, two a's, three a's, etc.

$$(-a-)^{+}.$$

Given a chain of a's as an axiom, we do not know how many of these expression should match, or where it should begin. We can alleviate some of the problem by demanding that there be only one occurrence of each path-extended rule at any one time. This ensures no node is rewritten more than once, but introduces the problem of picking which of all possible occurrences of a given path-extended production we should use. Even choosing the *largest* (in some order) occurrence possible, does not solve the problem – when matching our example to an axiom with a circular chain of a's, we still do not know where to start the occurrence. This can be dealt with by, for example, demanding each path-extended production include at least one node (in all possible graphs specified by the path-extended expression) which only appears once in each graph in the grammar language; this way a largest occurrence does constrain the possible matchings of a path-extended production.

The existence of such "anchors" can be ensured in a number of ways. Runtime resolution, dynamically verifying that no path-extended productions conflict, is the simplest, though most error-prone. To statically determine the existence of an anchor we first demand that the anchor, call it a, be identified in the path-extended production. Then, as long there is at most one a in the axiom or in the target of any production (in $\widehat{P}$), and each time a appears in the target of a production it also appears in the source, there will surely be only one a in any graph in the grammar language.

Proposition 45. *Let $\mathscr{P}$ be a set of* (*path-extended*) *productions. Then $\mathscr{P}$ is deterministic if $\mathscr{P}$ is SS-overlap free*, *path-extended productions match the largest possible occurrence* (*under some deterministic matching strategy*), *and each path-extended production includes a unique* anchor *in its source.*

Proof. This follows directly from the definitions of path-extended productions, anchors, and SS-overlap free. Because each path-extended production is anchored at a unique vertex and the matching is done deterministically, there is only one largest occurrence of each in the graph at any one time. The SS-overlap free property then ensures no two productions, path-extended or otherwise, interact. □

Remark. Note that the expression in Example 35 could not appear in the source of any production in an SS-overlap free set of productions. There are two concrete expressions,

$$\left(\begin{array}{c} |_{p} \\ n \\ {}_{l}/\backslash_{r} \end{array}\right) \text{ and } \left(\begin{array}{c} |_{p} \\ n \\ {}_{l}/\backslash_{r} \end{array}\right)$$

which trivially overlap. In order to state the partitioning properties of path-extended grammars, it will be convenient to reuse the ST-overlap free concept defined for regular grammars. This notion can be defined in a manner similar to that just used for SS-overlap free:

Definition 46. Let $\mathscr{P}$ be a set of (path-extended) productions. Then $\mathscr{P}$ is *ST-overlap free* if the set

$$\widehat{\mathscr{P}} = \{([S]_i, [T]_i, \delta_i) \mid 1 \leqslant \imath \leqslant |[S]| \wedge \exists (S, T, \delta_1, \ldots, \delta_{|[S]|}) \in \mathscr{P}\}$$

is ST-overlap free.

6.4. Modifications to the tree partition scheme

The ST-overlap free aspect of an ordinary dangling graph grammar ensures the existence of an associated tree partition scheme (TPS) for any graph in the language. With some modifications to deal with the $\frac{1}{2}$-edges in the free sets, this same concept can be used to generate TPSs for graphs generated by path-extended dangling graph grammars.

Each time a path-extended production P is applied, it is as if some number of distinct productions, $\widehat{P}$ (formed between concrete subexpressions of the source and target of P) were applied simultaneously. If we just consider the actions of $\widehat{P}$, then if $\widehat{P}$ is SS and ST-overlap free and has bounds (m, g, k), a bounded tree partition scheme $T(v, \tau, \beta = m,\ \mu = m,\ \lambda = gk)$ necessarily exists. If P maps G' to G then T is constructed from T', the TPS of G'. The path expression operators, though, permit connections also to be established between the embedded targets of productions in $\widehat{P}$ by linking $\frac{1}{2}$-edges in the free sets. These connections will not have been taken into account in constructing T.

Let $\overline{E}$ be the set of connections not considered in the construction of T. $\overline{E}$ can be included by modifying τ (the relation mapping connections to tree links) according to a simple observation about the elements of $\overline{E}$. If $e \in \overline{E}$ is a connection between embedded images of $[T]_i$ and $[T]_j$ resulting from a $+$-operator, then because $\partial S = \partial T$ for all productions and because any graph specified by S is connected, there is necessarily some connection e' between corresponding embedded images of $[S]_i$ and $[S]_j$ in G', found as a result of the application of a corresponding $+$-operator. The images of $[T]_i$ and $[T]_j$ are respective rewrites of the images of $[S]_i$ and $[S]_j$, so this implies that the existence of a connection between embedded images of $[S]_i$ and $[S]_j$ was already established in T'. Since T is an extension of T', the modifications to τ to include connections such as e can be expressed in terms of a simple expansion of τ'.

This still leaves $e \in \overline{E}$ which is not a connection between embedded images of $[T]_i$ and $[T]_j$ resulting from a $+$-operator. Fortunately, there can only be a fixed number of such connections in any graph specified by P, and so the number of such connections is bounded as a function of the length of P; specifically, there can be no more than kl such connections. The following results formalize this argument.

Proposition 47. *Let P be a path expression of length l over nodes with bounded degree k. Let C be the image of any concrete subexpression of P in a given $G \in \mathscr{L}(P)$. Then C is connected to G by no more than kl connections.*

Proof. Since P is of length l and C is the image of a fully defined graph within P, C can consist of no more than l nodes, each with degree bounded by k. Thus, there are no more than kl connections emanating from C. $\square$

Let $G=(A,\Upsilon)$ be an SS-overlap free and ST-overlap free path-extended dangling graph grammar with path-extended productions $\Upsilon'\subseteq\Upsilon$ where $u=|\Upsilon'|$. Let k be a bound on the degree of any node, and let l be a bound on the length of any path expression in Υ'. Let $(A,(\Upsilon-\Upsilon')\cup\widehat{\Upsilon'})$ have constant bounds (m,g,k), and let d be the maximum number of occurrences of concrete expressions in the occurrence of any path-extended production.

Lemma 48. *Let $G=(A,\Upsilon)$ be a path-extended dangling graph grammar as just described. Then for any non-empty dangling graph D such that $A\xrightarrow{s}_{\Upsilon}D$ for some s, there exists a bounded tree partition scheme $\mathscr{T}(v,\tau,\beta=m,\ \mu=m,\ \lambda\leqslant\max(\Delta,kl+gk))$ for $\Delta\leqslant\min((gk+kl)(kl)^s+u((kl)^{s+1}-1)/(kl-1)-1, gk+kl+ukls(d+1))$ such that $\forall n\in N$, $Fanout(n)+w_n\leqslant m$, 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, $v(v)=v(v')$.*

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

The base case, $s=0$, is of course trivial. 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 $\mathscr{T}'(v',\tau',\beta',\mu',\lambda')$ for D' with the above properties; we will show how to extend $\mathscr{T}'$ to a bounded tree partition scheme $\mathscr{T}$ for D.

The actions of the regular productions on the TPS have already been determined; assume then that a new TPS, $\mathscr{T}$, has been constructed from $\mathscr{T}'$ according to the actions of $(A,(\Upsilon-\Upsilon')\cup\widehat{\Upsilon'})$ and as described in Lemma 32, with two exceptions. First, no dead branch elimination has been performed; and second, all existing connections in G' which were identified with a connection established between free sets of any path-extended productions in the rewrite have been retained in τ. This latter condition means τ is still associating some connections which do not exist anymore in G with $\mathscr{T}$, but it does not increase the bound λ.

Such a construction ensures that β does not increase beyond m, μ does not increase beyond m, and that λ remains bounded by $\max(\lambda',gk+kl)$; as well, since dead branches have not been pruned, all nodes and links of $\mathscr{T}'$ are contained in $\mathscr{T}$. We will now show how to integrate the extra connections implied through the path-extended productions. There are three separate kinds of connections to establish.

(1) Each time a path-extended production $\rho=(S,T,\delta_1,\ldots)$ is applied, it is as if all of $\widehat{\rho}$ were applied with extra connections established between productions in $\widehat{\rho}$. Each one of these productions is between two concrete subexpressions of ρ, and thus by Proposition 47 the embedded image of any target in $\widehat{\rho}$ in D can be disconnected

with cost at most kl. Hence, if O is the image of $[S]_i$ for some i and $\kappa(O), n_O$ are the tree nodes (and by construction $\kappa(O)$ must exist in both $\mathscr{T}'$ and $\mathscr{T}$) associated with O and the corresponding embedded image of $[T]_i$, then τ can be increased to map the entire connectivity of the embedded image of $[T]_i$ to the tree link $(\kappa O, n_O)$. This amounts to λ being no more than kl for those links.

(2) If e is a connection established during the rewrite by the actions of a $+$-operator of T, then as discussed above there is some corresponding connection in S, and therefore there is some corresponding connection e' in G'. In other words, if e arises from a $+$-operator and connects the images of $[T]_i$ and $[T]_j$ in G, then there exists some e' connecting the corresponding images of $[S]_i$ and $[S]_j$ in G'. Moreover, our second exceptional requirement of $\mathscr{T}$ stipulates that τ still maps connections like e' to links of $\mathscr{T}$. We can replace each such e' with at most kl connections each application of each path-extended production. This implies a multiplicative increase in λ for any existing link of no more than kl. However, it is also true that no more than ukl connections can be introduced to any tree link in order to connect occurrences of the concrete parts of any occurrence of a path expression. Since there are at most d of these concrete occurrences for each path-extended production, the increase in λ is also bounded by an additive factor of $udkl$ for these connections.

(3) Again, this leaves the consideration of $e \in \overline{E}$ which is not a connection between embedded images of $[T]_i$ and $[T]_j$ resulting from a $+$-operator. As mentioned there are at most kl such connections, and so τ will have to associate at most kl more connections with any given link in $\mathscr{T}$ in order to accommodate them for each of the u path-extended productions.

Thus, all connections can be included in τ with an increase in λ of at most ukl for the latter connections, and either a multiplicative increase by kl or an additive increase by $udkl$. Applying these increases to the inductive hypothesis results in the described bounds. □

Theorem 49. *Let $G = (A, \Upsilon)$ be a path-extended grammar as just described. Then for any dangling graph D such that $A \xrightarrow{s}_{\Upsilon} D$, it must be that D is $O(c^{s+1}\log(|V|))$-partitionable, and $O(sd\log(|V|))$-partitionable.*

Proof. Lemma 48 establishes the existence of a tree partition scheme where λ is bounded as described. By Lemma 30 this TPS can be partitioned with cost $O(\log(|V|))$. □

7. 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.

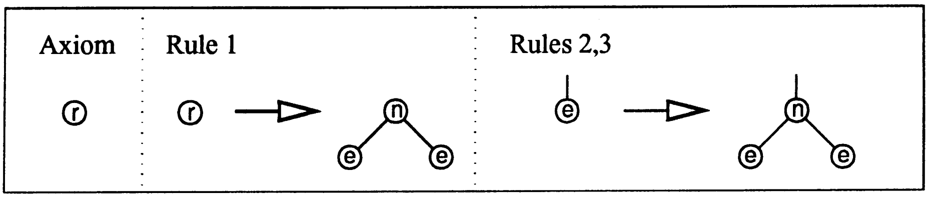

Fig. 8. A grammar generating trees.

7.1. Reasonably-partitionable structures

We have evinced two forms of grammar; one where the source and target of each production must be fully-defined, resulting in O(log(n))-partitionable graphs, and one where the source and target are specified through path expressions, resulting in O(s log(n))-partitionability for an s-step sequential derivation. Here we illustrate some possible grammars falling into the former category.

7.1.1. 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 $\frac{1}{2}$-edge connecting the rewritten leaf to its parent is associated with the $\frac{1}{2}$-edge extending from the internal node. This is illustrated in Fig. 8;[6] Rule 1 expands the root node, and Rules 2 and 3 (shown as one rule – there should actually be two rules, one if the e-node is a left child of its parent, and a symmetric one if e is the right child) expand a left child or a right child.

7.1.2. Threaded k-ary trees

By adding two $\frac{1}{2}$-edges to each node, for left and right threaded neighbours, threading can be maintained as the leaves are expanded (see Fig. 9; threading is shown using dashed lines). 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 (Fig. 10). Our examples illustrate binary trees and inorder threading, but clearly it is possible to accommodate any recursive threading policy in the same way.

7.1.3. 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 (Fig. 11). If the list is intended to be circular,

[6] Edge labels and the δ function are not illustrated; they should be obvious from the geometric positioning of the $\frac{1}{2}$-edges.

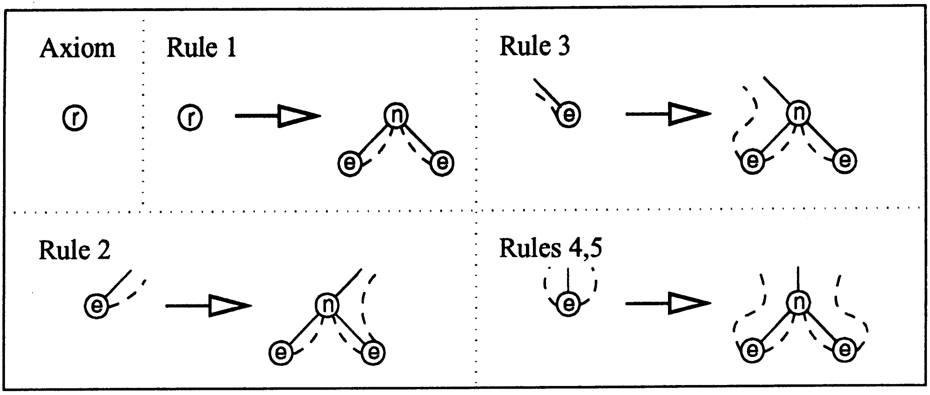

Fig. 9. A grammar generating inorder threaded (binary) trees.

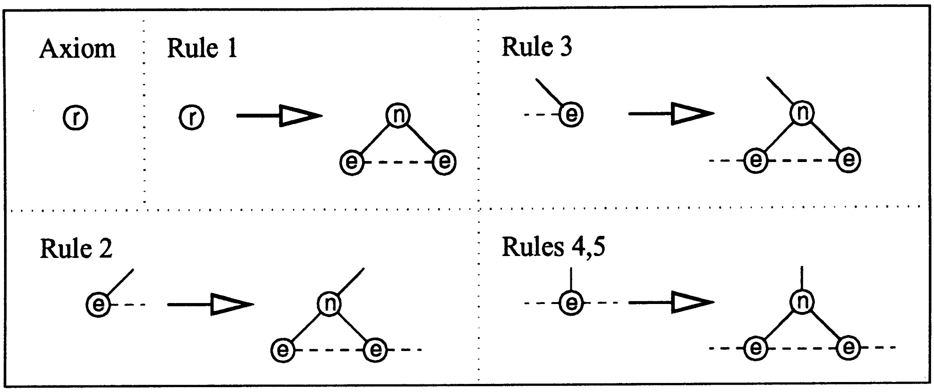

Fig. 10. A grammar generating leaf-threaded trees.

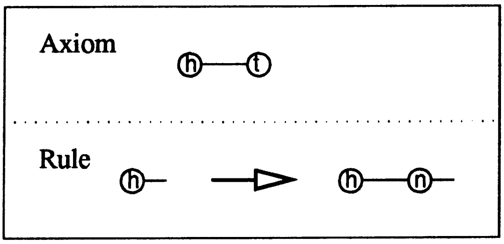

Fig. 11. A grammar generating linked lists.

then an extra connection between the head and tail is maintained through the rewrites (Fig. 12). Different orders of application for the rules then correspond to the different variations on lists – stacks, queues, double-ended queues, etc.

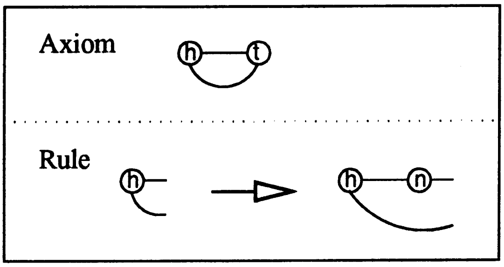

Fig. 12. A grammar generating circular linked lists.

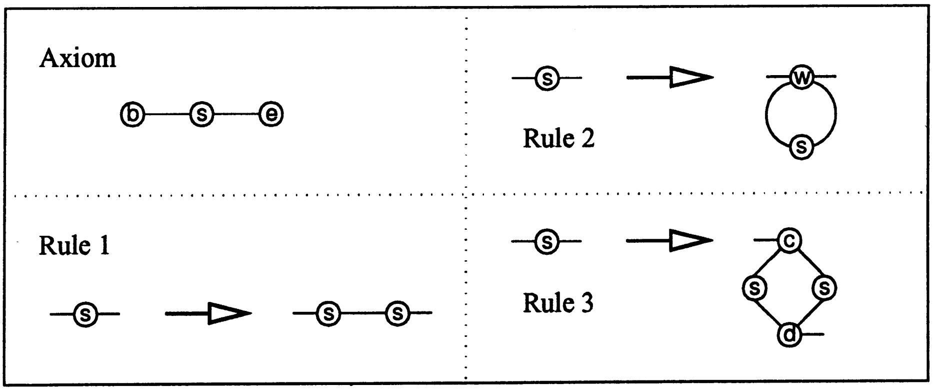

Fig. 13. A grammar generating compiler control flow graphs.

7.1.4. 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.

Such a grammar is shown in Fig. 13. The axiom thus consists of a single statement node bracketed by a begin and end marker. Rules exist to expand a statement node into a pair (or more) of statement nodes (Rule 1), into a loop statement consisting of a cycle including a statement node (Rule 2), or into a conditional, consisting of a cycle with true and false branches, with the conditional exit continuing control flow out of the conditional (Rule 3).

7.2. Path-extended grammars

Generating dense graphs is performed with a bound on partitionability proportional to $sd\log(n)$, where s is the number of times a path extended production applied and d is the maximum number of occurrences of concrete subexpressions in any path-extended occurrence, and an exponential bound as well. Thus, the number of times a

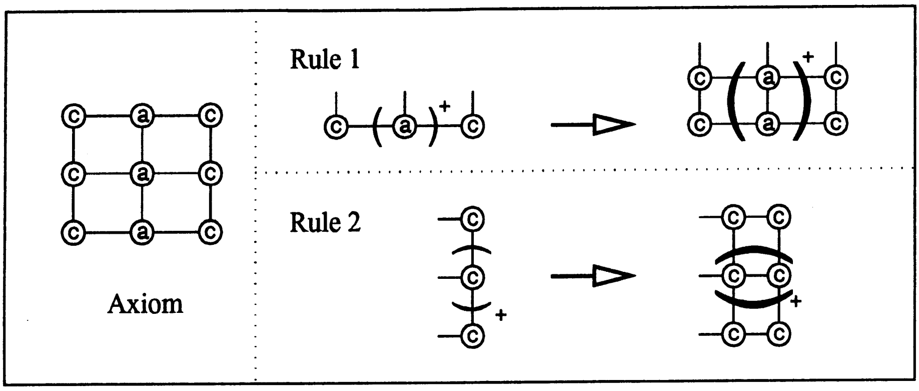

Fig. 14. A path-extended grammar generating rectangular grids.

set of productions can be applied is an important factor in these grammars. Below are a few interesting graphs which can be produced with this scheme, along with their actual partitionability bounds.

7.2.1. Rectangular grids

Rectangular grids are one of the more difficult classes of graphs to express using graph grammars; generating a rectangular grid requires either overlapped productions or coordinated action between productions, neither of which is possible with a normal dangling graph grammar. With path-extended productions, though, the process is quite straightforward (see Fig. 14). The axiom is an initial minimal grid, and there are only two rules. Rule 1 expands the width of a rectangular grid by one column, and Rule 2 rule expands the height by one row.

Our partitionability bounds as given by Theorem 49 are far from optimal in this case. Our exponential or length-driven bounds do not compare to the actual $\Theta(\sqrt{n})$ bounds on partitionability. In this case, though, the upper-bound on partitioning the TPS from which these bounds are derived is misleading. If the grid is produced by first generating the width and then the height, the TPS inductively constructed according to Lemma 32 will look isomorphic to

$$\left(\left(\begin{array}{c} |_p \\ n \\ {}_l/\backslash_r \end{array}\right)^{+_{l\to p}}\right)^{+_{r\to p}}$$

Each application of the path expression then increase the number of connections mapped to each tree link in the TPS by only a *constant* amount. Moreover, since the TPS itself is a path expression, by Theorem 42 the TPS can be partitioned with cost O(1) – reducing the cost of partitioning the grid to $O(\sqrt{n})$.

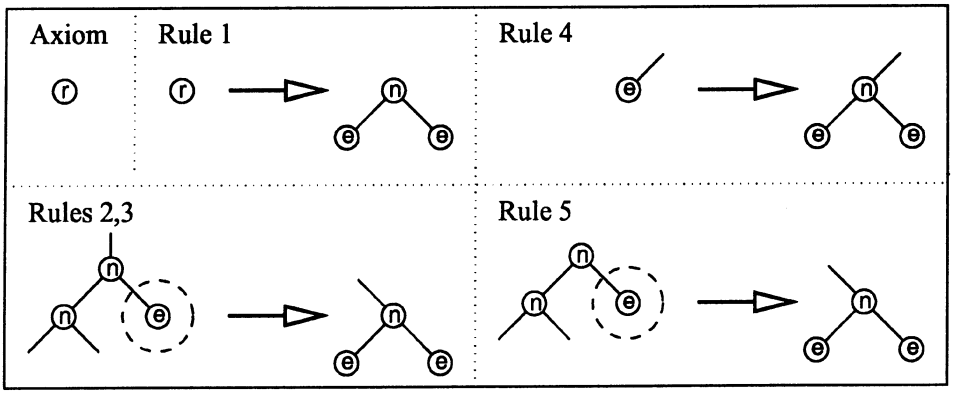

Fig. 15. A grammar requiring contexts; right leaves are expanding only if the left sibling is not a leaf.

7.3. 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 Fig. 15).

8. 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 [4, 5, 13, 64, 96]. 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 50. Let $T=(N,L)$ be a tree with nodes N and edges L. Then $\pi : N\times N \rightarrow \mathscr{P}(N)$ is a function returning the set of vertices forming the loop-free path connecting the two input vertices.

Definition 51. Let $G=(V,E)$ be a graph. A *tree-decomposition* of G is a tree $T=(N,L)$ and a function $\sigma : N \rightarrow \mathscr{P}(V)$ such that
(1) $\forall (v,v')\in E,\ \exists n\in N.\ v,v'\in\sigma(n)$.
(2) $\forall n,n'\in N,\ v\in\sigma(n)\cap\sigma(n') \ \Rightarrow\ \forall \widehat{n}\in\pi(n,n')\ v\in\sigma(\widehat{n})$
(3) $V=\bigcup_{n\in N}\sigma(n)$.

Let m_T be the maximum cardinality of any $\sigma(n)$ in T; *ie*., $m_T = \max|\sigma(n)|$ 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.

Lemma 52. *Let $D=(V,E,v_D,\phi,\psi,\Sigma_V,\Sigma_E,C)$ be a dangling graph, and let $T(v_T,\tau,\beta,\mu,\lambda)=(N,L)$ be a bounded tree partition scheme for D. Necessarily D has tree-width smaller than $2\beta\lambda+\mu$.*

Proof. We will construct a function σ which together with (N,L) forms a tree decomposition of D. Let $l(n)$ be the set of outgoing links attached to a node n in N. Define $\sigma : N \to \mathscr{P}(V)$ as:

$$\begin{aligned}\sigma(n) = &\{v\in V \mid v_T(v)=n\}\\ &\cup \{v\in V \mid \exists e,e'\in E,\ \exists l\in l(n).\ v_D(e)=v \wedge (e,e')\tau l\}.\end{aligned}$$

The function σ maps each tree node n in T to the set of vertices which are mapped by v_T to n, or which are included in a connection relation which is mapped by τ to an outgoing edge attached to n. We now verify that σ has the tree decomposition properties.

Let $\varepsilon : N \to \mathscr{P}(V)$ be a function returning the subset of V corresponding to a given node (subtree) in T. Note that if and only if $n \in \text{Subtree}(n')$ for two tree nodes n,n', then $\varepsilon(n) \subseteq \varepsilon(n')$. Also note that by definition of TPS, a vertex v is in $\varepsilon(n)$ for a tree node n if and only if $\exists n' \in \text{Subtree}(n).\ v_T(v)=n'$.

By definition of T, every vertex in V is already uniquely mapped by v_T to some node in N, so certainly σ covers the vertices of D. Every connection relation c in C 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 v_T to the same tree node. In the latter case, let $v,v' \in V$ be the two vertex endpoints. At least one of the subtrees rooted at $v_T(v)$ or $v_T(v')$ must not contain the other, and so c is a connection relation which must be broken to separate $\varepsilon(v_T(v))$ from $\varepsilon(v_T(v'))$. Now, by definition of T, c is associated by τ to each link along $\pi(v_T(v),v_T(v'))$, which must be a chain of at least 2 nodes. Thus, by construction of σ, there will be a node n with both $v\in\sigma(n)$ and $v'\in\sigma(n)$.

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 it is also contained in $\sigma(\widehat{n})$ for each $\widehat{n}$ along the simple path between n and n'. Let v,n,n' be a vertex and two nodes in such a situation. Vertex v is mapped by σ to node n (and to node n') for one of two reasons: (1) $v_T(v)=n$, or (2) some connection to v is associated by τ to an outgoing link of n.

(1) $v\in\varepsilon(n)$, $v\in\sigma(n')$ by reason (2). This means that in order to partition $\varepsilon(m')$ for some child m' of n' it is necessary to cut a connection c to v; or, equivalently, only one of v and some neighbour v' of v is in $\varepsilon(m')$.

  (a) $v\in\varepsilon(m')$. Then $\varepsilon(n)\subseteq\varepsilon(n')$ and $\forall\widehat{n}\in\pi(n,n')\ \varepsilon(\widehat{n})\subseteq\varepsilon(n')$. The connection c must then be broken to partition any $\varepsilon(\widehat{n})$, and is thus mapped by τ to all links along $\pi(n,n')$. By definition of σ, v will then be mapped to all nodes in $\pi(n,n')$.

(b) $v' \in \varepsilon(m')$. Then either $\varepsilon(n') \subseteq \varepsilon(n)$ or $\varepsilon(n)$ and $\varepsilon(n')$ are disjoint.

(i) $\varepsilon(n') \subseteq \varepsilon(n)$. There exists a node $\widehat{n} \in \pi(n',n)$ such that $\forall \widehat{n}' \in \pi(\widehat{n},n)$ it is the case that $v \in \varepsilon(\widehat{n}')$, and $\forall \widehat{n}' \in (\pi(n',\widehat{n}) - \{\widehat{n}\})$ we have $v \notin \varepsilon(\widehat{n}')$. Each of the former must be such that either $v_T(v) = \widehat{n}'$ or there is some child $\widehat{m}'$ with $v \notin \varepsilon(\widehat{m}')$ or there is some child $\widehat{m'}$ with a neighbour v'' of v outside $\varepsilon(\widehat{m}')$. In all situations $\sigma(\widehat{n}')$ will include v. Each of the latter cases must have the same connection c between v' and v cut to partition $\varepsilon(\widehat{m}')$, for some child $\widehat{m}'$, and so $\sigma(\widehat{n}')$ will include v.

(ii) $\varepsilon(n)$ and $\varepsilon(n')$ are disjoint. Then τ must map c to the link between any $\widehat{n} \in \pi(n,n')$ and its child $\widehat{m} \in \pi(n,n')$ where $\varepsilon(\widehat{m})$ contains only one of $\varepsilon(n)$ or $\varepsilon(n')$. Hence, by definition of σ, v is in $\sigma(\widehat{n})$.

(2) $v \notin \varepsilon(n)$, $v \in \sigma(n')$ by reason (2). Then $v \in \sigma(n)$ by reason (2) as well, and $\varepsilon(m)$ for some child m of n contains a neighbour v' of v. If $v \in \varepsilon(n')$ then of course the situation is symmetric to case 1, so we can assume $v \notin \varepsilon(n')$. Any $\widehat{n} \in \pi(n,n')$ such that $v \notin \varepsilon(\widehat{n})$ must have a child containing either (or both) v and v', so certainly $v \in \sigma(\widehat{n})$. If some $\widehat{n}$ does contain v then the situation is symmetric to a sub-case of case 1.

This establishes that the tree T and function σ represent 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 β, the number of connection relations mapped to a given tree link is bounded by λ, and μ is a bound on the number of vertices mapped by v_T to any node, there will never be more than $2\beta\lambda + \mu$ vertices of D mapped by σ to any single tree node in T. □

Theorem 53. *Let G be an ST-overlap free dangling graph grammar with constant bounds (m,g,k). Let $D = (V,E,v,\phi,\psi,\Sigma_V,\Sigma_E,C)$ be a dangling graph such that $A \xrightarrow{*}_{\Upsilon} D$, necessarily D has tree-width smaller than $2mgk + m$.*

Proof. By Lemma 32 a bounded tree partition scheme $T(v_T, \tau, \beta = m, \mu = m, \lambda = gk)$ exists for D. By Lemma 52 this implies an upper bound on tree-width of $2mgk + m - 1$. □

Theorem 54. *Let G be an ST-overlap free path-extended dangling graph grammar with path-extended productions $\Upsilon' \subseteq \Upsilon$ where $u = |\Upsilon'|$. Let ℓ be a bound on the length of any path expression in Υ', let d be the maximum number of occurrences of concrete subexpressions in any occurrence of a path-extended production, and let $(A, (\Upsilon - \Upsilon') \cup \widehat{\Upsilon'})$ have constant bounds (m,g,k). If $D = (V,E,v,\phi,\psi,\Sigma_V,\Sigma_E,C)$ is a dangling graph such that $A \xrightarrow{s}_{\Upsilon} D$, then necessarily D has tree-width smaller than $2m\max(\Delta, k\ell + gk)) + m$ where $\Delta \leqslant \min((gk + k\ell)(k\ell)^s + u((k\ell)^{s+1} - 1)/(k\ell - 1) - 1, gk + k\ell + uk\ell s(d+1))$.*

Proof. By Lemma 48 a bounded tree partition scheme exists for D with the given bounds. By Lemma 52 this implies the upper bound on tree-width. □

Corollary 55. *Rectangular grids of size $w \times h$ have tree-width of* $O(w + h)$.

Proof. By construction of the grammar in Fig. 14, we find that the bounds on the grammar are all small constants: $m = 9$, $u = 2$, $k = 4$, $\ell = 9$, $g = 2$. Since any rectangular grid can be generated by this grammar in $w + h$ steps, $s = w + h$. By Lemma 52, and the construction in Section 7.2.1 the upper bound on tree-width follows. □

Corollary 55 jibes nicely with existing results; it is known that square grids of $\sqrt{n} \times \sqrt{n}$ vertices ($n \geqslant 2$) have a tree-width of $\sqrt{n}$ [90].

9. Related work

In the interests of generality, we have investigated graph partitioning under the assumption that any number of partitions may be demanded. However, related problems such as determining the minimum number of edges to be cut to separate a graph into just k partitions for a fixed k, or determining the smallest set of vertices separating the graph into two partitions with no edges between them, have been examined extensively. Both problems are NP-complete in general, but have tractable versions for specific classes of graphs. In 1979, for instance, Lipton and Tarjan [75] solved the latter problem for planar graphs, by showing that all planar graphs have a set of $O(\sqrt{n})$ separator vertices; an extension of this classic result to graphs of fixed genus is available [97] and other variations have been explored [12, 21] from the viewpoint of graph embedding. Unfortunately, these results do not easily transfer to the p-partitioning problem for any arbitrary p, nor do they tend to produce partitionings with the tight balancing and communication costs we require.

One of the simpler structures to partition is of course trees, and there have been a variety of different approaches. Lukes gives an efficient algorithm based on dynamic programming for finding *connected* partitions of trees [78]. A similar problem has been looked at by Kundu and Misra, who give a linear algorithm for finding an optimal cut partitioning, where each subtree contains at most a given number of nodes, though there may or may not be a total of k partitions produced [67]. While these algorithms are interesting and have some similarity to our tree partitioning algorithm of Section 4, we do not necessarily require connectivity, and we do require there to be a given number of partitions.

Heuristic attacks on partitioning problems abound. Perhaps the most well-known is the Kernighan-Lin heuristic [62]: a graph is first partitioned arbitrarily, and the partitioning is then improved by exchanging vertices between partitions. This technique was later extended by Fiduccia and Mattheyses [38]. Feo and Khellaf have developed a heuristic based on a recursive pair-wise grouping of nodes for k-partitioning when

k is large [36]. More recent methods include Spectral Bisection [46], Simulated Annealing [59], and others; heuristic combinations of such methods have also been quite successful [19, 60, 98]. We of course are interested in deterministic methods; recently, direct approaches that find cuts within a specific bound of the optimal have been investigated; for instance, Saran and Vazirani find minimum k cuts within at most $(2-2/k)$ of the optimal [92]. While polynomial ($O(n^p)$ for some p), direct methods such as these are still too expensive for dynamic graphs.

Data partitioning explicity for irregular problems has also been explored. Nakanishi et al. [83], for instance, develop a "Heirarchical Data Partitioning" graph, incorporating a hierarchical representation of control flow and control-flow dependencies. This is a general model for partitioning that does not attempt to utilize any structure-specific information, and does not include cost bounds. Naturally, better results can be obtained if even general characteristics of such algorithms are known; this is the approach taken by Gautier, Roch and Villard [41]. They identify several programming paradigms for dealing with irregular problems in order to produce a classification scheme. Their efforts are meant to facilitate either manual or automatic load-balancing and not to actually specify such algorithms. Das et al. [18] also give a generic approach through the presentation of hardware designed to support irregular data accesses, called the "Intersecting Broadcast Machine". By distributing data randomly, and maintaining multiple copies of data through broadcasts, they can show (experimentally) very good load-balancing and processor utilization; however, their method is stochastic, rather than deterministic. Alternatively, there are many algorithms for tackling specific problem areas: backtracking search trees [91], Finite Element Methods on irregular domains [9, 20, 43, 98], particle systems [79], etc. Most of these make use of heuristics or randomized techniques, such as greedy graph clustering [35], simulated annealing and recursive bisection.

9.1. Analyzing irregular data structures

One possible approach to irregular data structures is to find some way to express them that makes their actions more predictable. We argue that most irregular data structures are simple variations on well-known structures; our graph grammar systems can be seen as a method of making irregular data structures more "regular", and hence analyzable. There have been a few similar approaches, with various goals in mind.

The *Abstract Description of Data Structures* (*ADDS*) formalism of Hummel et al. [48] falls into this category. Recursive data structure definitions are augmented by a set of keywords defining the general shape (*via* interacting *dimensions*), and the intended traversals as well. A doubly linked list, for instance, might be specified as consisting of two dimensions, one uniquely forward along the `next` pointer and the other backward along the `previous` pointer. The emphasis here is on increasing the compiler's ability to perform automatic error detection, optimizations, and fine-grain parallelization, and not to dictate graph structure.

A similar linguistic approach is given by Gupta [44], with intent to extend SPMD-style parallelism to dynamic data structures. Data structures may be *local* or *distributed,* and distribution and naming strategies are user-specified (default strategies exist too). While quite flexible, this approach still requires the programmer to define the partitioning (or accept the default), and is primarily a descriptive rather than prescriptive approach. The emphasis here is on correct and fast execution of the run-time system, and not on ensuring the quality of the partitions.

Klarlund and Schwartzbach [63] have developed an extension to data types by appending *routing expressions* to recursive type definitions: a spanning tree is specified using the normal recursive definition, and a regular string expression over edge labels and simple node predicates (such as "this is a leaf") is allowed to dictate further connectivity. This allows the expression of recursive data structures which do not strictly form trees, but without the generality of arbitrary and explicit pointers. Like our path expressions, their routing expressions are based on a generalization of regular expressions on character strings, though their formalism includes logical decision operators as well as simple pattern matching. This model is directed toward facilitating automated reasoning about pointer structures, though and not partitionability – graph types exist for structures that are relatively expensive to partition, like a binary tree with all leaves pointing to the root.

9.2. Graph grammars

We have used graph grammars as the basis for our representation of data structures. Because of their flexibility and necessary formalization of rewrites, graph grammars are an attractive model for data structure development; by eliminating many of the problems associated with pointers, such as the inevitable temporary inconsistencies as pointers are updated, graph grammars are able to represent data structures and modifications in a way that tends to make analyses and interpretive results considerably more feasible than with pointers. Our method constitutes a novel use of graph grammars even within this context; however, we are certainly not the first to use such a formalism to represent data structures. In the text that follows we offer a brief synopsis of other work on graph grammars, followed by a description of how they have been applied to algorithm and data structure development.

A paper by the ESPRIT Basic Research Working Group No. 3299 [84] gives a history of the different forms and directions of research into graph grammars. There have been a wide variety of approaches. The "algebraic" method of Ehrig and Schneider and Löwe [8, 26–28, 77], also known as the "Berlin Approach", concentrates on Categorical representations of graph grammars. This primarily theoretical body of work allows for the specification of properties that permit concurrent application and amalgamation of rules, in a non-specific setting. Courcelle [13, 15] gives another abstract approach based on the logical interpretation of graphs. By showing that various graph properties cannot be expressed in certain logical languages, he is able to define an expressiveness heirarchy, and relates this to a specific form of graph grammar (hyperedge replacement grammars).

More concrete definitions and results also exist. One of the first and most successful (i.e., long-lived) forms of graph grammar is the “Node Label Controlled” (NLC) formalism of Janssens and Rozenberg [50]. Here each production rewrites a single node to an arbitrary graph (of fixed size), and connections are established based on a connection (embedding) relation (a set of pairs of node labels); each time a rewrite is performed, nodes in the newly embedded graph are hooked up to the nodes surrounding the original rewritten node based on the pairs. There is only one connection relation for all productions. Janssens and Rozenberg have shown this model to be quite robust under many variations [51], and have used this as the basis for an expressiveness heirarchy [57, 58].

The restricted form of the embedding relation in NLC grammars can be inconvenient. “Neighbourhood controlled embedding” (NCE) grammars generalize this function, allowing each rule to specify its own embedded relation [53, 54]. NCE grammars also permit the left-side of each rule to be an arbitrary graph, not just a single node. These would seem to be extensions that would make NCE grammars strictly more powerful than NLC, and this is true of general NCE; however, if NCE grammars are constrained to have just one node on the left-side of each rule, “1-NCE” grammars, then it turns out NLC is just as powerful: $\mathrm{NLC} = 1 - \mathrm{NCE} \subset \mathrm{NCE}$ [54]. This makes NCE grammars a particularly flexible model.

There have been numerous variations on NCE and a readable introduction to the different forms of NLC and NCE grammars is given by Engelfriet and Rozenberg [34]. “dNCE” extends NCE to directed graphs [52], and “eNCE” includes edge-labels into NCE [11, 30]; the combination, “edNCE”, having both. The most useful extension seems to be “C-edNCE”, or *Confluent* edNCE grammars [31, 32]. Confluence in this context means that the order of rule applications is unimportant – any order generates the same graph (confluence is defined formally by Courcelle in [14]). This sort of determinism is useful for reasoning about expressiveness (and for parallelism in rule applications), and seems to produce a fairly natural class of grammars; it has been shown that C-edNCE grammars generate languages which can be characterized in terms of Monadic Second-Order Logic on Trees [32], “Separated Handle-Rewriting Hypergraph Grammars” (S-HH) [16], and others.

This last category hints at one of the major dichotomies in graph grammar theory: the distinction between *node-rewriting* and *edge-rewriting* grammars. While the former transform graphs by mapping nodes to graphs (and includes NLC and NCE), and so edges in the original graph are only manipulated as a consequence of node transformations, the latter rewrite (hyper)edges[7] to (hyper)graphs. (Hyper)edge rewriting grammars have been primarily investigated by Kreowski [45, 23], Lautemann [71, 70] and Courcelle [13, 15], where they have been successfully used to establish many decidability properties for graph languages. The extension to handle-rewriting hypergraph grammars is through the inclusion of the vertices to which the hyperedge is attached in the rewrite; such a structure is called a “handle”.

[7] A generalization of edges, hyperedges connect more than two nodes together.

There have been attempts to reconcile the two approaches. For instance, the schematic formalization of graph grammars developed by Kreowski and Rozenberg [65, 66] encompasses a large variety of grammars in both camps. They describe the actions of graph grammars in terms of five basic operations: *choose* a rule application, *check* its applicability, *remove* the designated parts of the graph, *add* parts to the graph, and finally *connect* graph elements. Unfortunately, such a high level of abstraction does not engender many useful results. More specific results, particularly for the context-free/confluent versions, have begun to appear in the last few years. Node and edge grammars are united, for instance in the paper (mentioned above) by Courcelle, Engelfriet and Rozenberg showing that S-HH grammars are expressively equivalent to C-edNCE [16, 17], as well as by Engelfriet and Heyker showing that “Context Free Hypergraph Grammars” (CFHG) have the same expressive power as C-edNCE when both are restricted to graphs of bounded degree [33].

Our grammars are designed for two goals; to allow for the easy expression of partitionings and associated problems, and to accurately model (doubly connected) data structures. The resulting formalism is distinct from any of these existing systems; like eNCE we permit more than one node on the left-side of a production, and have separate embedding relations for each rule, though our embedding relation more precisely resembles that used by Slisenko, in a work describing a polynomial-time solution to the Hamiltonian Circuit problem for certain graphs [96]. However, there are many important differences, such as the use of $\frac{1}{2}$-edges, and restrictions we have introduced to make the execution of our model practical: bounded-degree, no node can being allowed to have more than one $\frac{1}{2}$-edge attached with the same label, matching by bijection, ST-overlap free, etc. This makes comparisons somewhat difficult to perform, although the overall simularity makes it seem likely that our grammars have an expressive power somewhere between 1-eNCE and eNCE. Our model is also not context-free; our basic dangling graph grammars are permitted to have SS-overlap which can make the resulting language dependent on the order of rule application. When we introduce our parallel graph grammars we must of course ensure that no two rules overlap, but the use of (non-rewritten) *contexts* again makes the language order-dependent. 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 [73, 86].

9.3. *Using graph grammars*

Graph grammars have been used for a variety of purposes related to parallelism and data structure development. Graph grammars, for instance, have been used to analyse network reliability [85], solid modeling for CAD/CAM systems [39], compiler generation [47], and as a syntax for visual representations [89].

There have also been approaches to data structure manipulation based on graph grammars, though none have dealt with the partitioning problems arising from coarse-grained parallelism. One of the earliest was the IPSEN project of Nagl et al. [81]. The goal here was to give formal methods for software development, using graph grammars as a specification system. This project spawned the well-known “PROGRESS” (PROgrammed Graph Rewriting SyStems) language of Schürr, a graph rewriting formalism intended for generic software development [94, 95]. This is a complete system, including language and (textual) editing environment. As with the original IPSEN project, though, many of its constructs, such as the use of directed edges and matching rules through an (unrestrained) graph query sublanguage, make partitioning difficult, and so they are unsuitable for our purposes. The intermediate language “Lean” by Barendregt et al. [6] is another generic graph grammar language, with a similar drawback. There have also been many papers on implementing database queries and transformations using graph grammars [2, 3, 40], but again these contain constructs which make partitioning difficult.

Specifically for parallel applications, Janssens and Rozenberg [56] give a theoretical result where they model the behaviour of an Actor grammar using graph transformations. Barthelmann and Schied also use graph grammars as the semantics of a parallel language, “DHOP” [7], and Glauert et al. Sleep have developed “Dactl” as a graph grammar-based common target language for a number of other languages [42]. All of these approaches, while interesting and designed to deal with parallelism, take a relatively “fine-grained” approach to parallelism, essentially rendering the partitioning problem moot.

10. Conclusions

Our grammars cannot generate all graphs. A formalism which generates all graphs with their corresponding partitionings 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 structures commonly used in computer science applications. The grammar-based method we have defined also includes some capacity for use in incremental situations. Since the grammar rules used for graph construction can also be used for updates, the new partitioning can be related to the old in a manner corresponding to the actual update. Roughly speaking, this means a small change in the data structure will result in an equally small change in the partitioning. Such a natural correlation makes our method especially useful for dynamically-changing structures, and we are currently investigating the limits of this approach.

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 [4, 5, 13, 68, 69, 71, 90, 96]. We have taken the opposite approach, starting with a difficult problem and showing that a certain efficient solution implies an upper bound on tree-width related to the partitionability of the graph. Nevertheless, it is interesting to find our solution converging to the same class of graphs. 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.

Acknowledgements

This research was funded by grants from NSERC of Canada and FCAR of Quebec. All opinions expressed are those of the authors, and not necessarily of their respective institutions.

References

[1] N. Alon, P. Seymour, R. Thomas, Planar separators, SIAM J. Discrete Math. 7(2) (1994) 184–193.

[2] M. Andries, G. Engels, Syntax and semantics of hybrid database languages, in: H.J. Schneider, H. Ehrig (Eds.), Graph Transformations in Computer Science: Proc. Int. Workshop, Lecture Notes in Computer Science, vol. 776, Dagstuhl Castle, Germany, Springer, Berlin, January 1993, pp. 19–36.

[3] M. Andries, J. Paredaens, A language for generic graph-transformations, in: G. Schmidt, R. Berghammer (Eds.), Graph-Theoretic Concepts in Computer Science: Proc. 17th Int. Workshop, WG '91, Lecture Notes in Computer Science, vol. 570, Fischbachau, Germany, 17–19 June, Springer, Berlin, 1991, pp. 63–74.

[4] S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability – a survey, BIT 25(1) (1985) 2–23.

[5] S. Arnborg, J. Lagergren, D. Seese, Problems easy for tree-decomposable graphs, in: T. Lepistö, A. Salomaa (Eds.), Proc. 15th Int. Colloquium On Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 317, Tampere, Finland, 11–15 July, Springer, Berlin, pp. 38–51. Extended abstract.

[6] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, Towards and intermediate language based on graph rewriting, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (Eds.), Proc. PARLE – Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 258–259, vol. I, Eindhoven, The Netherlands, 15–19 June, Springer, Berlin, 1987, pp. 159–174.

[7] K. Barthelmann, G. Schied, Graph-grammar semantics of a higher-order programming language for distributed systems, in: H.J. Schneider, H. Ehrig (Eds.), Graph Transformations in Computer Science: Proc. Int. Workshop, Lecture Notes in Computer Science, vol. 776, Dagstuhl Castle, Germany, January, Springer, Berlin, 1993, pp. 71–85.

[8] P. Boehm, H. Ehrig, U. Hummert, M. Löwe, Towards distributed graph grammars, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Proc. 3rd Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Warrenton, Virginia, 2–6 December, Springer, Berlin, 1986, pp. 86–98.

[9] G.H. Botorog, H. Kuchen, Algorithmic skeletons for adaptive multigrid methods, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Int. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6, September, Springer, Berlin, 1995, pp. 27–41

[10] F.J. Brandenburg, The computational complexity of certain graph grammars, in: A.B. Cremers, H.P. Kriegel (Eds.), Theoretical Computer Science: 6th GI-Conf., Lecture Notes in Computer Science, vol. 145, Dortmund, West Germany, January, Springer, Berlin, 1983, pp. 91–99.

[11] F.J. Brandenburg, On partially ordered graph grammars, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Proc. 3rd Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Warrenton, Virginia, 2–6 December, Springer, Berlin, 1986, pp. 99–111.

[12] T.N. Bui, A. Peck, Partitioning planar graphs, SIAM J. Comput. 21(2) (1992) 203–215.

[13] B. Courcelle, Graph rewriting: an algebraic and logic approach, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, 1990, pp. 195–241.

[14] B. Courcelle, An axiomatic definition of context-free rewriting and its application to NLC graph grammars, Theoret. Comput. Sci. 55 (1987) 141–181.

[15] B. Courcelle, The logical expression of graph properties (abstract), in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 38–40.

[16] B. Courcelle, J. Engelfriet, G. Rozenberg, Context-free handle-rewriting hypergraph grammars, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 253–268.

[17] B. Courcelle, J. Engelfriet, G. Rozenberg, Handle-rewriting hypergraph grammars, J. Comput. System Sci. 46 (1993) 218–270.

[18] A. Das, L.E. Moser, P.M. Melliar-Smith, A parallel processing paradigm for irregular applications, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Int. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 249–254.

[19] R. Diekmann, R. Lüling, B. Monien, C. Spräner, A parallel local-search algorithm for the k-partitioning problem, Proc. 28th Hawaii Int. Conf. on System Sciences (HICSS '95), vol. 2, 1995, pp. 41–50.

[20] R. Diekmann, D. Meyer, B. Monien, Parallel decomposition of unstructured fem-meshes, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Int. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 199–215.

[21] K. Diks, H.N. Djidjev, O. Sýkora, I. Vrťo, Edge separators of planar and outerplanar graphs with applications, J. Algorithms 14 (1993) 258–279.

[22] W.E. Donat, A.J. Hoffman, Lower bounds for the partitioning of graphs, IBM J. Res. Dev. 17(5) (1973) 420–425.

[23] F. Drewes, H.J. Kreowski, A note on hyperedge replacement, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 1–11.

[24] H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990.

[25] H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Proc. 3rd Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Warrenton, Virginia, 2–6 December, Springer, Berlin 1986.

[26] H. Ehrig, Tutorial introduction to the algebraic approach of graph grammars, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rozenberg (Eds.), Proc. 3rd Internat. Workshop on Graph-Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Springer, Berlin, December 1987, pp. 3–14.

[27] H. Ehrig, P. Boehm, U. Hummert, M. Löwe, Distributed parallelism of graph transformations, in: H. Gottler, H.J. Schneider (Eds.), Proc. 13th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG '87), Lecture Notes in Computer Science, vol. 314, Springer, Berlin, July 1988, pp. 1–19.

[28] H. Ehrig, M. Korff, M. Löwe, Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 24–37.

[29] H. Ehrig, M. Magl, G. Rozenberg (Eds.), Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, Haus Ohrbeck, West Germany, 4–8 October, Springer, Berlin, 1982.

[30] J. Engelfriet, G. Leih, E. Welzl, Boundary graph grammars with dynamic edge relabeling, J. Comput. System Sci. 40 (1990) 307–345.

[31] J. Engelfriet, Context-free NCE graph grammars, in J. Csirik, J. Demetrovics, F. Gécseg (Eds.), Proce. Internat. Conference on Fundamentals of Computation Theory (FCT '89), Lecture Notes in Computer Science, vol. 532, Szeged, Hungary, August, Springer, Berlin, 1989, pp. 148–161.

[32] J. Engelfriet, A characterization of context-free NCE graph languages by monadic second-order logic on trees, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, 311–327.

[33] J. Engelfriet, L. Heyker, Hypergraph languages of bounded degree, J. Comput. System Sci. 48 (1994) 58–89.

[34] J. Engelfriet, G. Rozenberg, Graph grammars based on node rewriting: an introduction to NLC graph grammars, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 12–21.

[35] C. Farhat, A simple and efficient automatic FEM domain decomposer, Comput. Struct. 28(5) (1988) 579–602.

[36] T.A. Feo, M. Khellaf, A class of bounded approximation algorithms for graph partitioning, Networks 20 (1990) 181–195.

[37] A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Internat. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995.

[38] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, 19th IEEE Design Automation Conf., 1982, pp. 175–181.

[39] P. Fitzhorn, A linguistic formalism for engineering solid modeling, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Proc. 3rd Internat. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Warrenton, Virginia, 2–6 December, Springer, Berlin, 1986, pp. 202–215.

[40] A.L. Furtado, P.A.S. Veloso, Specification of data bases through rewriting rules, in: H. Ehrig, M. Magl, G. Rozenberg (Eds.), Proc. 2nd Internat. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, Haus Ohrbeck, West Germany, 4–8 October, Springer, Berlin, 1982, pp. 102–114.

[41] T. Gautier, J.L. Roch, G. Villard, Regular versus irregular problems and algorithms, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Internat. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 1–25.

[42] J.R.W. Glauert, J.R. Kennaway, M.R. Sleep, Dactl: an experimental graph rewriting language, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 378–395.

[43] P.W. Grant, M.F. Webster, X. Zhang, Solving computational fluid dynamics problems on unstructured grids with distributed parallel processing, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Internat. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 187–197.

[44] R. Gupta, SPMD execution of programs with dynamic data structures on distributed memory machines Proc. Internat. Conf. on Computer Languages, Oakland, California, 20–23 April, IEEE Computer Society Press, 1992, pp. 232–241.

[45] A. Habel, H.J. Kreowski, May we introduce to you: hyperedge replacement, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Proc. 3rd Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Warrenton, Virginia, 2–6 December, Springer, Berlin, 1986, pp. 15–26.

[46] B. Hendrickson, R. Leland, Multidimensional spectral load balancing, Technical Report SAND93-0074, Sandia National Laboratory, January 1993.

[47] B. Hoffmann, Modelling compiler generation by graph grammars, in: H. Ehrig, M. Magl, G. Rozenberg (Eds.), Proc. 2nd International Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, Haus Ohrbeck, West Germany, October 4–8, Springer, Berlin, 1982, 159–171.

[48] J. Hummel, L.J. Hendren, A. Nicolau, Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs, ACM Lett. Programm. Languages Systems, 1(3) (1992) 243–260.

[49] M. Jackel, ADA concurrency specified by graph grammars, in: G. Tinhofer, G. Schmidt, (Eds.), Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '86), Lecture Notes in Computer Science, vol. 246, Bernried, West Germany, 17–19 June, Springer, Berlin, 1986, pp. 41–57.

[50] D. Janssens, G. Rozenberg, On the structure of node-label-controlled graph languages, Inform. Sci. 20 (1980) 191–216.

[51] D. Janssens, G. Rozenberg, Restrictions, extensions and variations of NLC grammars, Inform. Sci. 20 (1980) 217–244.

[52] D. Janssens, G. Rozenberg, A characterization of context-free string languages by directed nodel-label controlled graph grammars, Acta Inform. 16 (1981) 63–85.

[53] D. Janssens, G. Rozenberg, Graph grammars with neighbourhood controlled embedding, Theoret. Comput. Sci. 21 (1982) 55–74.

[54] D. Janssens, G. Rozenberg, Graph grammars with node-label controlled rewriting and embedding, in: H. Ehrig, M. Magl, G. Rozenberg (Eds.), Proc. 2nd Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, Haus Ohrbeck, West Germany, 4–8 October, Springer, Berlin, 1982, pp. 186–203.

[55] D. Janssens, G. Rozenberg, Hypergraph systems generating graph languages, in: H. Ehrig, M. Nagl, G. Rozenberg (Eds.), Proc. 2nd Internat. Workshop on Graph-Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, October, Springer, Berlin, 1983, pp. 172–185.

[56] D. Janssens, G. Rozenberg, Structured transformations and computation graphs for actor grammars, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, 446–460.

[57] D. Janssens, G. Rozenberg, R. Verraedt, On sequential and parallel node-rewriting graph grammars, Comput. Graphics Image Process. 18 (1982) 279–304.

[58] D. Janssens, G. Rozenberg, R. Verraedt, On sequential and parallel node-rewriting graph grammars, II, Comput. Vision Graphics Image Process. 23 (1983) 295–312.

[59] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part 1: graph partitioning, Oper. Res. 37(6) (1989) 865–893.

[60] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, Technical Report 96-064, University of Minnesota, Department of Computer Science, Minneapolis, MN, 55455, August 1995.

[61] R. Kennaway, On "On graph rewritings", Theoret. Comput. Sci. 52 (1987) 37–58.

[62] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell System Tech. J. (1970) 291–307.

[63] N. Klarlund, M.I. Schwartzbach, Graph types, Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Charleston, South Carolina, 10–13 January, 1993, pp. 196–205.

[64] T. Kloks, Treewidth: Computations and Approximations, Lecture Notes in Computer Science, vol. 842, Springer, Berlin, 1994.

[65] H.-J.Kreowski, G. Rozenberg, On structured graph grammars I, Inform. Sci. 52 (1990) 185–210.

[66] H.-J. Kreowski, G. Rozenberg, On structured graph grammars II, Inform. Sci. 52 (1990) 221–246.

[67] S. Kundu, J. Misra, A linear tree partitioning algorithm, SIAM J. Comput. 6(1) (1997) 151–154.

[68] C. Lautemann, Decomposition trees: structured graph representation and efficient algorithms, in: M. Dauchet, N. Nivat (Eds.), Proc. 13th Colloq. on Trees in Algebra and Programming, Lecture Notes in Computer Science, vol. 299, Springer, Berlin, March 1988, pp. 28–39.

[69] C. Lautemann, Efficient algorithms on context-free graph languages, in: T. Lepistö, A. Salomaa (Eds.), Proc. 15th Internat. Colloquium On Automata, Languages and Programming, Lecture Notes in Computer Science, vo. 317, Tampere, Finland, 11–15 July, Springer, Berlin, 1998, 362–378.

[70] C. Lautemann, The complexity of graph languages generated by hyperedge replacement, Acta Inform. 27 (1990) 399–421.

[71] C. Lautemann, Tree automata, tree decomposition and hyperedge replacement, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 520–537.

[72] T. Lepistö, A. Salomaa (Eds.) Proc. 15th Internat. Colloq. on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 317, Tampere, Finland, 11–15 July, Springer, Berlin, 1988.

[73] A. Lindenmayer, Mathematical models for cellular interaction in development, J. Theoret. Biol. 18 (1968) 280–315.

[74] N. Linial, Locality in distributed graph algorithms, SIAM J. Comput. 21(1) (1992) 193–201.

[75] R.J. Lipton, R.E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36(2) (1979) 177–189.

[76] I. Litovsky, Y. Métivier, W. Zielonka, The power and the limitations of local computations on graphs, in: E.W. Mayr (Eds.), Proc. 18th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG '92), Lecture Notes in Computer Science, vol. 657, Wiesbaden-Naurod, Germany, 18–20 June, Springer, Berlin, 1992, pp. 333–345.

[77] M. Löwe, H. Ehrig, Algebraic approach to graph transformation based on single pushout derivations, in: R.H. Mohring (Eds.), Proc. 16th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG '90), Lecture Notes in Computer Science, vol. 484, June 1991, Springer, Berlin, pp. 338–353.

[78] J.A. Lukes, Efficient algorithm for the partitioning of trees, IBM J. Res. Dev. 18(3) (1974) 217–224.

[79] S. Miguet, J.-M. Pierson, Load balancing strategies for a parallel system of particles, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Int. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 255–260.

[80] U.G. Montanari, Separable graphs, planar graphs and web grammars, Inform. Control 16 (1970) 243–267.

[81] M. Nagl, G. Engels, R. Gall, W. Schäfer, Software specification by graph grammars, in: H. Ehrig, Manfred Magl, G. Rozenberg (Eds.), Proc. 2nd Internat, Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 153, Haus Ohrbeck, West Germany, 4–8 October, Springer, Berlin, 1982, pp. 267–287.

[82] M. Nagl, On the relation between graph grammars and graph l-systems, in: M. Karpinski (Ed.), Fundamentals of Computation Theory: Proc. Internat. FCT-Conf., Lecture Notes in Computer Science, vol. 56, Poznan-Kornik, Poland, September, Springer, Berlin, 1977, pp. 142–151.

[83] T. Nakanishi, K. Joe, H. Saito, C.D. Polychronopoulos, A. Fukuda, K. Araki, The data partitioning graph: extending data and control dependencies for data partitioning, in: K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Proc. 7th Internat. Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 892, Ithaca, New York, 8–10 August, 1994, Springer, Berlin, pp. 170–185.

[84] ESPRIT Basic Research Working Group No.3299, Computing by graph transformation: overall aims and new results, in H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 688–703.

[85] Y. Okada, M. Hayashi, Graph rewriting systems and their application to network reliability analysis, in: G. Schmidt, R. Berghammer (Eds.), Proc. 17th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG '91), Lecture Notes in Computer Science, vol. 570, Fischbachau, Germany, 17–19 June, Springer, Berlin, 1991, pp. 36–47.

[86] P. Prusinkiewicz, J. Hanan, L-systems: from formalism to programming languages, in: G. Rozenberg, A. Salomaa (Eds.), Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology, Springer, Berlin, 1992, pp. 193–211

[87] J.-C. Raoult, On graph rewritings, Theoret. Comput. Sci. 32 (1984) 1–24.

[88] R. Reischuk, Graph theoretical methods for the design of parallel algorithms, in: L. Budach, (Ed.), Proc. 8th Internat. Conference on Fundamentals of Computation Theory (FCT '91), Lecture Notes in Computer Science, vol. 529, Gosen, Germany, September, Springer, Berlin, 1991, 61–67.

[89] J. Rekers, On the use of graph grammars for defining the syntax of graphical languages, Technical Report 94-11, Department of Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands, 1994. Available by ftp: `ftp.wi.leidenuniv.nl` as `pub/cs-techreports/tr94-11.ps.gz`.

[90] N. Robertson, P.D. Seymour, Graph minors II: algorithmic aspects of treewidth, J. Algorithms 7 (1986) 309–322.

[91] P. Sanders, Better algorithms for parallel backtracking, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Int. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 333–347.

[92] H. Saran, V.V. Vazirani, Finding k cuts within twice the optimal, SIAM J. Comput. 24(1) (1995) 101–108.

[93] H.J. Schneider, H. Ehrig (Eds.), Graph Transformations in Computer Science: Proc. Internat. Workshop, Lecture Notes in Computer Science, vol. 776, Dagstuhl Castle, Germany, January 1993, Springer, Berlin.

[94] A. Schürr, Introduction to PROGRESS, an attribute graph grammar based specification language, in: M. Nagl, (Ed.), Proc. 15th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG '89), Lecture Notes in Computer Science, vol. 441, Castle Rolduc, The Netherlands, June, Springer, Berlin, 1989, 151–165.

[95] A. Schürr, PROGRESS: A VHL-language based on graph grammars, in: H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Proc. 4th Internat. Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 532, Bremen, Germany, 5–9 March, Springer, Berlin, 1990, pp. 641–659.

[96] A.O. Slisenko, Context-free grammars as a tool for describing polynomial-time subclasses of hard problems, Inform. Process. Lett. 14(2) (1982) 52–56.

[97] O. Sýkora, I. Vrťo, Edge separators for graphs of bounded genus with applications, Theoret. Comput. Sci. 112(2) (1993) 419–429.

[98] C. Walshaw, M. Cross, M.G. Everett, S. Johnson, K. McManus, Partitioning & mapping of unstructured meshes to parallel machine topologies, in: A. Ferreira, J. Rolim (Eds.), Parallel Algorithms for Irregularly Structured Problems: Proc. 2nd Internat. Workshop, IRREGULAR '95, Lecture Notes in Computer Science, vol. 980, Lyon, France, 4–6 September, Springer, Berlin, 1995, pp. 121–126.