A PARALLEL SOLUTION STRATEGY FOR IRREGULAR, DYNAMIC PROBLEMS

by

Clark Verbrugge

School of Computer Science
McGill University, Montréal
Québec, Canada

August 1996

A THESIS SUBMITTED TO THE SCHOOL OF COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 1996 by Clark Verbrugge

Abstract

Parallelizing irregular, dynamic data structures can be a very difficult problem. An efficient solution often demands that work on the data structure be divided up among processors, yet despite the large and growing number of such applications there has been little work done on general approaches to such situations. In part this is due to the extreme difficulty of ensuring enough generality to be useful, while still efficiently addressing each individual problem; irregular and dynamic problems can vary dramatically, and direct, efficient solutions simply do not exist. Most attempts have therefore either concentrated on problem-specific areas, where good results can be obtained at the expense of generality, or have defaulted to heuristic methods applicable to almost any possible situation.

In this thesis we present a hybrid method, combining a direct and problem-specific approach with a system general enough to be applicable to the majority of applications. Our method is based on local graph transformations; if the data structure does not change dramatically, then the parallelization should not need to either. By reflecting data structure manipulations in a particular *graph grammar* formalism, we can show deterministic bounds on the quality of the resulting division of work. Thus, since our formalism is reasonably general and accommodates many of the more common data structures (and also allows for considerable variation), we have a general strategy for dealing with problems requiring dynamic, pointer-based structures.

The utility of our technique is then established by a non-trivial application of it to a realistic problem: grid generation for irregular domains in the Control Volume Finite Element Method used in computational fluid dynamics. We first illustrate an algorithm solving the problem, and then show the use of our approach. Theoretical bounds on the partitioning issues pertinent to parallelization are then given, and supported by experimental measurements. We also compare our method to existing heuristic methods to ensure our results are competitive. Our method turns out to produce good quality results, with a number of distinct advantages over other techniques, particularly when applied to dynamic situations.

Résumé

La parallélisation de structures de données dynamiques et irrégulières peut être un problème plutôt difficile. Une solution efficace demande souvent que le travail sur la structure soit distribué parmi les différents processeurs. En dépit du nombre croissant de telles applications, peu de recherche a été menée sur les approches générales à utiliser dans de tels cas. C'est dû en partie à l'extrême difficulté d'assurer assez de généralité en tenant conte de l'utilité et en s'assurant que chaque problème est traité d'une façon efficace. Les problèmes irréguliers et dynamiques varient énormément, et les solutions directes et efficaces n'existent tout simplement pas. La plupart des tentatives par conséquent, se sont concentrées sur les champs spécifiques à un problème particulier où de bons résultats peuvent être obtenus en sacrifiant la généralité, ou bien sont retombées sur les méthodes heuristiques qui s'appliquent à plus ou moins n'importe quelle situation.

Cette thèse présente une méthode hybride qui combine une approche axée sur le problème et un système assez général pour être applicable dans la majorité des cas. Notre méthode se base sur les transformations locales de graphes; si la structure de données ne change pas de façon majeure, la parallélisation ne devrait pas non plus. En représentant les manipulations de la structure de données par un formalisme de grammaire de graphe particulier, nous pouvons montrer des bornes déterministes sur la qualité de la division du travail qui en résulte. Cependant, parce que notre formalisme est raisonnablement général et peut accommoder plusieurs des structures de données communes (et permet une variation considérable), nous avons une stratégie générale pour attaquer les problèmes nécessitant des structures dynamiques et basées sur des pointeurs (pointer-based).

L'utilité de notre technique est établie par son application non-triviale à un problème vraisemblable: la génération de grilles pour les domaines irréguliers dans la méthode "Control Volume Finite Element" qui est utilisée dans la dynamique de fluides de calcul (computational fluid dynamics). D'abord nous démontrons un algorithme qui trouve la solution au problème, et ensuite nous démontrons notre approche. Les bornes théoriques pour les issues concernant la division pertinente à la parallélisation sont ensuite données, et appuyées par les mesures expérimentales. Nous comparons notre méthode aux méthodes heuristiques qui existent déjà pour assurer que nos résultats soient compétitifs. Notre méthode nous donne des résultats de bonne qualité avec un nombre d'avantages à comparer avec autres techniques, particulièrement quand elle s'applique aux situations dynamiques.

Acknowledgements

I would like to thank my advisor, Prakash Panangaden, for his support (financial and otherwise), encouragement and invaluable advice; I have learned a great deal from him, and would certainly have never finished this without his guidance. I would also like to thank Rabi Baliga and Nabil Elkouh for tutoring me on fluid dynamics, and all their help. I also thank Laurie Hendren, who has helped me at many of the more difficult times. I am eternally grateful for the financial support provided by *Natural Sciences and Engineering Research Council* (NSERC), *Fonds pour la formation de chercheurs et l'aide à la recherche de Québec* (FCAR), *Centre de recherche informatique de Montréal* (CRIM), *Centre de Recherche en Calcul Appliqué* (CERCA), and Denis Thérien.

Finally, I thank my parents, without whom absolutely nothing would ever have existed, existentially speaking; I thank Michel and Miller for their tireless efforts, and Liz Breier for her timely help. My wife, Carrol Dufault, deserves the highest praise for her endless support, putting up with my incoherent ramblings, and for keeping me from going mad these last few years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Organizing data into *structures* in order to optimize data access patterns has become one of the dominant paradigms of computer algorithm development. Most modern computer languages include some method for constructing and dynamically altering data aggregates: in C or Pascal this is provided through recursive `struct` or `record` specifications and pointers, in Lisp by `cons` cells, and in object-oriented languages almost by definition. Despite the success of this approach, however, there have been remarkably few attempts to incorporate dynamic data structures into parallel applications with any amount of generality. In fact, while there is a large body of work on generic approaches to algorithms using arrays and small variations—so-called "regular" data structures—almost every attempt at parallelizing an algorithm involving dynamic data structures has required a painstakingly hand-crafted solution to achieve reasonable efficiency. Of course such a problem-specific approach to algorithm development is far too labour-intensive to be useful; we need efficient, general methods for working with dynamic data structures in a parallel environment.

1.1 Goals

Unfortunately, it is evident that practical and truly general approaches to dynamic, or irregular, data structures do not exist; algorithms capable of dealing with the myriad of diverse possibilities would simply be much too computationally expensive to be effective. It is our contention, however, that general methods *do* exist for the efficient expression of large classes of useful algorithms employing irregular data structures: we do not need to be able to deal with all irregular data structures, just enough to encompass the bulk of "useful" ones. This is the focus of this thesis: by concentrating on data structures which can be efficiently parallelized and exploring their commonality, we are able to show a general method for using a wide variety of irregular data structures in a parallel setting, including most of the structures regularly employed by computer scientists.

1.2 Requirements

A parallel implementation of an algorithm reduces execution time by having several processes work on independent subproblems. Simply finding such a division is not difficult in most cases, since few algorithms necessarily have a totally linear pattern of computation; there are almost always enough subtasks for parallel execution. Unfortunately, this problem is not really so straightforward; even fewer algorithms decompose into *completely* independent subtasks, and *communication cost,* the extra time required for distinct processors to exchange information, is a factor in the parallelization of any problem. Indeed, the time for one processor to send data to another is often several orders of magnitude more expensive than a local machine instruction, and so communication cost can be a critical factor.

Of course the *speed-up,* the decrease in time exhibited by a parallel program compared with an equivalent sequential program, will also be affected by the *distribution* of tasks. If one processor is given more work than others, then not all processors will complete their work at the same time, and some processors will be idle waiting for slower processors to catch up, effectively reducing the intended parallelism. Thus, an efficent parallel implementation should minimize communication cost while ensuring a *balanced workload* for all processors.

One approach to the problem of finding a balanced, low-communication division of work is to divide up the associated data structure, and assume that work is divided as a consequence. Communication cost is then reflected in the number of connections/communications in our structure between data elements belonging to different partitions, and load-balancing is given by the number of data elements in each partition. This reduces the problem to one of *graph partitioning,* and is the tactic employed by parallel languages such as Fortran-90 and C*, though primarily toward regular data structures. In these languages syntactic "array distribution directives" are used to specify which processor "owns" array elements, and hence which processor executes a given operation on its data. Similarly, the popular Single Processor Multiple Data (SPMD) paradigm [CK88] expresses parallelism by creating (and locating) multiple instances of a datum: operations are performed in parallel only by having each processor work on a distinct datum. The programmer in these cases is responsible for designating ownership to ensure a minimal communication strategy, and usually for load-balancing as well (though heuristic automatic load-balancers are sometimes included). A general method, however, should be able to *automatically* partition the graph in accordance with both load-balancing and communication cost constraints, and without extra human intervention.

We need a simplified model of our problem. Finding a partitioning of a graph that minimizes the number of "cut edges" (edges between rather than within partitions) is provably hard (NP-complete; see problem ND14 in [GJ79]), so we cannot reasonably hope to compute an efficient, fully-general answer in practice. Fortunately, the bulk of dynamic data structures used in computer science do not span the gamut of possible graphs to partition: we do not need to be able to partition *all* graphs to have a practical solution. It will be sufficient if we can show a method applicable to some large class of graphs encompassing many of the more common and useful graphs. Our method should include trees for certain, various threaded-trees, grids, lists and most combinations, and as many others as possible. In particular our method should be robust in the face of "minor" variations in these structures—many algorithms rely on small differences in a structure from its general form, and this should not force us to abandon our approach.

1.3 Simplified Model

Our problem model will therefore consist of an irregular graph, which must be partitioned with balancing and communication cost constraints. We will make two further reductions: *bounded-degree* and *doubly-connected.* Neither of these conditions is strictly necessary in order to find an acceptable partitioning, but the combination reduces the number of situations which are difficult to partition, and generally make the computation of partitionings and costs simpler. Even with these extra constraints we can still include almost all of the data structures in which we are interested, so these reductions do not dramatically reduce the utility of our method.

The degree of a node in a graph is the number of edges attached to it. Graphs containing nodes with any number of attached edges, *i.e.,* unbounded degree, can easily become unmanageable from a partitioning perspective—if all nodes in a tree point to the root, then whichever partition contains the root will have an inherently high communication cost. If our general method makes any worst-case approximations, then structures such as these will seriously dilute the efficacy of our approach. Thus, we avoid these situations by demanding that each node have a fixed maximum number of attached edges (bounded-degree).

Edges in a graph may be directed (one-way), or undirected (two-way). Verifying the bounded-degree property in the presence of directed graph edges would require non-trivial bookkeeping—each time we add an edge somewhere, we have to ensure that we have not violated the bounded-degree property for both the source and the target of the edge. This could be solved by keeping a count of the number of incoming edges for each node; however, in order to efficiently provide the complete partitioning we also need to know which edges are directed at a particular node. The use of undirected graphs makes this whole process considerably easier, and for these reasons we have focussed on undirected graphs, corresponding to doubly-connected data structures.

We are therefore representing the data by a graph, with edges between nodes for communication paths; by partitioning this graph we partition the data, and hence the work. Of course some communication paths will be used more often then others, and the usual technique for expressing this is to add "weights" to edges representing relative communication costs, and then solve the edge-weighted graph partitioning problem. For simplicity, though, we consider only the "unweighted" problem, where all edges and nodes are considered equivalent with respect to partitioning. For large graphs with computation reasonably evenly-distributed throughout the data structure this is an adequate model, and given the generality of data structures and algorithms we hope to include, an acceptable approximation. It may also be possible to extend our methods to weighted graphs, though we do not explore this here.

1.4 Methodology

Our method will be based on enforcing the *locality* of operations. Intuitively, if we have a graph which we already know we can partition efficiently, and we alter it in some small way we should be able to discover the new partition with a correspondingly small effort. This is the basis for our technique: if the changes to a graph are both small and local—*i.e.,* changes occur within only a bounded region of the graph—and we know the partitioning of the graph before the change, then we can incrementally compute the partitioning for the changed graph. Our method will therefore rely on a formalism for maintaining a graph while expressing updates in a local manner, allowing for the quick and efficient computation of partitionings.

We must of course also show that this allows us to produce the kinds of graphs we would like to include in a general method. There are no precise characterizations of "useful" data structures, and the large number of subtle variations make it a difficult concept to capture, so our primary means for showing expressiveness will be by example. We will show how to express a number of common data structures used in applications, and give corresponding bounds on partitioning. Such examples will provide evidence of the broad applicability of our technique, but not how one might use it in a real application. For this reason we also include an extended example of a non-trivial problem (irregular grid generation for the finite element method, as used in computational fluid dynamics) and show the application of our technique.

Our method would seem to suffer from the same drawback as many other data partitioning techniques: including it into an algorithm requires a significant amount of additional programming, increasing the difficulty and complexity of program design. For this reason we have also developed a new, explicitly parallel language, called "**eL**." This language merges our partitioning model with a model of computation, producing a single linguistic formalism which automatically and transparently incorporates partitioning requirements. Any programs written in our language must be partitionable with specific and easily-calculable upper bounds on balancing and communication cost.

1.5 Outline of Thesis

We begin with Chapter 2, which outlines the major contributions of this work. We describe our original results and methods and give reasons for the significance of our results. We also point out the areas in which we have suggested further potentially-useful areas of investigation.

Our main result is described in Chapter 3. Here we develop our primary methodology for partitioning data structures where the operations on the data structure all have only local effects. This method relies on the dual generation and manipulation of the data structure using a particular kind of *graph grammar*. If a data structure and its operations can be expressed in this formalism, then a simple method exists which will guarantee bounds on the communication cost of the resultant data structure.

This approach is further explored in Chapter 4, where a novel parallel computing language based on the ideas of the previous chapter is developed. By limiting the computations which can be expressed to ones which can be retained in a data structure representing the locality of interaction, the language essentially forces all algorithms to be efficiently parallelizable (provided some straightforward conditions are met). Non-trivial examples of algorithms written in this language are discussed, and upper bounds on the cost of both sequential and parallel implementations of the language are examined.

In Chapter 5, we illustrate a complex example of the application of our ideas on partitioning. Here we develop an algorithm for generating adaptive grids for the Control Volume Finite Element Method (CVFEM) on unstructured domains. This algorithm produces the domain decompositions required by the method, and implements a solution algorithm as well. Moreover, it maximizes the local nature of adaptations and computations on the grid, and hence is eminently suitable for parallelization by our method. We give a detailed explication of the structure of the algorithm, followed by results from an actual (sequential) implementation of the algorithm. Upper-bounds on related partitioning problems (for parallelism) are discussed in the context of an implementation of the algorithm using our graph grammar formalism.

The relation of our work to existing results is described in Chapter 6. Here we present the various facets of our method: graph partitioning, graph grammars and associated applications, and CVFEM, and situate our results in relation to the myriad of other approaches to the same problems. Finally, in Chapter 7 we draw some conclusions about our method. We find that the formalism we have developed is adequate for many non-trivial problems; our extended example demonstrates both the feasibility and applicability of our method.

Chapter 2

Outline of Contributions

The primary contribution of our work has been to develop a general model of graphs which we are certain can be efficiently partitioned, and yet is general enough to be applicable to a wide variety of algorithms. The key idea is to permit all graphs that can be generated by a certain class of rewrite systems called *graph grammars.* One can then use the intermediate structures that arise in the generation—the so-called parse-tree—to guide the partitioning. The grammars are sufficiently general as to allow many useful families of graphs to be expressed. Below we describe the relevant aspects of our approach and how each forms a distinct contribution.

2.1 Partitionability Using Grammars

In the beginning of Chapter 3 we develop the notion of "partitionable" graphs to describe the demands of parallelism on dynamic data structures. This concept allows for a quantitative attack on the problem of partitioning dynamic data structures. Our model is more realistic than existing quantitative approaches—we deal with an unknown number of processors, and require load-balancing to within a constant *additive* factor. The "traditional" approach is to balance within a *multiplicative* factor, and provide communication costs for just k partitions, for a fixed k. By providing a well-defined model with realistic criteria, we have created one of the very few approaches to partitioning dynamic problems for parallelism with any real hope of producing useful solutions.

Our definition of dangling graph grammars is unique, and has a number of practical advantages over existing graph grammar models. Most results on graph grammars have been theoretical, due to the various NP-complete problems (subgraph isomorphism in particular) associated with their definition. By combining strict labelling requirements on our graphs with a specific and realizable graph matching algorithm, we are able to define a feasible execution model for our grammars—something neglected in most other definitions of graph grammars. This is what permits us to actually use our grammars in our partitioning algorithm.

In order to be able to derive our partitionings from the actions of graph grammars, we impose a restriction on *overlap* between parts of graph grammar rules. Overlap properties to avoid critical pairs during parallel rule application are not new in rewrite systems; our form of overlap, however, is designed to impose a specific structure on the graph grammar derivation sequence, in order to facilitate partitioning. Moreover, our overlap property is distinct from the overlap typically used to ensure the resulting grammars are *context-free.* Since the majority of significant results on graph grammars are based on the use of context-free grammars, our result is one of the few results on graph grammars that applies to context-sensitive, as well as context-free grammars.

Our partitioning algorithm relies on being able to partition trees. The cost bounds we derive for this purpose are not surprising, and both upper and lower bounds on this problem are well-known. Nevertheless, our solution has the advantage of being simple and constructive, by which mean that we give a precise algorithm for producing the partitions rather than just a proof of existence.

Finally, we also prove upper bounds on the *tree-width* of the graphs we generate. Computing tree-width is a very difficult problem, and no general techniques exist. By showing bounds on the large class of graphs we produce, we have demonstrated a feasible method for calculating upper-bounds on tree-width for certain kinds of graphs.

2.2 Parallel Languages

The language we develop in Chapter 4 is the only explicitly parallel language to include a non-trivial guarantee related to parallel performance. It is also one of the few parallel languages to explicitly deal with dynamic data structures—other languages, such as many of the parallel C hybrids, require one to use communication and synchronization primitives in order to build an efficient program in such situations. Alternatively, as with Linda [CGL86, CG90] or the Chemical Abstract Machine (CHAM) [BB90], arbitrary data accesses may be supported but with no attempt to retain structure. Our language allows for relatively efficient access based on the structure itself while retaining a simple conceptual model.

In creating a working prototype for the language we found it more convenient to use a graphical editing environment, where the data structure is represented and manipulated visually. This graphical user interface emphasizes the qualities of **eL** as a *visual language,* in the tradition of *LabView,* and various dataflow models. By graphically incorporating both computation and control flow in a manner quite distinct from such models, **eL** can be seen as a contribution to this field as well.

2.3 Grid Generation

Our example of CVFEM grid generation described in Chapter 5 is itself a novel algorithm. Although the problem of producing non-obtuse triangular grids has been solved in an optimal sense [BMR94], dealing with the extra demands of adaptivity and practical implementation with an eye to parallelism has not been addressed. Our algorithm produces non-obtuse grids and allows for incremental updates, all while ensuring calculations and modifications are as local as possible. This makes our algorithm an ideal candidate for parallelization, and certainly one of the very few that attempts to deal with adaptive irregular meshes in a practical manner. By using our algorithm as an example application for our partitioning method, we have demonstrated a method for efficiently dealing with irregular grids in CVFEM, as well as a new algorithm for adaptive grid generation itself.

Chapter 3

Partitionable Data Structures

3.1 Introduction

In this chapter we propose a linguistic mechanism, based on graph grammars, that generates families of graphs for which a "good" partitioning *must* exist. Moreover, this method is constructive, and the resultant partitionings are quite simple to produce once the graph family has been specified. The efficacy of this scheme is illustrated by first giving a precise definition of "good" partitionability, 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.

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* [RS86]; thus they are all "tree-like" to some degree, and in a manner correlated with the nature of the grammar.

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

In the following section we formalize the notion of "partitionability." Section 3.3

develops the basis for the grammars in which we are interested. In Section 3.4 we prove that weighted k-ary trees can be partitioned with guaranteed bounds on load-balancing. This result is used in Section 3.5, 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 3.6 extends this result to a larger class of graphs, showing how we can generate denser graphs with a corresponding sacrifice in partitionability. Section 3.7 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 3.8, where we establish limits on the "tree-width" of all graphs we generate.

3.2 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 3.2.1 *A* p-**partitioning** *of a graph $D = (V, E)$ is an equivalence relation $\cong$ on the vertices of D such that there are exactly p equivalence classes.*

A p-partitioning induces a communication cost from a partition to the rest of the graph (and of course to any other partition), 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 i^{th} piece.

Definition 3.2.2 *The* communication cost *of V_i is defined as:*

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

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

Definition 3.2.3 *Given some function f of n, an f-***partitionable** *graph of n vertices is a graph that can be partitioned into p pieces of size $(n/p) \pm c$ for any $1 \leq p \leq n$ 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 Diks *et al.* [DDSV93]). 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 3.2.4 *Let G be an f-partitionable graph. Then G is* **reasonably partitionable** *if $f \in O(\log(n))$.*

REMARKS: Almost all data structures fall into this category, other than direct representations of densely-connected data (such as grids or triangulations). Obviously, this also excludes any graph with a node having degree in $\omega(\log(n))$; for example, a tree with each leaf connected to the root is not $k \log(n)$-partitionable for any constant k, since some partition piece must include the root (of degree n). An example of a reasonably-partitionable graph is in Figure 3.1; 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

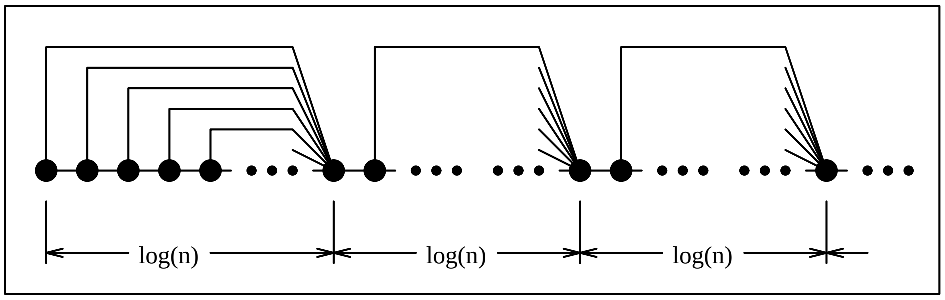


Figure 3.1: A reasonably partitionable graph.

connected to every node 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.

3.3 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.

3.3.1 Dangling Graphs

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

Definition 3.3.1 *A* **dangling graph***, D is an 8-tuple* $(V, E, \nu, \phi, \psi, \Sigma_V, \Sigma_E, C)$*, where:*

> V *is a set of* vertices *(or* nodes*),*
>
> E *is a set of 1/2-edges,*
>
> $\nu : E \to V$ *is an injective function returning the vertex associated with a given 1/2-edge.*
>
> Σ_V *is a finite set of node labels,*
>
> Σ_E *is a finite set of 1/2-edge labels,*
>
> $\phi : V \to \Sigma_V$ *is a node labelling function,*
>
> $\psi : E \to \Sigma_E$ *is a 1/2-edge labelling function, with the property that no two 1/2-edges connected to the same vertex have the same label:*
>
> $$\forall\ e, e' \in E,\ \nu(e) = \nu(e') \Rightarrow \psi(e) \neq \psi(e')$$
>
> $C \subseteq E \times E$ *is a connection relation between 1/2-edges, such that:*
>
> $$\forall\ (e, e') \in C,\ \neg\exists\ (e, e'') \in C \textit{ for } e'' \neq e',\ \textit{and}\ (e', e) \in C$$
>
> *In other words, C describes the connected pairs of 1/2-edges, and is symmetric.*

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

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

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

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

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

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

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

Definition 3.3.2 *A* trimmed *dangling graph is a dangling graph with the dangling edges removed: if D is a dangling graph then the trimmed version is given by the function ξ, where $\xi(D) = D[E_D - \Xi(D)/E_D]$.*

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

3.3.2 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 3.3.3 *A* **production** *is a pair of dangling graphs, a (connected)* **source** *and a* **target,** *along with a partial mapping between dangling edges. If S and T are dangling graphs, then (S, T, δ) is a production if both $\delta : \Xi(S) \rightarrow \Xi(T)$ and $\delta^{-1} : \Xi(T) \rightarrow \Xi(S)$ are partial functions.*

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

Definition 3.3.4 *A dangling graph S is a* subgraph *of another dangling graph D if:*

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

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

Definition 3.3.5 *An* induced subgraph *of a dangling graph D is a subgraph D of D', such that:*

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

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

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

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

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

Definition 3.3.7 *If D_1, D_2 are two dangling graphs, then $D_1 \sqsubseteq D_2$ iff there exist two label-preserving injections, $\alpha : V_1 \longrightarrow V_2$ and $\beta : E_1 \longrightarrow E_2$, such that:*

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

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 3.3.8 *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 3.3.9 *The* application *of a production $\rho = (S, T, \delta)$ to a dangling graph G involves locating an occurrence, S' of S within G, and replacing S' with (a copy[1] of) T. The function δ describes how to modify the connection relation so T is embedded in $G - S'$, utilizing only the connections in $CSet(S', G - S')$. A production ρ then* derives *a dangling graph H from a dangling graph G if ρ can be applied to G, and H is the result once dangling edges are suitably replaced.*

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

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

The connection relation is somewhat more complicated; if

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

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

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

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

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

Remarks: An application involves locating an occurrence matching the source of the production, removing the occurrence, and attaching a distinct copy of the target by reassigning connection relations involving dangling edges of the occurrence to dangling edges of the (copy of the) target. There are restrictions on the occurrence—the pattern matching of the source graph must result in a label and structure-preserving bijection h between the nodes and 1/2-edges in the source graph and the nodes and 1/2-edges in its *occurrence* in the graph. As well, if a node in the graph is included in the occurrence, then there must be corresponding matches in the source for *every* 1/2-edge attached to that node.

An example of a production being applied is shown in Figure 3.2; the input graph (axiom) is on the top left, the output is on the top right, and the production is shown on the bottom. Dotted arrows indicate the δ mapping for the production, and the the region enclosed on the input graph is the occurrence being rewritten. Node labels are illustrated by colour (shade), but 1/2-edge labels are not shown. Note that the other two white nodes (marked with x's) cannot be rewritten by this production; even if all labels matched, they do not form an exact image of the source graph (both x-marked nodes have degree 4, whereas the source requires one node with degree 4 and one with degree 3).

Once the occurrence is located, the nodes and 1/2-edges of the occurrence are removed and a distinct copy of the target graph is inserted. If within the graph a dangling edge e of the occurrence is paired with some other dangling edge e' to form a connection $c = e \times e'$, then the 1/2-edge designated by δ of the corresponding 1/2-edge of the source graph, $\delta(h(e))$, is substituted into c in place of e. If $\delta(h(e))$ is undefined for e, the connection relation c is discarded. In the example in Figure 3.2, three connection relations are transferred from the source to the target graph (indicated by dotted arrows), and any connections involving the other three 1/2-edges are deleted by the rewrite.

Within a single derivation, because of the restrictions on how the target is 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

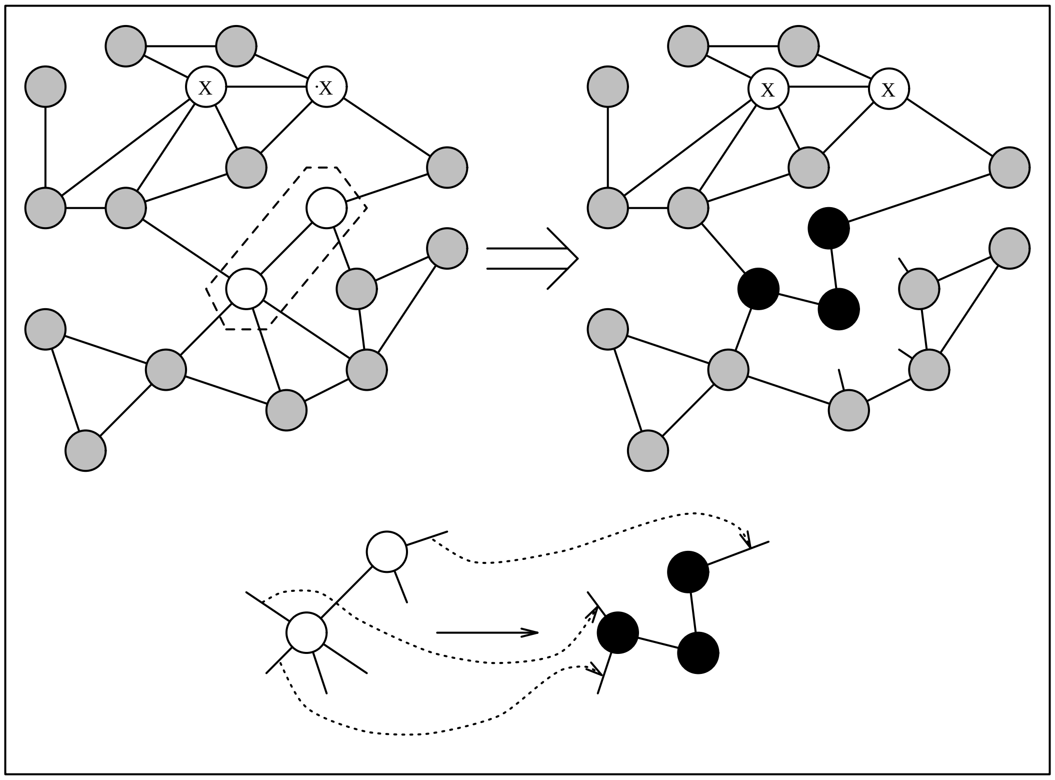


Figure 3.2: A production (bottom) is applied to a graph.

occurrence to the rest of the graph. This property will prove critical to partitionability:

Proposition 3.3.1 *Given a production $\rho = (S, T, \delta)$ and a graph G to which ρ applies, $G \rightarrow_{\rho} H$, with S' the image (occurrence) of S in G as above, the number of connection relations linking the embedded target to the rest of H is no more than the number of connection relations linking S' to G.*

3.3.3 Grammars

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

Definition 3.3.10 *The* language *generated by a graph grammar $G = (A, \Upsilon)$ is the set of all dangling graphs which can be derived from A using productions in Υ:*

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

3.3.4 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 3.3.11 *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) such W form the actual overlap of S and S'.*

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

Fortunately, it easy to restrict 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 1/2-edge is determined by only one production, and there can be no conflict in specification. This is precisely the no overlap property between all production source graphs:

Definition 3.3.12 *If $G = (A, \Upsilon)$ is a dangling graph grammar, and for all $\rho_1, \rho_2 \in \Upsilon$, $\rho_1 = (S_1, T_1, \delta_1)$ and $\rho_2 = (S_2, T_2, \delta_2)$ it is the case that $Overlap(S_1, S_2) = \emptyset$ or $\rho_1 = \rho_2$ and $Overlap(S_1, S_2)$ is just the trivial overlap, then G is* **SS-overlap free.**

Proposition 3.3.2 *If (A, Υ) is SS-overlap free, then the grammar is deterministic even if some productions are applied simultaneously.*

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

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

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

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

REMARKS: 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).

3.3.5 Contexts

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

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

Definition 3.3.14 *A production* **with context** *is one $\rho = (S, T, \delta)$ with a context I as defined above, with the property that each occurrence of S must be included in an occurrence of I. If a grammar G includes a production with context then the grammar is with context.*

3.4 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 3.4.1 *Given a natural number n, and a set N of any other m natural numbers which sum to n, it must be that if n_i is the i^{th} largest number in N then $n_i \leq n/i$.*

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

We will use Lemma 3.4.1 to prove a cost bound on a certain kind of partitioning of trees. First, we define some essential terminology.

Definition 3.4.1 *Let $T = (N, E)$ be a tree with nodes N and edges $E \subset N \times N$. Then Subtree(n) for $n \in N$ is the set of all nodes in N which are in the subtree rooted at n, including n itself, and Fanout(n) is the number of children of a given node n.*

Definition 3.4.2 *A* postorder *tree traversal is a total ordering of the vertices of a tree such that if v_i represents the i^{th} vertex in the ordering, then v_j is a vertex in the subtree rooted at v_i only if $j < i$.*

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

Lemma 3.4.2 *Let T be a tree of n nodes with maximum fanout b, with a positive integer weight w_i assigned to each node v_i, such that $\sum_i w_i = W \geq 1$. Let W_i be the total weight of all nodes in the subtree rooted at v_i; we also require that W_i is at least 1 for all subtrees. If a postorder search is performed where the child nodes are examining in decreasing order of total subtree weight, ordering the vertices as $v_1, \ldots, v_n$, then for any partition of T into two parts, $\cong_i = \{\{v_1, \ldots, v_i\}, \{v_{i+1}, \ldots, v_n\}\}$, it must be that Cost$(\cong_i) \leq (b-1)\log_2(W) + b$.*

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

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

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

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

Assume, then, that $j > 1$. By definition of the search strategy, the $j-1 > 0$ subtrees with larger weights have already been enumerated, and so the cost must include the $j-1$ links to the parent. By Lemma 3.4.1, the j^{th} subtree can have weight no more than $(W - w_r)/j$. Hence, using the inductive hypothesis, partially enumerating t_j can cost no more than $(b-1)\log_2((W - w_r)/j) + b$, or equivalently $(b-1)\log_2(W - w_r) - (b-1)\log_2(j) + b$.

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

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 3.4.1 *If T is a tree with total weight W as described in Lemma 3.4.2 with the extra condition that for all weights w_i, $w_i \leq m$ for some $m > 0$, then for any $0 \leq \omega \leq W$, there exist two partitionings, $\cong_s$ and $\cong'_s$, of T into two parts T_1, T_2 and T'_1, T'_2 respectively such that T_1 has total weight $\omega - m'$ for some $0 \leq m' \leq m$, T'_1 has total weight $\omega + m''$ for some $0 \leq m'' \leq m$, and and the total cost of either partitioning is no more than $(b-1)\log_2(W) + b$.*

Proof: Let $v_1, \ldots, v_n$ be the n vertices of T ordered as per a post-order search examining child trees in order of decreasing total weight. Since no vertex has weight larger than m, there must exist some i such that $w_1 + \cdots + w_i = \omega - m'$ for some $0 \leq m' \leq m$. Similarly, there must exist some j such that $w_1 + \cdots + w_j = \omega + m''$ for some $0 \leq m'' \leq m$. By Lemma 3.4.2, both partitionings have cost no more than $(b-1)\log_2(W) + b$. □

Lemma 3.4.2 and Corollary 3.4.1 establish an upper bound on the cost of a partitioning and the maximum difference between partition sizes, respectively. Partitioning, however, is into p pieces where p can be anywhere between 1 and W, the total weight of the tree. We would like, then, bounds on the cost and size of partitions when dividing the tree into p pieces for any $1 \leq p \leq W$. Given a tree T as in Lemma 3.4.2, and a post-order search of its weighted vertices, we can consider the problem of producing such a weight-balanced p-partitioning T to be equivalent to the problem of p-partitioning an ordered sequence of non-negative integers summing to W, each of which is no more than m.

Lemma 3.4.3 *Given an ordered list of n integers, $N = w_1, \ldots, w_n$ such that $0 \leq w_i \leq m$ and $W = \sum w_i$, N can be partitioned into $1 \leq p \leq W$ disjoint, contiguous and covering sets, such that each partition has sum $W/p \pm m$.*

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

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

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

Assume true then for $x-1$, and let $x > 1$. Let the actual weight of N be $W' = Wx/p + m'$, for some $0 \leq m' \leq m$ (the other case, $W' = Wx/p - m'$, is symmetric). By Corollary 3.4.1, N can be split either at $\omega + m_1$ or $\omega - m_2$, for any given $0 \leq \omega \leq W'$ and some $0 \leq m_1, m_2, \leq m$, so remove from the front a contiguous partition N_1 of size $W/p + m_1$. The remaining partition, N_2 is also contiguous and has weight $W' - W/p - m_1 = W(x-1)/p \pm m$, so by inductive hypothesis N_2 can be partitioned into $x-1$ pieces, each with weight $W/p \pm m$.

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

□

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

Corollary 3.4.2 *If T is a b-ary tree, as per Lemma 3.4.2 with an upper bound m on the weight associated with each vertex, then T can be partitioned into $1 \leq p \leq W$ pieces, each of which has total weight $W/p \pm m$, and total cost no more than $2(b-1)\log_2(W) + 2b$.*

Proof: By Lemma 3.4.3, T can be partitioned into p pieces such that each partition has weight $W/p \pm m$. Each such partition is completely separated from the rest of the tree by no more than two cuts, each of which can be seen as a split of T into two pieces. Hence, by Corollary 3.4.1, each partition can have cost no more than twice $(b-1)\log_2(W) + b$. □

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

3.5 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.

3.5.1 Tree Partition Schemes

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

Definition 3.5.1 *A* **tree partition scheme** *for a dangling graph D with nodes V, 1/2-edges E and connection relations $C \subset E \times E$ is a tree T with nodes N and directed links L, together with a function $\nu : V \rightarrow N$ and a relation*[2] $\tau \subseteq C \times L$ *such that:*

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

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

An example of a graph embedded in a tree partition scheme is shown in Figure 3.3. 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

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

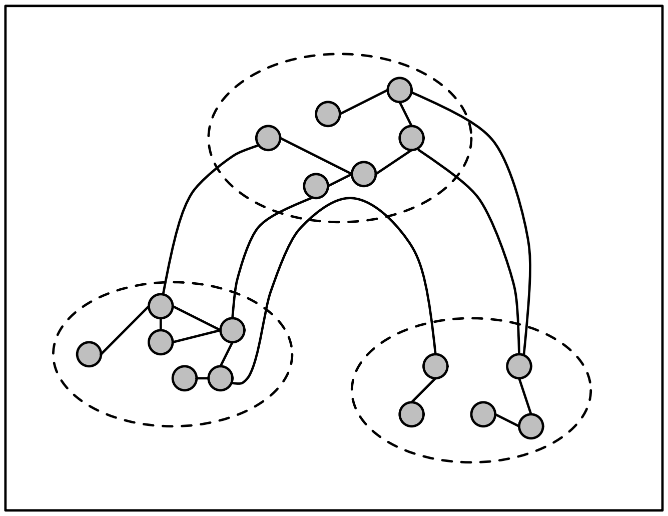


Figure 3.3: A graph embedded into a tree partition scheme.

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

1. *β is a bound on the branching factor of T.*
2. *μ is a bound on the size of $\nu^{-1}(n)$, $\forall n \in N$.*
3. *λ is a bound on the size of $\tau^c(\ell)$, $\forall \ell \in L$.*

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

Lemma 3.5.1 *Let $T(\nu, \tau, \beta, \mu, \lambda)$ be a bounded tree partitioning scheme of n nodes for some dangling graph D of $|V|$ nodes, as detailed above. Then D can be partitioned into p pieces each of size $|V|/p \pm \mu$ with maximum cost $2\lambda(\beta - 1)\log_2(|V|) + 2\lambda\beta$.*
Proof: This follows directly from Corollary 3.4.2. □

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

Definition 3.5.3 *The* **bounds** *of a dangling graph grammar $G = (A, \Upsilon)$ are three positive integers, m, g, and k such that $|A| \leq m$, $\forall n \in V_A$, Degree$(n) \leq k$, and $\forall\ (S, T, \delta) \in \Upsilon$ we have $|T| \leq m$, $|S| \leq g$, and Degree$(v) \leq k$ for all vertices v in T.*

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

Lemma 3.5.2 *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$(n) + w_n \leq m$, where w_n is the weight of node n. Moreover, let O be an occurrence of a production in Υ in D; then if v, v' are graph vertices in O, $\nu(v) = \nu(v')$.*

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

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

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

The graph D is the rewrite of D' by the productions in Υ. Hence, there is a set $\mathcal{O}$ of all occurrences that transformed D' to D. As well, and because no node is rewritten by more than one production, a function exists $\kappa : \mathcal{O} \rightarrow \{Z | Z \subseteq_i D\}$, 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 $\sigma : \mathcal{O} \rightarrow N'$ exists associating occurrences with the tree node containing the vertices forming the occurrence.

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

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

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

Include new edges in τ Let $C_O = \{(e, e') \in C_D | \ (e, e') \in \text{CSet}(\kappa(O), D - \kappa(O))\}$. Increase the relation τ to map each connection in C_O to the tree edge $(\sigma(O), \zeta(O))$.

Each $\kappa(O)$ is linked to the rest of D only by modifications to the original connection set between O and D' (see Proposition 3.3.1). Since each occurrence consists of at most g graph nodes, of degree at most k (by assumption), there can be at most gk distinct[3] connection relations between $\kappa(O)$ and $D - \kappa(O)$. Hence, τ maps no more than gk graph edges to the tree edge $(\sigma(O), \zeta(O))$. Note that all other differences between τ'

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

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 are have been altered by a substitution using some δ operator(s). Increase τ to map (e, e') onto $\tau'(f, f')$. This process does not alter the mappings of any newly created tree edge, and only replaces a former mapping ((f, f') will not exist in D) with a corresponding new one, so any bounds on the size or claims about connectivity for T' will continue to hold in T.

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

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

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

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

It is necessary to ensure that any future occurrences of these productions will have all their graph vertices mapped by ν to the same tree node. Consider an occurrence 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 $O_v \in \text{Overlap}(S_\rho, \tau)$; every 1/2-edge of O_v is either a 1/2-edge of $O \equiv S_\rho$, or it matches a 1/2-edge of τ—a 1/2-edge e of O_v which is not in $\Xi(O)$ and is connected to some other 1/2-edge e' and vertex v' in O. If there is a corresponding match for e' and v' in τ, then O_v is not maximal; if there is not and the image of e is not dangling in τ, then it cannot be that O is an occurrence. Thus, there is a member of $\text{Overlap}(S_\rho, \tau)$ which is neither $\emptyset$, nor the same as S_ρ (O_v includes v but not v'), and the grammar cannot be ST-overlap free.

The only remaining property to check is the assertion that no subtree of T exists with total weight 0. The above construction generates tree nodes for each 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. $\square$

This lemma leads directly to our main result:

Theorem 3.5.1 *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 $A \overset{*}{\rightarrow}_{\Upsilon} D$, it must that D is $(2gk(m-1)\log_2(|V|) + 2gkm)$-partitionable.*

Proof: This follows trivially from Lemma 3.5.2 and Lemma 3.5.1. By the former, for each dangling graph generated by G there is a corresponding bounded tree partition scheme $T(\nu, \tau, m, m, gk)$, and by the latter such a tree can be partitioned into pieces of size $|V|/p \pm m$ with maximum cost $2gk(m-1)\log_2(|V|) + 2gkm$, for any $1 \leq p \leq |V|$. $\square$

3.6 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 Figure 3.4).

$$\begin{array}{ccccccc}
c- & -a- & -a- & \ldots & -a- & -a- & -c \\
| & | & | & \ldots & | & | & | \\
c- & -a- & -a- & \ldots & -a- & -a- & -c \\
| & | & | & \ldots & | & | & | \\
c- & -a- & -a- & \ldots & -a- & -a- & -c \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}$$

Figure 3.4: A schematic rectangular grid.

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 Figure 3.5, one as in Figure 3.6, and so on. It would certainly be easier to write one rule just indicating the pattern, as in Figure 3.7.

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 graphs; including, for example, the class of rectangular grids shown in Figure 3.4. 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.

$$\begin{array}{ccc} | & | & | \\ c- & -a- & -c \end{array} \quad \Rightarrow \quad \begin{array}{ccc} | & | & | \\ c- & -a- & -c \\ | & | & | \\ c- & -a- & -c \end{array}$$

Figure 3.5: One of an infinite family of rules.

$$\begin{array}{cccc} | & | & | & | \\ c- & -a- & -a- & -c \end{array} \quad \Rightarrow \quad \begin{array}{cccc} | & | & | & | \\ c- & -a- & -a- & -c \\ | & | & | & | \\ c- & -a- & -a- & -c \end{array}$$

Figure 3.6: Another of an infinite family of rules.

$$\begin{array}{c} | \\ c- \end{array} \left(\begin{array}{c} | \\ -a- \end{array} \right)^{*} \begin{array}{c} | \\ -c \end{array} \quad \Rightarrow \quad \begin{array}{c} | \\ c- \\ | \\ c- \end{array} \left(\begin{array}{c} | \\ -a- \\ | \\ -a- \end{array} \right)^{*} \begin{array}{c} | \\ -c \\ | \\ -c \end{array}$$

Figure 3.7: A schematic rule, representing an infinite family of rules.

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.

3.6.1 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 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 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 3.6.1 *A* path expression *is defined inductively as follows:*

1. *A dangling graph G of one node is a path expression. All 1/2-edges are considered* free.

2. *If G and H are path expressions with free edges $E = \{e_1, \ldots, e_n\}$ and $F = \{f_1, \ldots, f_n\}$, and $\lambda : E \leftrightarrow F$ is a partial bijection, then $(G \circ_\lambda H)$ is a path expression with free edges $\{x | (x \in E \wedge \nexists y \in F. \lambda(x) = y) \vee (x \in F \wedge \nexists y \in E. \lambda(y) = x)\}$.*

3. *If G and H are path expressions with free edges $E = \{e_1, \ldots, e_n\}$ and $F = \{f_1, \ldots, f_n\}$, and $\lambda : E \leftrightarrow F$ is a partial bijection, then $(G|_\lambda H)$ is a path expression with free edges: $\{(e|f) | \lambda(e) = f\}$.*

4. *If G is a path expression with free edges $E = \{e_1, \ldots, e_n\}$, 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 1/2-edge not involved in λ as there are replications of G, but that there will be only one copy of each 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 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 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 1/2-edge involved in λ, and the very first copy of any 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 $\mathcal{L}(P)$.

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 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 3.1 Consider the following path expression.

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

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

Definition 3.6.3 *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 3.2 As an example, consider the following path expression[4] and its associated skeleton:

$$\begin{aligned} P &= (a \circ b)|(((d \circ e)+)|((f)+)) \circ ((g)+) \\ \partial P(\#1, \#2, \#3, \#4) &= (\#1)|(((\#2)+)|((\#3)+)) \circ ((\#4)+) \end{aligned}$$

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 3.6.4 *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 $|P| = 1$.*

[4] 1/2-edges and 1/2-edge labels are not shown.

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 $|P| = |G| + 1$.*

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 3.6.1 *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 $|F| \leq k\ell$.*

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 $\ell - 1 \geq 1$, 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+_λ), 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 3.6.1 *Let P be a path expression of length ℓ over nodes with bounded degree k. Then any graph $G \in \mathcal{L}(P)$ 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 $\leq \ell - 1$, 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 $1 \leq \ell_1, \ell_2 \leq \ell - 1$. By inductive assumption then, any graph specified by P_1 or P_2 is $k(\ell-1)^2$-partitionable. Moreover, by Proposition 3.6.1, 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+_λ), 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 3.6.1 *Let P be a path expression with length and maximum degree bounded by a constant. Then any graph $G \in \mathcal{L}(P)$ is $O(1)$-partitionable.*

Proof: This follows immediately from Lemma 3.6.1. □

3.6.2 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:

$$(a \circ b) \longrightarrow (e|f)$$

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:

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

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 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 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 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 3.6.5 *Let S and T be path expressions, such that $\partial S = \partial T$. Let $[S] = (C_1, \ldots, C_c)$ and $[T] = (D_1, \ldots, D_c)$, for some c, with $F_1, \ldots, F_c$ and $G_1, \ldots, G_c$ their corresponding free sets. Let $\delta_1, \ldots, \delta_c$ 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 $\forall G \in \mathcal{L}(S)$ G is connected, $(S, T, \delta_1, \ldots, \delta_c)$ form a* path-extended production.

REMARKS: 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.

3.6.3 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 3.6.6 *Let $\mathcal{P}$ be a set of (path-extended) productions. Then $\mathcal{P}$ is* SS-overlap free *if the set:*

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

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 this 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 productin 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 3.6.2 *Let $\mathcal{P}$ be a set of (path-extended) productions. Then $\mathcal{P}$ is deterministic if $\mathcal{P}$ 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. □

REMARKS: Note that the expression in Example 3.1 could not appear in the source of any production in an SS-overlap free set of productions. There are two concrete expressions,

$$\begin{pmatrix} |_p \\ n \\ {}_l/\backslash_r \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} |_p \\ n \\ {}_l/\backslash_r \end{pmatrix}$$

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

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

is ST-overlap free.

3.6.4 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 1/2-edges in the free sets, this same concept can be used to generate TPS's 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 $T(\nu, \tau, \beta = m, \mu = m, \lambda = gk)$ 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 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 $k\ell$ such connections. The following results formalize this argument.

Proposition 3.6.3 *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 $G \in \mathcal{L}(P)$. Then C is connected to G by no more than $k\ell$ 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 $k\ell$ connections emanating from C. □

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 $(A, (\Upsilon - \Upsilon') \bigcup \widehat{\Upsilon'})$ 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 3.6.2 *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 $\mathcal{T}(\nu, \tau, \beta = m, \mu = m, \lambda \leq \max(\Delta, k\ell + gk))$ for $\Delta \leq \min((gk + k\ell)(k\ell)^s + u((k\ell)^{s+1} - 1)/(k\ell - 1) - 1, gk + k\ell + uk\ell s(d+1))$ such that $\forall n \in N$, $Fanout(n) + w_n \leq m$, where w_n is the weight of node n. Moreover, let O be an occurrence of a production in Υ in D; then if v, v' are graph vertices in O, $\nu(v) = \nu(v')$.*

Proof: By induction on the size of s. Let $\mathcal{T} = (N, L)$, 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, $s = 0$, 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 $\mathcal{T}'(\nu', \tau', \beta', \mu', \lambda')$ for D' with the above properties; we will show how to extend $\mathcal{T}'$ to a bounded tree partition scheme $\mathcal{T}$ for D.

The actions of the regular productions on the TPS have already been determined; assume then that a new TPS, $\mathcal{T}$, has been constructed from $\mathcal{T}'$ according to the actions of $(A, (\Upsilon - \Upsilon') \bigcup \widehat{\Upsilon'})$ and as described in Lemma 3.5.2, 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 $\mathcal{T}$, 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 $\max(\lambda', gk + k\ell)$; as well, since dead branches have not been pruned, all nodes and links of $\mathcal{T}'$ are contained in $\mathcal{T}$. 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 3.6.3 the embedded image of any target in $\widehat{\rho}$ in D can be disconnected with cost at most $k\ell$. 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 $\mathcal{T}'$ and $\mathcal{T}$) 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 $k\ell$ 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 $\mathcal{T}$ stipulates that τ still maps connections like e' to links of $\mathcal{T}$. We can replace each such e' with at most $k\ell$

connections each application of each path-extended production. This implies a multiplicative increase in λ for any existing link of no more than $k\ell$. However, it is also true that no more than $uk\ell$ 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 $udk\ell$ 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 $k\ell$ such connections, and so τ will have to associate at most $k\ell$ more connections with any given link in $\mathcal{T}$ 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 $uk\ell$ for the latter connections, and either a multiplicative increase by $k\ell$ or an additive increase by $udk\ell$. Applying these increases to the inductive hypothesis results in the described bounds. □

Theorem 3.6.2 *Let $G = (A, \Upsilon)$ be a path-extended grammar as just described. Then for any dangling graph D such that $A \overset{s}{\rightarrow}_{\Upsilon} D$, it must be that D is $O(c^{s+1}\log(|V|))$-partitionable, and $O(sd\log(|V|))$-partitionable.*

Proof: Lemma 3.6.2 establishes the existence of a tree partition scheme where λ is bounded as described. By Lemma 3.5.1 this TPS can be partitioned with cost $O(\log(|V|))$. □

3.7 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.

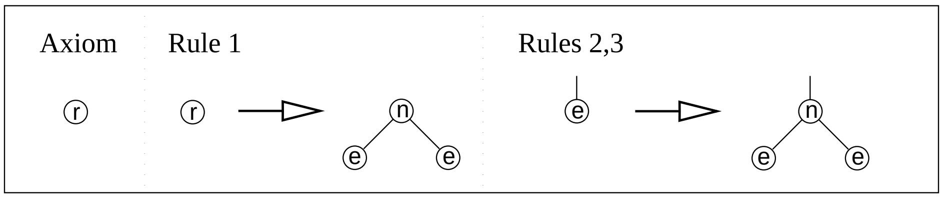


Figure 3.8: A grammar generating trees.

3.7.1 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.

k-ary Trees

The class of k-ary trees is trivial to generate. The axiom consists of just a single node, labelled `root`. Two rules then suffice to expand either the root or a leaf into an internal node and k child leaves. In the case of a leaf being rewritten, the 1/2-edge connecting the rewritten leaf to its parent is associated with the 1/2-edge extending from the internal node. This is illustrated in Figure[5] 3.8; 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.

Threaded k-ary Trees

By adding two 1/2-edges to each node, for left and right threaded neighbours, threading can be maintained as the leaves are expanded (see Figure 3.9; 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 (Figure 3.10). Our examples illustrate binary trees and inorder threading, but clearly it is possible to accommodate any recursive threading policy in the same way.

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

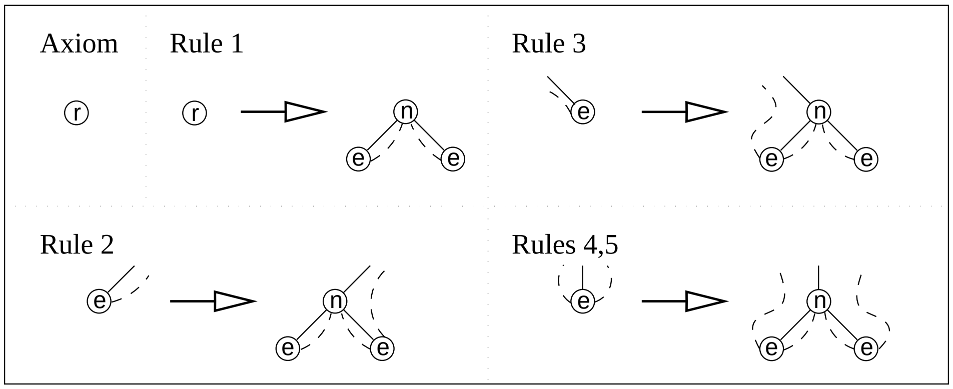


Figure 3.9: A grammar generating inorder threaded (binary) trees.

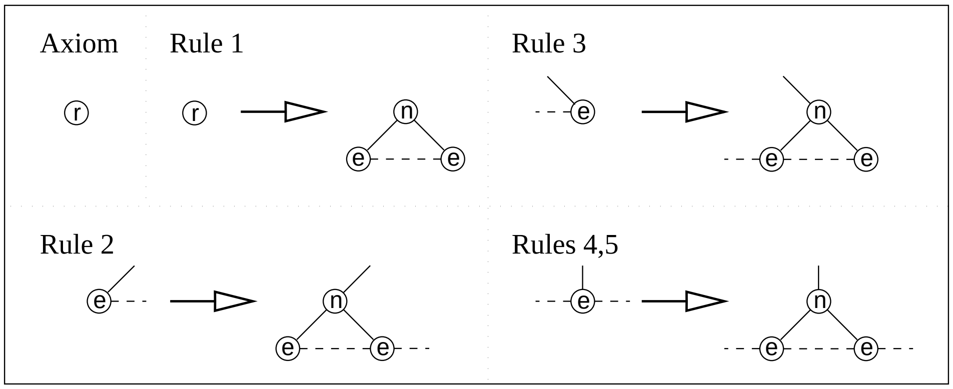


Figure 3.10: A grammar generating leaf-threaded trees.

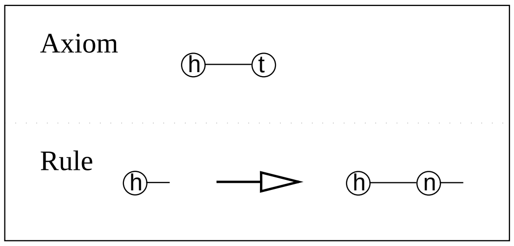


Figure 3.11: A grammar generating linked lists.

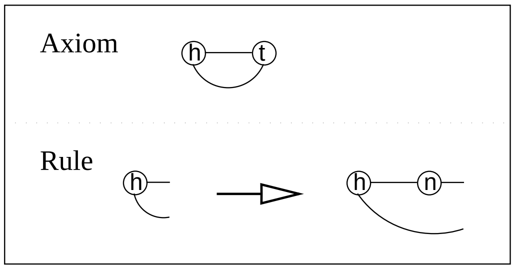


Figure 3.12: A grammar generating circular linked lists.

Linked Lists

Normal linked lists can be easily generated by marking the tail and/or head of the list distinctly, and then generating new entries by expanding the tail or head into an internal node and a new tail or head (Figure 3.11). If the list is intended to be circular, then an extra connection between the head and tail is maintained through the rewrites (Figure 3.12). Different orders of application for the rules then correspond to the different variations on lists—stacks, queues, double-ended queues, *etc.*

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 Figure 3.13. 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

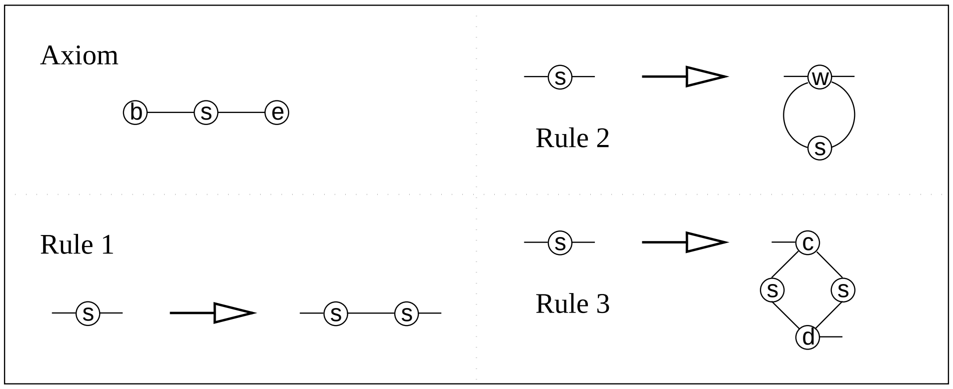


Figure 3.13: A grammar generating compiler control flow graphs.

control flow out of the conditional (Rule 3).

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 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.

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 Figure 3.14). 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 3.6.2 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

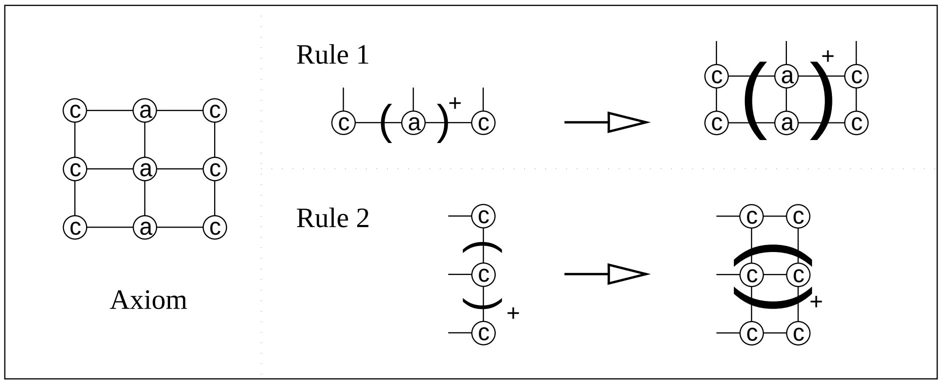


Figure 3.14: A path-extended grammar generating rectangular grids.

is produced by first generating the width and then the height, the TPS inductively constructed according to Lemma 3.5.2 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 3.6.1 the TPS can be partitioned with cost $O(1)$—reducing the cost of partitioning the grid to $O(\sqrt{n})$.

3.7.3 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 Figure 3.15).

3.8 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

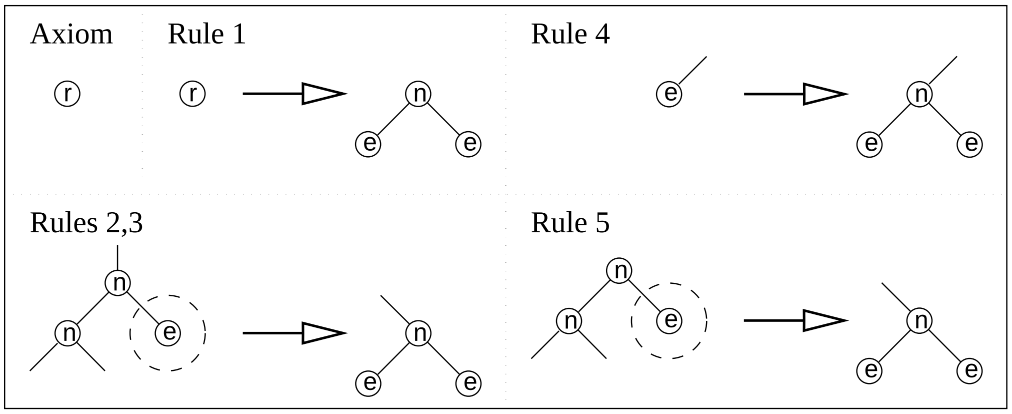


Figure 3.15: A grammar requiring contexts; right leaves are expanding only if the left sibling is not a leaf.

for which polynomial-time (and often linear-time) algorithms exist for a variety of problems in NP [Arn85, ALS88, Cou90a, Klo94, Sli82]. 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 3.8.1 *Let $T = (N, L)$ be a tree with nodes N and edges L. Then $\pi : N \times N \rightarrow \mathcal{P}(N)$ is a function returning the set of vertices forming the loop-free path connecting the two input vertices.*

Definition 3.8.2 *Let $G = (V, E)$ be a graph. A* **tree-decomposition** *of G is a tree $T = (N, L)$ and a function $\sigma : N \rightarrow \mathcal{P}(V)$ such that:*

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

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

Lemma 3.8.1 *Let $D = (V, E, \nu_D, \phi, \psi, \Sigma_V, \Sigma_E, C)$ be a dangling graph, and let $T(\nu_T, \tau, \beta, \mu, \lambda) = (N, L)$ 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 $\ell(n)$ be the set of outgoing links attached to a node n in N. Define $\sigma : N \rightarrow \mathcal{P}(V)$ as:

$$\begin{aligned}\sigma(n) \;=\; & \{v \in V \mid \nu_T(v) = n\} \bigcup \\ & \{v \in V \mid \exists e, e' \in E,\ \exists l \in \ell(n).\ \nu_D(e) = v \ \wedge\ (e, e')\tau l\}\end{aligned}$$

The function σ maps each tree node n in T to the set of vertices which are mapped by ν_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 $\epsilon : N \rightarrow \mathcal{P}(V)$ 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 $\epsilon(n) \subseteq \epsilon(n')$. Also note that by definition of TPS, a vertex v is in $\epsilon(n)$ for a tree node n if and only if $\exists n' \in \text{Subtree}(n).\ \nu_T(v) = n'$.

By definition of T, every vertex in V is already uniquely mapped by ν_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 ν_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 $\nu_T(v)$ or $\nu_T(v')$ must not contain the other, and so c is a connection relation which must be broken to separate $\epsilon(\nu_T(v))$ from $\epsilon(\nu_T(v'))$. Now, by definition of T, c is associated by τ to each link along $\pi(\nu_T(v), \nu_T(v'))$, 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) $\nu_T(v) = n$, or 2) some connection to v is associated by τ to an outgoing link of n.

1. $v \in \epsilon(n)$, $v \in \sigma(n')$ by reason 2). This means that in order to partition $\epsilon(m')$ 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 $\epsilon(m')$.
   (a) $v \in \epsilon(m')$. Then $\epsilon(n) \subseteq \epsilon(n')$ and $\forall \widehat{n} \in \pi(n, n')\ \epsilon(\widehat{n}) \subseteq \epsilon(n')$. The connection c must then be broken to partition any $\epsilon(\widehat{n})$, and is thus mapped by τ to all links along $\pi(n, n')$. By definition of σ, v will then be mapped by to all nodes in $\pi(n, n')$.
   (b) $v' \in \epsilon(m')$. Then either $\epsilon(n') \subseteq \epsilon(n)$ or $\epsilon(n)$ and $\epsilon(n')$ are disjoint.
      i. $\epsilon(n') \subseteq \epsilon(n)$. 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 $v \in \epsilon(\widehat{n}')$, and $\forall \widehat{n}' \in (\pi(n', \widehat{n}) - \{\widehat{n}\})$ we have $v \notin \epsilon(\widehat{n}')$. Each of the former must be such that either $\nu_T(v) = \widehat{n}'$ or there is some child $\widehat{m}'$ with $v \notin \epsilon(\widehat{m}')$ or there is some child $\widehat{m'}$ with a neighbour v'' of v outside $\epsilon(\widehat{m}')$. 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 $\epsilon(\widehat{m}')$, for some child $\widehat{m}'$, and so $\sigma(\widehat{n}')$ will include v.
      ii. $\epsilon(n)$ and $\epsilon(n')$ 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 $\epsilon(\widehat{m})$ contains only one of $\epsilon(n)$ or $\epsilon(n')$. Hence by definition of σ, v is in $\sigma(\widehat{n})$.
2. $v \notin \epsilon(n)$, $v \in \sigma(n')$ by reason 2). Then $v \in \sigma(n)$ by reason 2) as well, and $\epsilon(m)$ for some child m of n contains a neighbour v' of v. If $v \in \epsilon(n')$ then of course the situation is symmetric to case 1, so we can assume $v \notin \epsilon(n')$. Any $\widehat{n} \in \pi(n, n')$ such that $v \notin \epsilon(\widehat{n})$ 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 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 ν_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. $\square$

Theorem 3.8.1 *Let G be an ST-overlap free dangling graph grammar with constant bounds (m, g, k). Let $D = (V, E, \nu, \phi, \psi, \Sigma_V, \Sigma_E, C)$ be a dangling graph such that $A \xrightarrow{*}_{\Upsilon} D$, necessarily D has tree-width smaller than $2mgk + m$.*

Proof: By Lemma 3.5.2 a bounded tree partition scheme $T(\nu_T, \tau, \beta = m, \mu = m, \lambda = gk)$ exists for D. By Lemma 3.8.1 this implies an upper bound on tree-width of $2mgk + m - 1$. □

Theorem 3.8.2 *Let G be an ST-overlap free path-extended dangling graph grammar with 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 $(A, (\Upsilon - \Upsilon') \bigcup \widehat{\Upsilon'})$ have constant bounds (m, g, k). If $D = (V, E, \nu, \phi, \psi, \Sigma_V, \Sigma_E, C)$ 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 $\Delta \leq \min((gk + k\ell)(k\ell)^s + u((k\ell)^{s+1} - 1)/(k\ell - 1) - 1, gk + k\ell + uk\ell s(d + 1))$.*

Proof: By Lemma 3.6.2 a bounded tree partition scheme exists for D with the given bounds. By Lemma 3.8.1 this implies the upper bound on tree-width. □

Corollary 3.8.1 *Rectangular grids of size $w \times h$ have tree-width of $O(w + h)$.*

Proof: By construction of the grammar in Figure 3.14, 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 3.8.1, and the construction in section 3.7.2 the upper bound on tree-width follows. □

Corollary 3.8.1 jibes nicely with existing results; it is known that square grids of $\sqrt{n} \times \sqrt{n}$ vertices $(n \geq 2)$ have a tree-width of $\sqrt{n}$ [RS86].

3.9 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.

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 [Arn85, ALS88, Cou90a, Lau88a, Lau88b, Lau90b, RS86, Sli82]. 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.

The grammars we have defined form an interesting basis for synchronous parallel algorithms on irregular graphs. Any graph, and any algorithm which can be expressed in the grammar formalism, is either reasonably-partitionable or has an easily computed bound on partitionability, and hence is amenable to parallelization through automatic partitioning. The implicit load-balancing and guaranteed bounds on communication cost mean that as long as computation is reasonably uniform throughout the data structure, the computation will be efficiently parallel. As well, because the partitioning is often determined incrementally by the derivation, the efficiency can sometimes be maintained as the graph is grown and modified. In the next chapter we illustrate the application of graph grammar generated partitionable data structures through a new explicitly parallel language, called "**eL**".

Chapter 4

A Graph Grammar Language: eL

The parallelization of programs involving irregular, dynamic data structures is very complicated. There are no general techniques that one can apply, and usually one resorts to an *ad hoc* solution for each individual problem. However, the grammars developed in the previous chapter demonstrate that many such structures nevertheless possess enough regularity to permit automated partitioning. If a dynamic algorithm can be expressed, then, using the prescribed grammar formalism we should be able to efficiently parallelize the algorithm.

There are obstacles to this solution. Primarily, existing parallel languages (and parallel language hybrids) do not support the necessary constructs. In order to discover the partitioning in the constructive manner we have indicated, it is necessary for the programmer to separately design the grammar and generate the associated TPS, and use this to develop the correct partitioning—a non-trivial amount of extra work. This also points to another unacceptable aspect; the actions of the grammar are images of the actions of the program on the data structure: what is done to the data structure by the program is also done to an equivalent graph by the grammar. The grammar thus not only requires a significant amount of work, but it seems like most of that extra work is a duplication of work already done.

Both of these problems can be eliminated if the computation and the grammar are integrated. This is the basis for the language "**eL**[1]"; **eL** is an inherently parallel programming formalism modelling the construction and manipulation of sparse, doubly-connected dynamic data structures. **eL** explicitly represents the data structure and computations/manipulations on it as a graph grammar. In **eL** the data

[1] "The language formerly known as L".

structure *is* the graph, and actual data values are stored in the nodes of the graph. Connections between nodes then represent bidirectional pointers, and dangling edges in the graph are meant to represent `NULL` pointers. Computations are integrated by allowing productions to rewrite the data in nodes as well as the nodes themselves; target graphs specify data values using arithmetic expressions involving data in the source nodes. The execution model is one of synchronous parallelism; the program runs by iteratively applying all rules concurrently and everywhere possible, in a manner similar to cellular automata. Thus, all data accesses, all computations and even the parallelism is subsumed by the graph grammar model.

4.1 Language Definition

The definition of a dangling graph grammar merely requires a graph and some rules. A programming language, however, requires considerably more structure; for instance, procedural programs are typically organized as a sequence of "phases." Each phase includes only a limited number of the total set of computations, and requires the completion of the previous phase in order to begin. In C and Pascal, for instance, phases usually correspond to procedures.

In **eL**, phases are facilitated by grouping rules together into *blocks.* Each block is a small, encapsulated graph grammar, iteratively and synchronously applying the rules in its scope to the current graph. Control is maintained through a stack of block names; only the rules contained within the block given by the top of the stack are actively applied. Once the block phase is complete, control transfers to whichever block is now at the top of the stack. The current state of an **eL**-program is completely described by the current graph and the block stack.

4.1.1 Blocks

A block actually consists of a collection of rules, a *graph expression* (GE), a *preblock list* and a *postblock list.* The GE is a boolean expression on graphs, and determines when a block phase is over: once a block becomes active, it remains active until its GE is satisfied. Each iteration all graphs g in the GE of the active block are replaced by a boolean (*tt* or *ff*), representing their presence or absence in the current graph (iterated axiom). If after these replacements the GE evaluates to *tt* then the block terminates and its name is popped from the top of the stack, otherwise another iteration ensues. The evaluation of the GE is carried out at the same time as the rules are applied.

Postblock lists and *preblock lists* are used for manipulating the block stack in more complex ways. If the current block is terminating and it has a postblock list, then this list of block names is pushed onto the stack after the current block is popped; this allows blocks to chain computations. The *preblock list* is used to iterate blocks; each iteration that the GE evaluates to *ff*, the preblock list is pushed on top of the current block. A preblock allows a sequence of blocks to be iterated until the GE is met. Note that if we consider the preblock list to always begin with the current block name then even blocks with an empty preblock are subsumed by this behaviour.

EXAMPLE 4.1 Let

$$B_1, \ldots, B_{n-1}, B_n$$

be a stack of block names, with B_n being the top of the stack. Then B_n is the *active* block, and only the rules in block B_n are applied each iteration. Suppose B_n has a preblock list $R_1, \ldots, R_r$ and a postblock list $O_1, \ldots, O_o$. After one iteration, if the GE of B_n is not satisfied, then the preblock list is pushed onto the stack, producing a stack like:

$$B_1, \ldots, B_{n-1}, B_n, R_1, \ldots, R_r$$

and making R_r the active block. Alternatively, if the GE of B_n is satisfied the postblock list is pushed, producing:

$$B_1, \ldots, B_{n-1}, O_1, \ldots, O_o$$

making O_o the active block.

This form of control structure has been chosen to allow easy integration with the graph-rewriting concept which is the basis for our language. One can imagine an actual stack of block names being maintained alongside the iterated axiom. Each rule in a given block includes the block name attached to a "top of stack" marker as a context, and so is restricted to acting only when the appropriate block is active. Transfers of control between blocks are then just rules rewriting the active block and top-of-stack marker to other lists of block names (the preblock and postblock lists).

Note that while this metaphor of control conceptually fits the grammar formalism, it would require every node in the graph to be connected to every node in the block stack—our grammars demand bounded-degree, and so this sort of structure would have to be treated as an exception. Still, the use of a block stack provides a smooth paradigm for thinking about the grammars.

4.1.2 Rules

Rules are the basis for computation in **eL**. Each rule specifies a source graph, which is pattern-matched to the current graph, a target graph (the intended replacement for the source graph), and a mapping (δ) from the dangling edges of the source to the target (also known as the "embedding function"). In order to accommodate data manipulations, each datum in the target graph is allowed to be expressed as an arithmetic function of the data found in the source. A rule is then applied by locating an exact replica of the source graph, inserting a (new) copy of the target in its stead with data values computed from the data in the actual occurrence of the source, and linking the new target to the rest of the graph by replacing any half-edge of the source involved in a connection relation with its image under δ (or discarding the connection relation if it has no image under δ).

Often it is useful to have several rules apply to the same source graph, but perform different rewrites based on the data of the occurrence. Each rule can therefore include a *guard,* which is just a boolean expression on any or all data contained in the nodes of the source graph. If a guard is present then the rule will only apply if its source matches, *and* the guard evaluates to *tt*. Note that this enhancement can cause conflicts with the no SS-overlap requirements of our base dangling graph grammars; our grammar will not be SS-overlap free if two identical rules can exist, both of which might be applicable to the same occurrence. For this reason it is necessary that the guards of any SS-overlapping rules specify mutually exclusive boolean expressions.

Contexts

When programming, one often finds that the same datum is altered differently depending on the surrounding context; for instance, in a red-black tree[2] we may wish

[2]A self-adjusting binary search tree where each node has an associated colour—red or black.[CLR90]

to rewrite each node to a different colour based on the colours of its neighbours. We could rewrite the node *and* its neighbours, but our rules are applied concurrently and everywhere possible. Each neighbour may therefore be itself being rewritten, and so we run the risk of two occurrences overlapping, forming a critical pair.

For this reason, **eL** permits the source graph of any production to be included in a *context* graph. This context is not rewritten and is not factored into the embedding (δ) function (though data in the context is accessible to the functions computing target data), but it must *occur* (pattern match) in the same way as a regular source graph; overlap, however, is still determined entirely by the actual rewritten source graph. This way several rules which all depend on the same local context, but which rewrite distinct portions of that context, can be applied without introducing extraneous overlap. Contexts affect the graph grammar bounds, but only by a constant amount—and since their only function is to control rule application, they do not otherwise need to be factored into the semantics.

4.2 Operational Semantics

A dangling graph is usually represented by several parameters, including labelling functions. In order to keep the operational semantics uncluttered, we can abbreviate the definition to just three elements: (V_G, E_G, C_G) indicates a dangling graph G with nodes V_G, half-edges E_G and connection relation $C_G \subset E_X \times E_G$. Also note that guards and arithmetic functions from source to target will not be described in the semantics either; these constructs are straightforward language extensions which do not appreciably change the structure of the semantics.

Each program consists of a collection of graph, node and block definitions. The block definitions contain the rules, which define all actual operations on the graph. By specifying how blocks are brought into existence, and back out again, we can define the overall operation of the program as a series of block transitions. The primary operation, though, is the transformation (rewrite) of the graph by the set of rules.

4.2.1 Rules

First, we define an aggregate replacement operation to model the concurrent application of some number of rules. Given a dangling graph G, and a set of pairs $\mathcal{O}$, each pair consisting of a rule r and an occurrence O of r in G, we can define the result of rewriting all these occurrences simultaneously ($G[\mathcal{O}]$) as follows.

Let $R_i = (S_i, T_i, \delta_i)$ be a rule in the current block. It is the intention that each $\delta_i : \Xi(S_i) \to \Xi(T_i)$ maps some dangling edges of the source graph to dangling edges of the target graph. This mapping is used to perform substitutions within the connection relation for the graph so as to insert (a copy of) the target graph in place of the (matched) source. Any dangling edges of the matched source which are not mapped by δ_i have their connection relation (if any) discarded.

It is somewhat easier if we promote each individual δ_i to a full function, returning a special character `0` for each member of $\Xi(S_i)$ not mapped to a member of $\Xi(T_i)$. We can then compose each of these individual rule functions to build a larger function representing the synchronous operation of all rules, without confusing half-edges untouched by the composite transformation (mapped to $\perp$) with half-edges whose connections should be discarded (mapped to `0`).

$$\frac{G, \mathcal{O} = \{(O_1, R_{j_1}), \ldots, (O_w, R_{j_w})\} \ \bigwedge \ \forall 1 \le i \le w, \ \left(\begin{array}{l} O_i \subseteq_i G \ \wedge \\ O_i \equiv_{h_{j_i}} S_{j_i} \wedge \ T^e_{j_i} \equiv_{g_{j_i}} T_{j_i} \wedge \\ \widehat{\delta}_i = g_{j_i} \circ \delta_{j_i} \circ h_{j_i} \ \bigwedge \ \delta = \bigsqcup_i \widehat{\delta}_i \end{array} \right)}{G = (V, E, C)[\mathcal{O}] \to G' = (V', E', C')} \tag{4.1}$$

where:

$$\begin{aligned} V' &\triangleq V \backslash (\{V_{O_i}.\ 1 \le i \le w\}) \bigcup \{V_{T^e_{j_i}}.\ 1 \le i \le w\} \\ E' &\triangleq E \backslash (\{E_{O_i}\ 1 \le i \le w\}) \bigcup \{E_{T^e_{j_i}}.\ 1 \le i \le w\} \\ C' &\triangleq e_1 C' e_2 \text{ if } \begin{cases} e_1 C e_2 \ \wedge \ \delta(e_1) = \perp, \ \delta(e_2) = \perp, \ e_1, e_2 \in E, \text{ or} \\ e'_1 C e_2 \ \wedge \ \delta(e'_1) = e_1, \ \delta(e_2) = \perp, \ e_2 \in E, \text{ or} \\ e_1 C e'_2 \ \wedge \ \delta(e'_1) = \perp, \ \delta(e'_2) = e_2, \ e_1 \in E, \text{ or} \\ e'_1 C e'_2 \ \wedge \ \delta(e'_1) = e_1, \ \delta(e'_2) = e_2, \text{ or} \\ \exists i.\ e_1, e_2 \in E_{T^e_{j_i}} \ \wedge \ e_1 C_{T^e_{j_i}} e_2 \end{cases} \end{aligned}$$

In other words, we remove all occurrences, add in distinct copies of all associated target graphs, and modify the connection relation by the composite δ. Connection relations are preserved if neither half-edge is touched by any rewrite, or if one half-edge is mapped to some embedded target half-edge and the other is untouched, or if both are mapped, or if the two half-edges were introduced as part of a connected pair in a target graph. Note that if either or both half-edges are mapped to 0, then the connection relation is discarded.

Of course we need to identify the occurrences before we can apply this operation. Let $\Upsilon_B = \{R_1, \ldots, R_r\}$ be the r rules in block B. Recall that for each rule $R_i = (S_i, T_i, \delta_i)$, there exists a function $\text{Occurs}_{R_i}(G)$ which returns $O \subseteq_i G$, such that $O \equiv_{h_i} S_i$, for some structure and label-preserving isomorphism h_i. The target itself is not embedded in the graph, of course, rather a fresh copy T_i^e is, and $T_i \equiv_{g_i} T_i^e$. The modifications to the graph each iteration depend on the aggregated replacement of all such occurrences of all rules in the block:

$$\frac{(\Upsilon_B = \{R_1, \ldots, R_r\}) \ \wedge \ \left(\mathcal{O} = \bigcup_i \{(O, R_i) |\ O \in \text{Occurs}_{R_i}(G)\}\right) \ \wedge \ G[\mathcal{O}] \Rightarrow G'}{G \Rightarrow_{\Upsilon_B} G'} \tag{4.2}$$

The above semantic rule requires that each occurrence be identified. Each rule is applied wherever possible, potentially producing several occurrences, so it is also necessary to retain which rule produced which occurrence. This is encoded by pairing occurrences and rules.

4.2.2 Blocks

The active block changes according to the evaluation of the associated *graph expression,* which is just a boolean combination of graph definitions. Let E be such an expression formed from graphs $G_1, \ldots, G_n$, and let $E[b_1, \ldots, b_n]$ be an identical expression with each G_i being substituted by the corresponding boolean value b_i. Let $E[b_1, \ldots, b_n] \rightarrow b$ represent the evaluation of the boolean expression, either resulting in $b = tt$ or $b = ff$. Then:

$$\frac{E \ \wedge (\forall i, b_i = (\text{Occurs}_{G_i}(G) \neq \emptyset)) \ \wedge \ E[b_1, \ldots, b_n] \rightarrow tt}{E \text{ satisfied in } G} \tag{4.3}$$

Let $B = (E_B, (B_1, \ldots, B_n), (P_1, \ldots, P_m), \Upsilon_B)$ be a block. E_B is the graph expression which terminates the block, $(B_1, \ldots, B_n)$ (for $n \geq 0$) is an ordered sequence of block names, forming the preblock, $(P_1, \ldots, P_m)$ (for $m \geq 0$) is an ordered sequence of block names, forming the postblock, and Υ_B is the set of rules in B. In the more usual **eL** notation, we might write schematically:

$$\texttt{block}\ B\ B_n \cdots B_1\ :\ E_B\ \rightarrow\ P_n \cdots P_1\ \{\ \Upsilon_B\ \}$$

A *state* in the computation is then just a dangling graph (initially the axiom), and a stack of blocks (initially just the starting block). The state can be represented as $(B : S, G)$, where B is name of the active block, S is the rest of the block stack (possibly empty), and G is the current graph.

Based on the two possible outcomes of the block expression, we then have two semantic rules to specify the major state changes which occur each iteration of the grammar. Let the active block, B, have preblock, postblock and graph expression as above, then:

$$\frac{(B : S, G)\ \bigwedge (G \Rightarrow_{\Upsilon_B} G')\ \bigwedge (E_B \text{ satisfied in } G)}{(P_1 : P_2 : \cdots : P_m : S, G')} \tag{4.4}$$

$$\frac{(B : S, G)\ \bigwedge (G \Rightarrow_{\Upsilon_B} G')\ \bigwedge (E_B \text{ not satisfied in } G)}{(B_1 : B_2 : \cdots : B_n : B : S, G')} \tag{4.5}$$

4.3 Grammar

eL is inherently visual. The most suitable method for developing an **eL** program is to represent it as a graph, and interactively draw and modify graphs. Unfortunately, most other computer languages are textual in nature; the sort of graphical editing environment which would be ideal for **eL** simply does not exist on most platforms. For this reason, the following text-based grammar has been developed. Note that because this grammar is from an **eL** $\rightarrow C$ translator, the grammar includes the ability to embed C-code directly in the **eL**-code, and in fact requires such embedded code in order to perform arithmetic operations on data. A self-contained **eL** grammar would define a complete set of arithmetic and boolean operations on data.

This grammar does not permit the use of path extended productions. All graphs must be fully-specified, with one exception. Instead of specifying a particular node type in the source graph of a production, a pair of nodes which are identical except for type name can be |-ed together, matching either node type. This is a simple optimization on code size, which does not change any theoretical bounds.

```
program          decls ;
decls            ε | decls decl ;
decl             nodetype | graphtype | axiom | ruletype
                 | blocktype | include String | c_code ;
c_code           ‘@’ c_codelines ‘@’ ;
nodetype         node Id ‘{’ nodedecls ‘}’ ;
nodedecls        ε | nodedecls nodedecl ;
nodedecl         nodedecltype nidlist ‘;’ | node Id ‘;’ ;
nodedecltype     Type | Id | link ;
nidlist          Id | nidlist ‘,’ Id ;
graphtype        graph idornot ‘{’ graphdecls ‘}’ ;
graphdecls       graphdecl | graphdecls graphdecl ;
graphdecl        link linkref ‘,’ linkref ‘;’
                 | Id gorlist pidlist ‘;’
                 | graph Id ‘;’ ;
gorlist          ε | gorlist ‘|’ Id ;
pidlist          Id | pidlist ‘,’ Id ;
linkref          Id linkrefend ;
linkrefend       ε | ‘.’ linkrefenddot ;
linkrefenddot    ε | Id | link ;
blocktype        block Id idseq ‘:’ graphexprornot ‘->’ idseq ‘{’ blockdecls ‘}’
                 | start Id c_codeornot ‘;’ ;
blockdecls       ε | blockdecls blockdecl ;
blockdecl        ruletype | nodetype | graphtype | c_code ‘;’ | Id ‘;’ ;
idseq            ε | idseq Id ;
ruletype         rule idornot ‘:’ idorgraph by ridlist c_codeornot
                 ‘->’ idorgraphornot conlinks c_codeornot ‘;’ ;
c_codeornot      ε | c_code ;
idornot          ε | Id ;
ridlist          Id | ridlist Id ;
conlinks         ε | linklist ;
linklist         relink | linklist relink ;
relink           ‘(’ linkref ‘=’ linkref ‘)’ ;
axiom            axiom idorgraph ‘;’ ;
```

idorgraphornot	ϵ | *idorgraph ;*
idorgraph	*Id* | *graphtype ;*
graphop	'&' | '|' ;
simplegraphexpr	*idorgraph* | '!' *simplegraphexpr* | '(' *graphexpr* ')' ;
graphexpr	*simplegraphexpr* | *graphexpr graphop simplegraphexpr ;*
graphexprornot	ϵ | *graphexpr ;*

4.4 Complexity

An **eL** program is composed of a series of block transitions, each of which is itself an iteration over a set of rules. Thus, the time complexity for an **eL** program can be calculated from the number of rules in each block, the number of times each block is active, and the size of the graph.

Let P be an **eL** program. Suppose on a particular run r of P there are ${}_rb$ block transitions, so the sequence of active blocks is ${}_rB_1, \ldots, {}_rB_b$. Let ${}_rr_i$ be the number of rules in block ${}_rB_i$, let ${}_rb_i$ be the number of times block ${}_rB_i$ is iterated while active, and let ${}_re_i$ be the number of graphs in the graph expression associated with block ${}_rB_i$. Finally, let ${}_rg_i$ be the maximum size of the graph during ${}_rB_i$, and assume that determining whether any source graph (or graph in a graph expression) occurs at a particular node in the graph costs at most k time units. Let ψ be the maximum cost of any rewrite.

4.4.1 Sequential Complexity

For each block ${}_rB_i$, every iteration ${}_rr_i$ rules must be applied to the graph of at most size ${}_rg_i$. Hence, the time complexity of locating all occurrences during ${}_rB_i$ is just $({}_rb_i)({}_rg_i)({}_rr_i + {}_re_i)k$; rewriting will require at most another $({}_rb_i)({}_rg_i)\psi$ time units. Since k, ${}_rr_i$, ${}_re_i$ and ψ will all be constant in a normal **eL** program, time complexity will be as follows:

Proposition 4.4.1 *Let r be a run of P, and let*

$$ {}_rT = \sum_{i=1}^{b} ({}_rb_i)({}_rg_i) $$

Then the maximum time complexity of P is just the maximum such ${}_rT$ over all runs:

$$ \textit{time of } P = O(\max_r({}_rT)) $$

4.4.2 Parallel Complexity

Assume that as well as the other parameters, we have p processors available for the execution of P. Whether we have a distributed memory or a shared memory environment, our graph will be divided among the processors according to the tree partition scheme developed in Chapter 3. With distributed memory there will be a cost c associated with communicating one graph node from any processor to any processor; with shared memory the same variable c can be used to represent the cost of ensuring exclusive access to a given node. Each processor is assumed to have a complete copy of the code.

With parallel execution, each iteration of each block is divided among the p processors. However, each iteration also will require the communication of nodes between processors so rules can be applied to nodes with neighbours on bordering processors. Load balancing requirements must also be factored in.

Due to the partitioning of the graph, each processor must communicate at most $O(\log({}_rg_i))$ nodes with other processors every iteration of the graph. As well, each processor, having checked the result of the current graph expression in its portion of the graph, must exchange this information with all other processors in order to ensure the correct block transition occurs on all processors; with any reasonable hardware this latter step (a *gather* followed by a *broadcast*) can be implemented in time $O(\log(p))$. In exchange, however, each processor only iterates over ${}_rg_i/p$ nodes. Let $\beta({}_rg_i, p)$ be the cost of rebalancing the tree partition scheme over the processors; then:

Proposition 4.4.2 *Let r be a parallel run of P, and let*

$${}_rT = \sum_{i=1}^{b}({}_rb_i)({}_rg_i/p + \log(p) + c\log({}_rg_i) + \beta({}_rg_i, p))$$

Then the maximum time complexity of P is just the maximum such ${}_rT$ over all runs:

$$\textit{time of } P = O(\max_r({}_rT))$$

The partitioning scheme developed in Chapter 3 requires that the tree partition scheme be divided as a post-order search of the tree, where child subtrees of each node are ordered by subtree weight. Since there is no control over which subtrees can grow,[3] if perfect load-balancing (modulo a small constant) is to be maintained for each iteration, then the entire graph may need to be moved each iteration.

Such a high upper bound on rebalancing requires drastic changes in the graph each iteration. For many algorithms, after an initial building phase, the graph does not change much between iterations. In these cases rebalancing cost is correspondingly less, and a parallel implementation of an **eL** program can be quite cost efficient: if there are at most n graph nodes, rebalancing is insignificant and $p \in O(n)$, then cost becomes $O(\log(n)\sum_i b_i)$.

4.5 Implementations

An **eL** $\rightarrow C$ translator has been written, which converts **eL** code to a C program which emulates the actions of the grammar. This program, called "el," is written in ANSI-C and itself can be compiled on almost any operating system with an ANSI-C compiler, Flex, and Bison (or Lex and Yacc). Code produced by "el" does not require Flex or Bison, just an ANSI-C compiler.

A more visually interesting implementation is the interpreter, "Tuna," which currently runs only under OS/2. This program presents the **eL** program as graphs, which can be interactively defined and manipulated. It also graphically shows the iterative changes in the axiom as the program progresses, and allows for a graphical manipulation of the control structure as well. A sample Tuna session is shown in Figure 4.1. Blocks are edited as units, dividing the window into two main sections; on the left are vertical lists of the defined rules, graphs and nodes, and on the right is an editing area where these same objects can be constructed and changed. Figure 4.1 actually shows a production "AA_to_BB" being edited, which rewrites an "A" node (dotted boundary) in the context of being connected to another "A" node (solid boundary) to a "B" node connected to an "A" node. The embedding function is illustrated by the dashed lines.

Computations are performed at the same time as rewrites, and may determine values of data on the target side as functions of the data found on the source side.

[3] In fact, the entire graph (or any portion of it) can be erased in one iteration by a suitable set of rules rewriting nodes to the empty graph. Growth is a little more constrained; the graph (or any portion) can increase in size by at most a constant multiple per iteration.

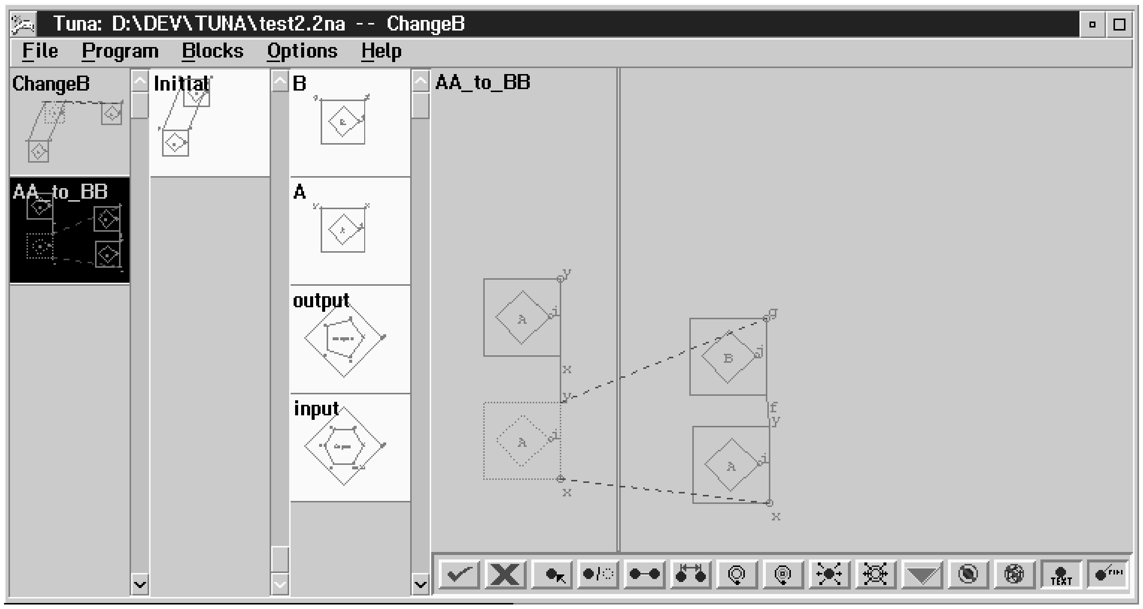
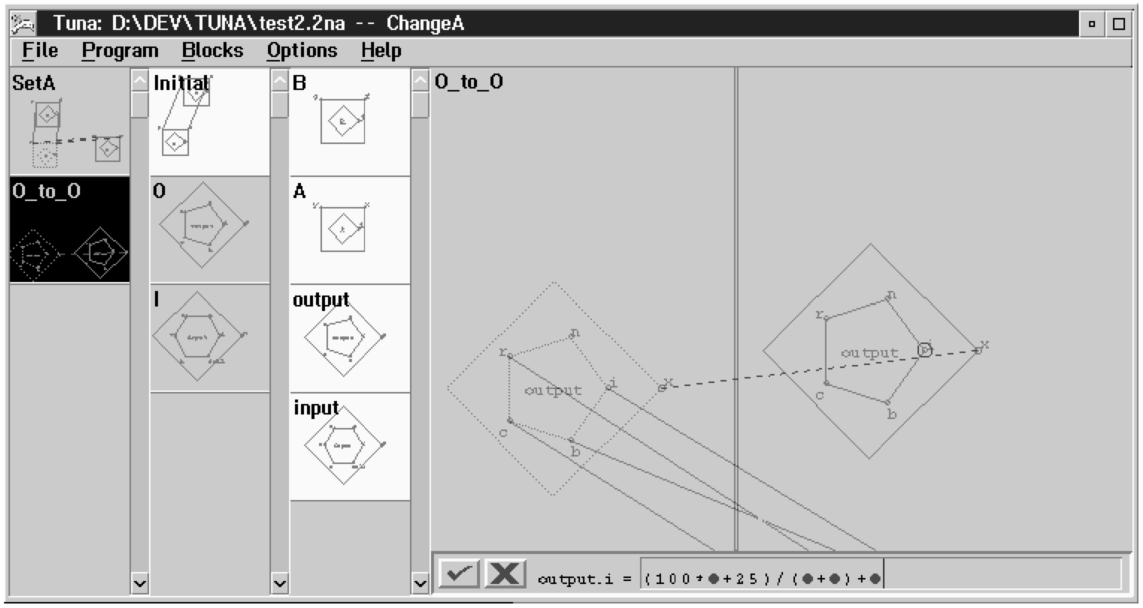


Figure 4.1: Editing a rule in Tuna.

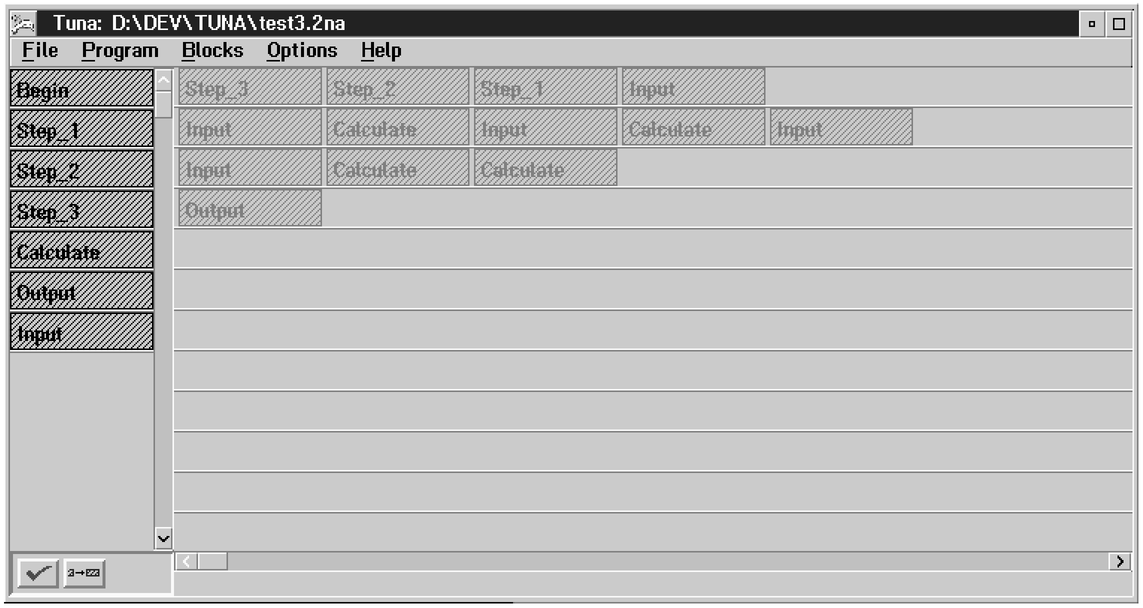
Since there may be several nodes with the same type, we need a method for unambiguously referring to a specific datum on either side of the rewrite. In the textual grammar this is accomplished by giving each node on both sides a distinct local name. To retain the visual flair of Tuna, though, it would be preferable to just refer to “that datum of that node over there” rather than typing in names for all our nodes. Thus, the references in a computation in Tuna are depicted by coloured lines connecting the “location” in the equation to the indicated datum. Figure 4.2 shows what a typical expression might look like (without the benefit of colour of course; the actual line is drawn in a bright purple to avoid confusing it with node links).

The control structure itself, the block stack, is also represented visually. Through another window one can define preblock and postblock lists for each block, as well as specify which block is defined to be the starting block. In Figure 4.3 we illustrate the process; each block is represented by a shaded rectangle, and two horizontal lists of other blocks. A copy of any block can then be dragged to any other preblock or postblock list, and similarly moved or deleted. This way all aspects of defining and running an **eL**-program in Tuna are visual.



output.i = (100*●+25)/(●+●)+●

Figure 4.2: Defining a computation within a rule in Tuna.



Begin	Step_3	Step_2	Step_1	Input	
Step_1	Input	Calculate	Input	Calculate	Input
Step_2	Input	Calculate	Calculate		
Step_3	Output				
Calculate					
Output					
Input					

Figure 4.3: Defining a postblock list in Tuna.

4.5.1 Partitionability

One of the advantages of programming in **eL** is the guarantee of partitionability provided by using the graph grammars of Chapter 3. To test this, an **eL** program was developed that generates inorder threaded binary search trees from a sequence of uniformly-distributed random numbers. The complete program is given as Appendix A. In Table 4.1, the resultant partitionings are described for an tree consisting of 2004 nodes and 4003 edges (the few extra nodes and edges are due to a root marker node, a begin node, and an end node attached to the tree, and a single unconnected done node). The partitionings induced by **eL** are compared with the best partitionings possible generated by a generic, heuristic graph partitioning program called "Jostle" [WCJE94]. The total number of edges crossing partitions and the minimum and maximum partition sizes (number of nodes in a partition) are illustrated for 2 partitions up to 1000 partitions. Note that the number of edges cut by the **eL** partitioning is generally smaller when the number of partitions is small, and is always more balanced; when the number of partitions gets large, Jostle seems to trade balance for cuts, something the **eL** partitioner does not currently do.

	Jostle			**eL**		
Partitions	*Cuts*	*Min*	*Max*	*Cuts*	*Min*	*Max*
2	28	1002	1002	12	1002	1002
3	29	668	668	23	668	668
4	58	500	502	29	501	501
5	115	399	402	50	400	401
6	68	333	335	53	334	334
7	75	286	287	85	286	287
8	98	249	251	74	250	251
9	106	222	224	79	222	223
10	114	200	201	92	200	201
20	189	97	102	161	100	101
30	259	63	73	243	66	67
100	409	18	22	655	20	21
500	1357	5	2	2081	4	5
1000	2005	1	3	3154	2	3

Table 4.1: Comparison of partitionings of a threaded binary tree by **eL** and Jostle.

4.6 Conclusions

The ability of programs in C to link pointers into an arbitrary topology is difficult to match. Unfortunately, this same algorithmic flexibility is the very source of problems in parallelization—if pointers can be directed arbitrarily then we will have an equally arbitrary problem disentangling the communication costs.

By incorporating the graph grammar model of Chapter 3 into a useable model of computation, we can ensure that any data structures we develop in the course of programming an application *can be* efficiently partitioned. By including the computation in the system as well, we provide an integrated parallel environment with specific and easily calculable guarantees of parallel efficiency. To this end we have given upper-bounds on the cost of both a sequential and a parallel implementation of the language. We have also developed two environments for our language: a traditional text-based grammar (el) which compiles **eL**-code to C, and a more intuitive visual interface (Tuna), combining **eL**-editor and interpreter. Both implement the same language of course, though the latter provides a much more intuitive user interface; by expressing computations, rewrites and even control flow visually, we eliminate much of the complexity and difficulty of working with fundamentally visual objects (graphs) in a textual manner. This reduces errors in designing graphs and transformations, and allows for the easy visualization of the actual graph as it is transformed, making debugging considerably easier too. The Tuna implementation of **eL** also emphasizes the qualitites of **eL** as a visual language—we have given a complete visual paradigm for computation, in an appealing and practical environment.

Chapter 5

Experimental Work: Grid Generation

So far we have illustrated the application of generating partitionable data structures only through small, toy examples. If the technique is to be in any way practical it needs to be demonstrated through a non-trivial program, preferably one which would seriously benefit from parallelization.

To this end we have chosen an adaptive version of the Control Volume Finite Element Method (CVFEM) of computational fluid dynamics, applied to irregular two-dimensional domains. This is a realistic problem, combining heavy computational requirements with necessarily dynamic data structures. Moreover, the solution algorithm implies a certain spatial organization of data, such that all computations require only "locally-available" information. This makes the problem particularly amenable to our approach, despite the relatively dense data connectivity ultimately demanded by the method.

First we motivate the development of our algorithm by giving a brief introduction to the adaptive CVFEM problem, illustrating its dynamic and irregular requirements. This is followed by a detailed explication of our algorithm, including the method employed for (and relative costs of) adaptivity. In Section 5.6, we prove some theoretical upper bounds on the cost of our algorithm. However, since our algorithm is highly dependent on the exact geometry of the input domain, experimental results can give a more accurate assessment of the algorithm's efficacy. In Section 5.7, therefore, we use a sequential implementation in C of the algorithm to provide experimental results for several different domains of varying complexity.

Finally, in Section 5.8 we illustrate how the algorithm can be implemented in the graph grammar formalism of Chapter 3, and show an upper bound on partitionability, and hence parallel performance of our algorithm. This is augmented with experimental results comparing the partitionings as produced by our method with existing heuristic approaches.

5.1 Introduction to CVFEM

Until recently, algorithms in computational physics have largely focussed on using regular geometric figures as the basis of domain discretization. For physical situations where the geometry is irregular, and more importantly where the physics requires a dynamically adaptive grid, such regular tesselations are obviously inadequate. One can fairly easily develop data structures that express irregular grid structures and, with rather more effort, develop satisfactory algorithms that create irregular grids consistent with the requirements of the numerical aproximations being used. Fairly sophisticated grid-generation algorithms have been developed for use with finite element methods; see for example the recent book by P. L. George [Geo91], the lecture notes by Weatherill [Wea90] or the review article by Bern and Eppstein [BE92].

There is a basic problem with all these schemes if one attempts to use them in conjunction with an adaptive algorithm. In physical applications, one adapts the grid in response to some local criterion (typically the gradient of one of the physical variables exceeds a prescribed bound), and thus one wants to refine (or coarsen) a grid locally. With existing Delaunay-based algorithms, though, even such local changes can cause the entire grid to be scrapped and recomputed—an expensive procedure for large grids. It is of interest therefore to develop an algorithm that allows incremental recomputation of the grid while maintaining the geometrical exigencies of the finite element method. In essence, it should be possible to exploit the inherent locality of the both the method and the phenomenon.

In 1988, Baker, Grosse and Rafferty [BGR88] described an algorithm for grid generation which satisfies the specific numerical constraints of grid generation for the finite element method (no obtuse angles), while still conforming to an arbitrary polygonal boundary. Here, we develop an incremental algorithm. We build on some of the geometric insights of their algorithm but we are forced to deal with a variety of new problems, including a significant explosion in the number and intricacy of the cases that need to be analyzed, and the need to maintain "balance" conditions on the tree structures used with an eye to future parallel implementation.

5.1.1 Physical Background

The finite element method computes an approximate solution to a differential equation in the following way. One breaks the region into small subregions called "elements." The solution to the differential equation is approximated by some standard function, depending on a few parameters, across the element. Often one uses linear or constant functions. One matches the solution in neighbouring elements across their boundaries and obtains in this way a set of algebraic equations that partially constrain the approximate solution. The iteration to the solution proceeds by using the approximate solution in the differential equation and successively refining the approximation.

This method imposes some basic requirements on the mesh. In order to ensure that the linear interpolation across the boundary of each element leaves the system consistent (*i.e.* the system is not overconstrained by imposing matching at three points on a linear function), the finite elements must be connected edge-wise, with no vertices located along any edge except at its endpoints. It also imposes the condition that the finite elements completely and disjointly cover the domain of interest. For an irregular domain approximated by a straight-line polygon, this essentially means that the domain must be covered by a mesh formed from triangular elements.

Unfortunately, not just any triangular mesh will do. For many fluid flow and heat conduction problems, the finite element method demands that the cosine of every angle in the triangulation be positive. Obtuse triangles, ones containing angles larger than $\pi/2$, can cause the generation of physically unrealistic results—such as an increase in temperature given a decrease in energy input—and are therefore undesirable in the mesh.

5.1.2 Computational Background

In practice one tries to construct a *Delaunay* triangulation. A Delaunay triangulation is a mesh where the circumscribing circle of each triangular element is free of any grid points in its interior. It also has the advantage that the minimum angle in the mesh is itself minimized over all possible triangulations of the same point set [Aur91], and this tends to reduce the number of obtuse triangles. A Delaunay triangulation, however, despite its otherwise very nice properties does not *guarantee* that the mesh will not contain obtuse triangles.

If the mesh is to be dynamically adapted, it must be able to efficiently increase and decrease the density of grid points in a specified area. Since this problem is being investigated with respect to parallel computation, particularly distributed-memory machines, these operations should be both efficient and as *local* as possible, to minimize any inherent communication costs. In other words, any changes made to the grid due to adaptivity should have a localized effect, not requiring the reconstruction of the entire grid. This is actually a requirement only for a coarse-grained parallelization strategy, where the grid itself is distributed among a relatively small number the processors. However, given that the finite element method requires only local computation at each grid point, and has a great deal of dependency between adjacent elements, surface-to-volume ratio arguments suggest that a coarse-grained approach is the most workable.

Unfortunately, the more popular Delaunay algorithms, like Watson's [Wat81], and Bowyer's [Bow81] are not certain to modify a triangulation with only bounded effect. The "edge flipping" technique of Watson, while guaranteed to terminate, is not guaranteed to restrict its locus of activity, and can spread throughout the entire mesh. Bowyer's algorithm has a similar problem: inserting a single point can (in the worst case) require a complete retriangulation of the domain. A dynamic adaptive scheme cannot afford such expensive operations, particularly when the grid is large.

5.1.3 The Baker, Grosse and Rafferty Algorithm

The algorithm of Baker *et al* [BGR88] works by overlaying a regular square grid of sufficient resolution over the domain. As long as each vertex of the domain coincides with a grid-point, and the grid is fine enough to ensure no more than one edge of the domain intersects a given square, it is possible to triangulate the domain. Each square can be triangulated separately with only acute triangles, including squares with an input edge intersecting them. Some extra effort is needed to deal with acute angles in the domain description—acute angles also constrain the minimum grid resolution. The complete details can be found, of course, in [BGR88].

At first glance this algorithm does not seem to have any special advantages when considering adaptivity; a fixed resolution is certainly not amenable to local changes. However, as they mention (but do not develop) in their discussion of the algorithm, quadtrees can be used to allow for some local variation in grid size. This permits guaranteed local grid refinements, and a significant overall reduction in number of triangles in the grid too.

5.2 Outline of Our Algorithm

To summarize, the mesh and/or mesh generation algorithm should have the following properties:

1. The mesh should be a triangulation, and all vertices must be only at the corners of the triangles.

2. All triangles should be non-obtuse.

3. The density of grid points within a specified region should be dynamically adjustable.

4. Grid modifications should be as localized as possible.

The grid generation algorithm presented here possesses the following features:

1. It generates a triangulation respecting arbitrary polygonal boundaries.

2. No triangle has an internal angle larger than $\pi/2$.

3. Starting from a "base" triangulation, the grid can be increased in density (within a specified region), and subsequently reduced as needed.

4. Modifications have a small and greatly-restricted non-local effect.

It should also be noted that since the grid generated has no obtuse angles, it is also automatically a Delaunay triangulation [Aur91], and so it inherits the well-established numerical properties thereof. Also note the one-way nature of this relationship: a non-obtuse triangulation is always a Delaunay triangulation, but a Delaunay triangulation is *not* automatically a non-obtuse one.

The algorithm consists of two main stages. First, the quadtree structure itself is generated—a square large enough to contain the entire domain is recursively decomposed into four smaller squares, until a base level is reached. This base level will depend on the geometry of the input domain, the choice of input vertices, and the necessity of being able to generate acute triangles. In order to ensure this base level can be reached, a number of conditions need to be guaranteed: a *vertex* condition (vertices lie on quadtree vertices), an *edge* condition (roughly, no more than one or two edges lie within any leaf quad), and a *balance* condition (leaf quads sharing an edge differ in depth by no more than 1).

Once the quadtree has been constructed to a base level, the leaves of the quadtree are triangulated. The vertices forming the corners of each quad are added to the domain, and (acute) triangles are generated respecting the individual boundaries of each quad. Once each quad is consistently and completely triangulated with acute triangles, so will be the entire domain. Completing this process for each of the possible quad leaves forms the bulk of the effort in implementing the algorithm.

Adaptivity applies after the initial triangulation is complete: it may be necessary to increase (and subsequently decrease) the density of the grid within a specified region, in accordance with the physical criteria discussed above. Our algorithm is such that incremental modifications dampen out exponentially; this is important for efficient adaptivity, as well as any parallel implementation.

5.3 Generating the Quadtree

The quadtree can be generated in one of two fashions; either depth-first or breadth-first. The former involves generating the branches of the quadtree one at a time, making each branch as deep as needed before moving on to the next branch. The latter generates the quadtree level-by-level, building all branches at an equal rate. While each version attempts to minimize the consumption of different resources (space and time, respectively), for reasons that will become clear shortly, the breadth-first approach is preferred.

Part of the algorithm for generating the quadtree requires a definition:

Definition 5.3.1 *A* two-edge case *is a pair of edges meeting at the corner of a quad that form an acute interior angle to the domain.*

We begin with an initial square large enough to contain the entire input polygon. We will then recursively divide a given square s, at depth d in the quadtree, into four squares of depth $d+1$ if any of the following properties hold.

Vertex Condition An input vertex of the polygon is contained within s (including the boundaries), and is not coincident with one of the four corners of s.

Edge Condition More than one input edge of the domain properly intersects s, and the half-planes (domain side) determined by edges of at least one pair of such edges intersect within s, and such a pair is not a two-edge case.

Balance Condition Any square at depth d' with $d' > d+1$ shares a side with s, and s does not contain only two-edge cases, nor is s entirely external to the domain.

5.3.1 Vertex Condition

The vertex condition is self-explanatory—it merely ensures that each input vertex lies on the corner of a grid square, which is the primary operating assumption for this algorithm. Note that since the quadtree recursively divides itself in two with respect to both the x and y axes, in order for the algorithm to terminate the input vertices must have some finite base-2 representation. The maximum base-2 precision will then be a lower bound on the maximum depth of the quadtree.

5.3.2 Edge Condition

The edge condition is necessary to keep the number of quad configurations that must be triangulated small. It is not possible to demand that each quad be triangulated acutely when one might be intersected by an arbitrary number of input edges, each of which must be taken into account. Hence, quads are generally restricted to just a single input edge. However, even if there are multiple input edges intersecting a quad, when the domains to be triangulated (as indicated by the interior half-plane of each input edge) do not intersect, there can be no conflict if each such domain is triangulated separately.

There is one exception to the edge condition. If two edges meet at a corner of the quad and form an acute interior angle (to the domain), the *two-edge case*, it will still be possible to triangulate the acute region, and so this situation need not cause the quad to be deepened.

Note that while it is possible to triangulate a quad containing more than two edges, it is also possible to demand that no more than two edges properly intersect any quad. Quads containing more than two edges can be forced to be subdivided, and the algorithm will still terminate. This follows because in a simple polygon (or even one with holes) only pairs of lines can intersect, and between any non-intersecting pair of lines is a minimum distance. Once quads are smaller than that minimum distance, the two non-intersecting lines cannot be in the same quad. Thus, at a cost of slightly more triangles in complex regions, the task of generating the tree can be made significantly easier.[1]

5.3.3 Balance Condition

The final condition, the balance condition, is the one that makes triangulating the quadtree leaves possible. By ensuring each quad is adjacent to another quad of no more than one level deeper, one can be certain to find an acute triangulation of each quad that places no new points on the boundary of the quad. If the balance condition is not enforced, a quad s may exist that is adjacent to arbitrarily deeper quads. The corners of each such adjacent deeper quad will then lie on the boundary of s, and so will need to be considered when triangulating s—and an arbitrary multiplicity of boundary points makes it quite difficult to generate an acute triangulation that does not change the boundary of the quad. By enforcing the balance condition it can be guaranteed that s will have to consider no more than a single such side point, and moreover that this side point, if it exists, will be precisely midway along the side. Such regular conditions do allow acute triangulations to be independently generated.

Thus, each quad can be triangulated individually and it is still certain that the triangulations within quads sharing a side will match up. Note that the condition has some caveats; if a quad lies entirely external to the domain then it will not

[1]Easier in that deciding when to subdivide a quad is simpler—otherwise, all edges would have to be checked each time to verify no two non-two edge cases have intersecting domain sides, which is an expensive operation.

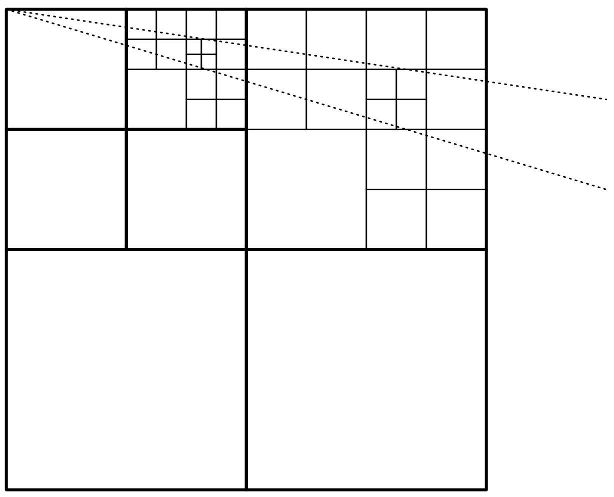


Figure 5.1: Enforcing the balance condition on the two-edge case causes infinite recursion.

need to be triangulated, and so the balance condition does not have to apply, saving some memory and effort. For reasons of correctness, though, the balance condition *cannot* apply to the two-edge case. To do so would create an infinite recursion (see Figure 5.1), as the balance requirements force the quad containing the two-edge case to be recursively subdivided, the result of which will be a smaller but identically unbalanced situation.

It is also the balance condition that makes a breadth-first approach more viable than depth-first. In order to ensure no neighbour of a quad is adjacent to a neighbour more than one level deeper, a depth-first approach would require multiple traversals—each time a branch is built, all its neighbours must be checked to ensure the balance condition is not violated. Retaining the balance condition during a level-by-level construction is somewhat simpler.

Once the above properties are not satisfied by any of the leaves of the quadtree, the tree construction terminates, and a case-by-case triangulation of the leaves ensues.

5.4 Triangulating the Quadtree Leaves

It is not *a priori* clear that once the quadtree has been constructed according to the above criteria, every resulting leaf/square of the tree has a non-obtuse triangulation. Indeed, most of the subtlety of the algorithm is in the cases, and there are many of them. Below, all possible cases are illustrated, as well as arguments about each triangle's acuteness. The following two concepts will be required.

Definition 5.4.1 *If some triangle (a, b, c) is such that a vertex c lies on or between lines drawn perpendicular to ab from a and from b, and c also lies on or outside the circle with diameter $|ab|$ centered midway along ab, then (a, b, c) is not obtuse, and c is in* acute position *with respect to edge ab. The circle/disk defined by two points a and b will be specified as $\mathcal{D}\mathrm{isk}(a, b)$ (see figure 5.2).*

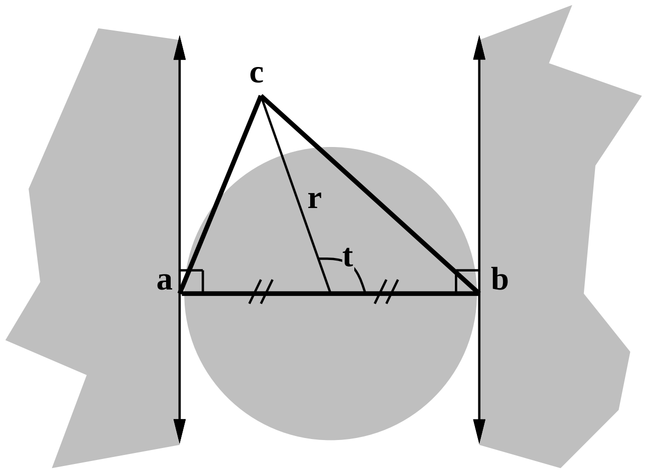


Figure 5.2: If c is not within the shaded region, (a, b, c) is acute.

Note 5.4.1 *An obtuse triangle can always be decomposed into two right-angle triangles by drawing a line intersecting the obtuse vertex which is perpendicular to the opposing edge (see figure 5.3).*

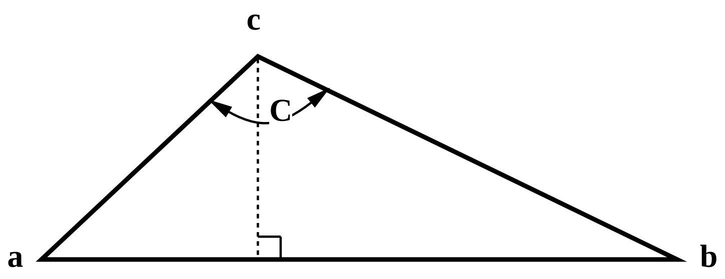


Figure 5.3: Splitting an obtuse triangle into 2 right-angle triangles.

Each one of our leaf-squares created by the above quadtree construction falls into one of the following mutually-exclusive and exhaustive categories:

1. Exterior; the quad is entirely outside the domain.
2. Interior; the quad is entirely inside the domain.
3. Boundary; the quad contains some portion of the domain boundary.

The last case, boundary quads, includes a large number of subcases. It itself is then subdivided into cases based on the manner in which a quad can intersect the boundary:

1. One-Edge properly intersects the quad:
   (a) The edge intersects adjacent quad sides
   (b) The edge intersects opposing quad sides
2. Two-edges intersect the quad:
   (a) Domain side of edges do not intersect
   (b) The two-edge case

For each of the above cases, the balance condition forces the consideration of the possibility that each of the four quad leaf sides may or may not have a vertex at its midpoint. If a quad shares a side with another quad of depth one greater, then there will be a vertex midway along the same side corresponding to a corner of the smaller square. Each side that the square shares with another square that is at the same or higher depth will not have such a midpoint. Fortunately, this same property also ensures that there is no possibility of there being any other additional vertices on the sides of a quad. Thus, there are at most 16 possible configurations of midpoints for each subcase. As it will turn out, there are far fewer cases than this would indicate—symmetries, as well as not being concerned with the exterior domain allow for many combinations to be ignored.

Naturally, quads that are entirely exterior to the domain of interest do not need to be triangulated.

5.4.1 Interior Quads

When no input edges intersect, the task of triangulating the square is much simplified. Beyond the four corner vertices, there are only four possible other vertices (a midpoint on each side) to consider. Symmetries reduce the number of cases to a mere six (see figure 5.4). In each case, the triangles are trivially not obtuse.

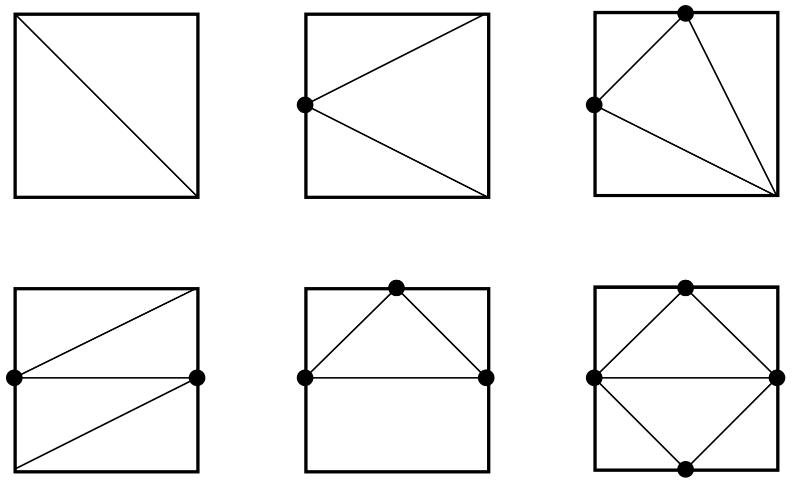


Figure 5.4: Possible triangulations of interior quads.

5.4.2 Case 1a: An Input Edge Intersects Adjacent Sides

If an input edge enters from one side, s, of a square, and exits from one of the two sides that share a corner with s, it can be classified into one of four cases. The edge enters either above or below the midpoint on s, and exits similarly (though to keep things straight, instead of 'above' or 'below' the two halves of the exit side are called 'left' and 'right'). Of course there are symmetries; the edge can be assumed to enter from the left and exit on the bottom, and then any above-left configuration is a counter-clockwise rotation by $\pi/2$ of a below-right configuration. Thus, there are actually only 3 subcategories to consider: above-right, below-left and below-right.

A note about numbering within the diagrams: the three cases, below-left, below-right and above-right are indicated by the prefixes "BL," "BR" and "AR" respectively. This is followed by a number, 0 to 4, indicating the number of midpoints, then a decimal point and then a number indexing the possible combinations of midpoints. When the other "side" of the edge is being triangulated, the cases follow the same pattern, with the prefix being followed by a prime (*e.g.,* "BL$'$-2.1" instead of "BL-2.1").

Individual diagrams also have a consistent labelling scheme. Corners of the quad are labelled c_1 to c_4, going clockwise starting from the lower-left corner. The edge typically enters from the left at point e_1 and exits to the right at point e_2. Points added to the interior are labelled a, b, p or q, with the intention that a always lies at the center of the quad, b is usually the point corresponding to the third corner of a right-angle triangle (interior to the domain) made with e_1 and e_2, and p and q are individually placed. Midpoints along the sides of the quad are labelled m_1 to m_4, clockwise beginning with the left side (see figure 5.5). Finally, the horizontal line bisecting the quad is referred to as the *horizontal bisector,* and similarly the vertical line bisecting the quad is referred to as the *vertical bisector.* By *center* we mean the center of the quad.

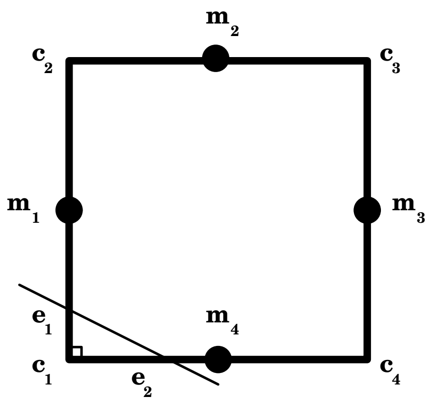


Figure 5.5: Labelling for individual quad diagrams.

Below Left Intersections

When the input edge intersects the square below the midpoint on the left side, and left of the midpoint on the bottom, there are either 16 cases or one case. If the domain of interest lies to one "side" of the directed input edge then there are 4 midpoints which may or may not be present, for a total of 16 possibilities. If the domain lies to the other "side," then there is only one trivial case (a below-left intersection cuts off a right-angle triangle corner, which can contain no midpoints). Here the 16 non-trivial cases are presented.

Note that for this case the input edge can be constrained such that $|c_1 e_1| \leq |c_1 e_2|$. If this is not true, the quad can be transformed by a vertical reflection followed by a counter-clockwise rotation by $\pi/2$, and then this constraint will hold.

The discussions that follow frequently make reference to the following simple results. Though sometimes intricate, none of these proofs require math more sophisticated than high school trigonometry.

Lemma 5.4.1 *Given a rectangle of size $w \times z$, let a be the upper-right corner, and let the origin, O, be the lower-left corner (see figure 5.6). If a line is drawn from a to some point e_2 on the bottom side, and a perpendicular to ae_2 is projected from e_2, intersecting the left side at e_1, then the vertical distance of e_1 from O is $y = x(w-x)/z$, and this value is maximal for $0 \le x \le w$ at $x = w/2$, whereupon $y = w^2/(4z)$.*

Proof: Note that $\theta = \angle e_2am$ and that $\theta = \angle e_1e_2O$. Thus, $\tan(\theta) = (w-x)/z$.

We can compute the value of y then as $y = x\tan(\theta) = x(w-x)/z$.

To establish that this is maximal for $0 \le x \le w$, we simply take the derivative of the function for y with respect to x and note that y is 0 at $x = 0$ and $x = w$. □

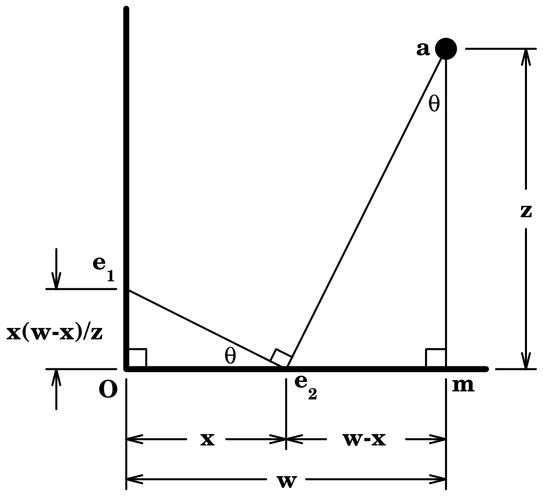


Figure 5.6: Height y is defined by $y = x(w-x)/z$

Corollary 5.4.1 *If $\angle ae_2e_1$ is not acute, then $y \le x(w-x)/z$, and if angle $\angle ae_2e_1$ is not obtuse, $y \ge x(w-x)/z$.*

Proof: This follows trivially from lemma 5.4.1. □

BL-0 See figure 5.7. Only two triangles are not right angle, (c_2, c_3, b) and (c_3, c_4, b). However, the construction of b forces b to lie in the lower left quarter of the quad. According to definition 5.2, then, b must be in acute position with respect to c_2c_3 and c_3c_4, and so both triangles must be acute.

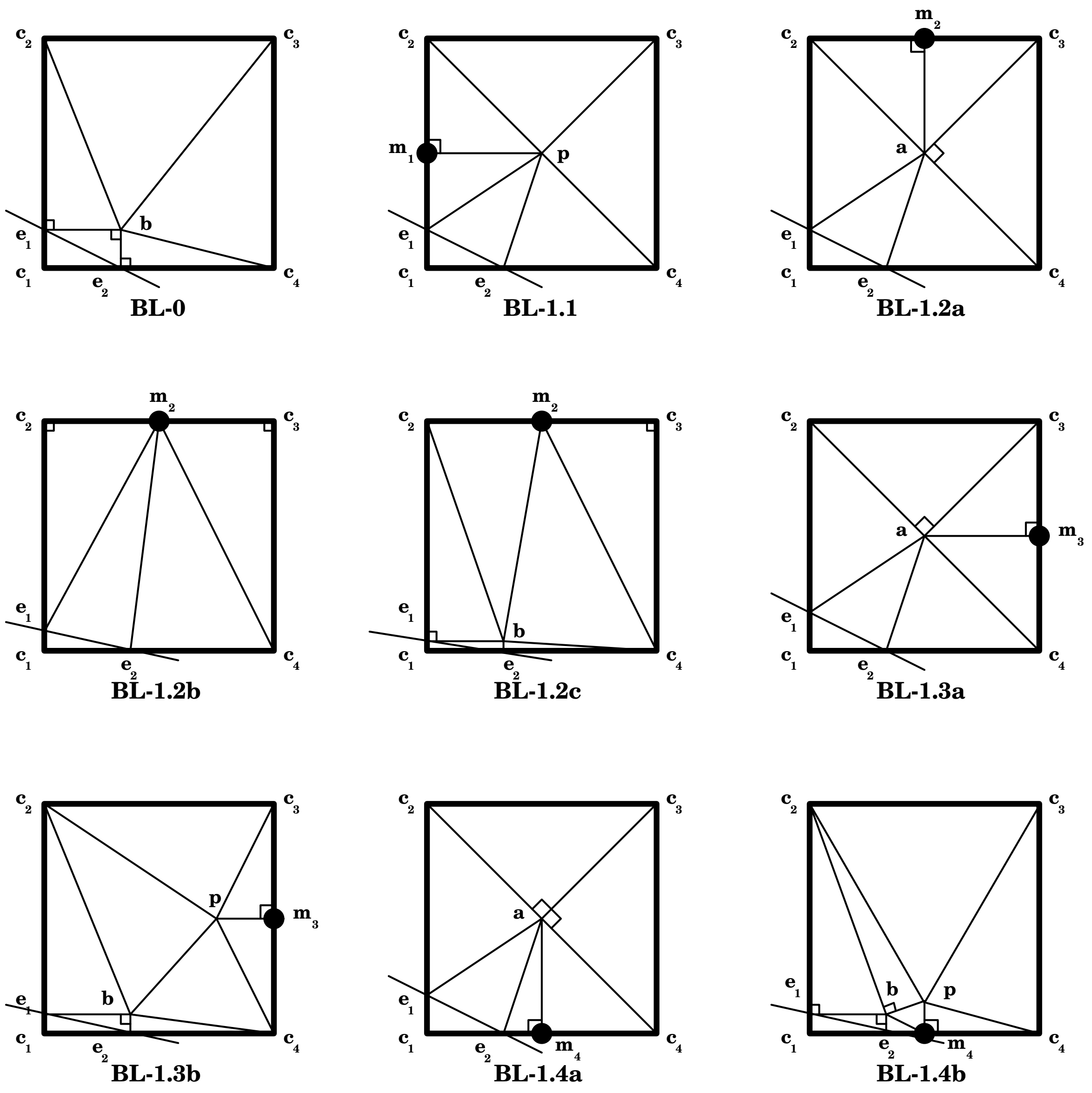


Figure 5.7: Subcases for adjacent below-left intersections with zero or one midpoint.

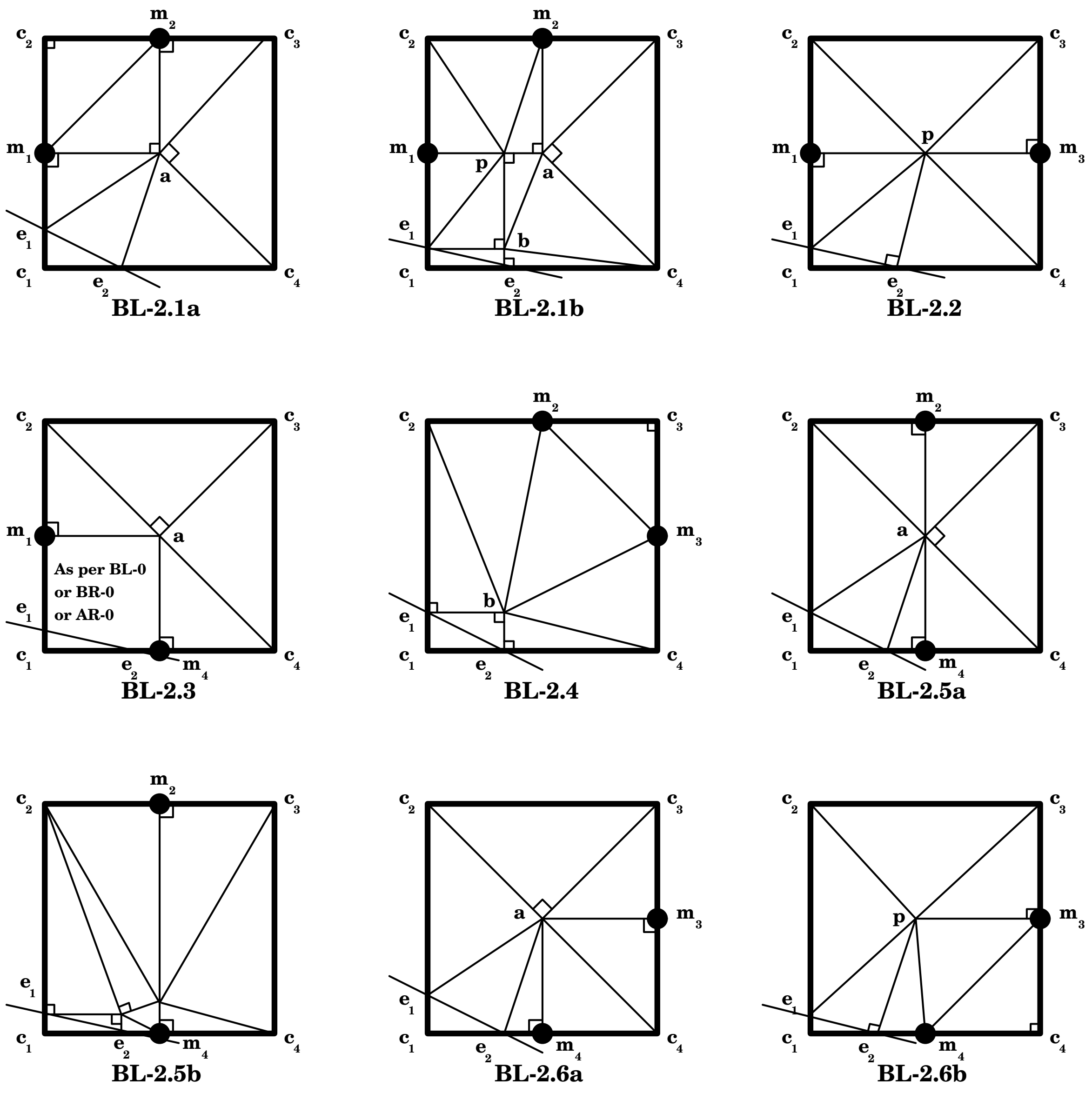
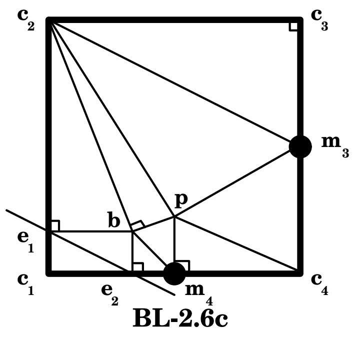
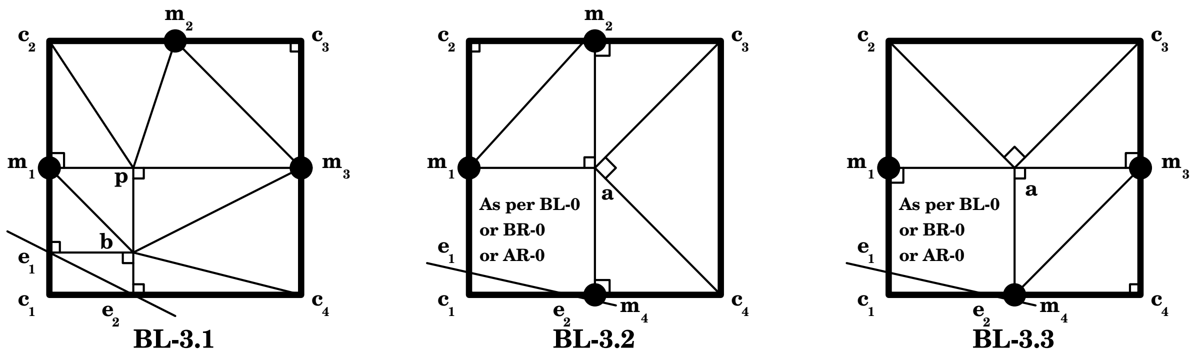


Figure 5.8: Subcases for adjacent below-left intersections with two midpoints.





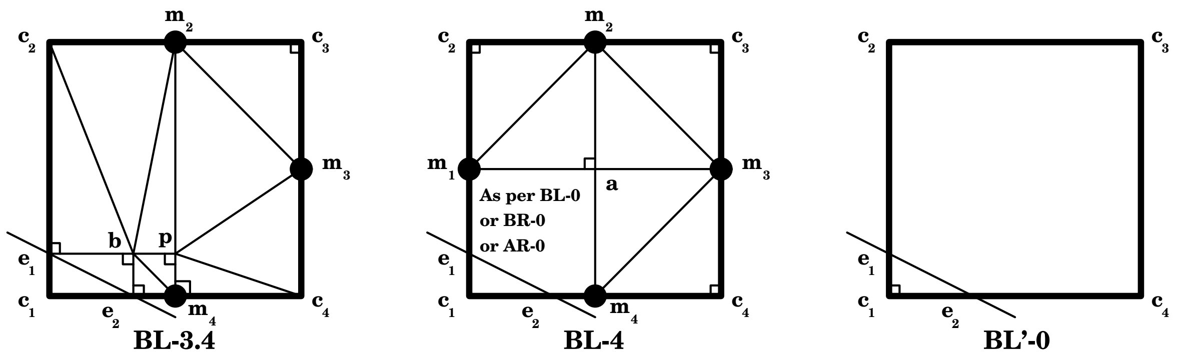


Figure 5.9: Remaining BL and all BL' subcases.

BL-1.1 See figure 5.7. Point p is located as the point closest to the center along the horizontal bisector between m_1 and the center, such that $\angle e_1e_2p$ is no larger than $\pi/2$. Note that p will lie horizontally somewhere between e_2 and the center. Because $|e_1c_1|$ is no bigger than $|c_1e_2|$, a perpendicular to e_1e_2 extended from e_1 will always intersect left of the center and left of the intersection of a similar perpendicular extended from e_2, and so $\angle e_2e_1p$ will be acute as long as $|e_1e_2| > 0$. Since p is to the right of e_2, the angle $\angle e_1pe_2$ is contained in the $\pi/2$-angle formed between the horizontal bisector and a vertical passing through p, and hence is certainly not obtuse. The other triangles are simple to establish: because point p is on the horizontal bisector, p is necessarily in acute position to e_2c_4 and c_2c_3. Because p is left of the center, p is also in acute position to c_3c_4.

BL-1.2 See figure 5.7. If $\angle ae_2e_1$ is not obtuse, then this configuration can be triangulated as per subcase BL-1.2a. In such a case point a, being the center of the quad, is trivially in acute position relative to all of e_2c_4, c_3c_4 and e_1c_2. Angle $\angle e_2e_1a$ is acute for the same reasons as in BL-1.1, and angle $\angle e_1ae_2$ is acute because e_1 and e_2 lie in the lower-left quadrant.

Alternatively, if $\angle ae_2e_1$ is obtuse, then if $\angle m_2e_2e_1$ is acute and e_2 is horizontally at least 1/4 of the way along the bottom side, the quad can be triangulated as per BL-1.2b. Point m_2 is then guaranteed to be in acute position relative to e_2c_4. Because of the constraints on the vertical coordinate of e_1 imposed by corollary 5.4.1, the angle $\angle c_1e_1e_2$ varies between approximately 1.1 and $\pi/2$ radians within the allowed range of e_2. Angle $\angle m_2e_1c_2$ only varies between 0.46 and 0.52 radians, but in any case the two angles sum to more than $\pi/2$, leaving $\angle e_2e_1m_2$ less than $\pi/2$. Finally, because e_1 and e_2 lie in the lower-left quadrant, $\angle e_1m_2e_2$ is acute.

Finally, if $\angle m_2e_2e_1$ is also obtuse or e_2 is horizontally within the leftmost 1/4 of the bottom side, point b can be introduced and the quad triangulated as in BL-1.2c. Point b is then in acute position relative to edge m_2c_4: b is certainly within perpendiculars to m_2c_4 extended from m_2 and c_4, and (within its horizontal bounds) b necessarily lies below a tangent to $\mathcal{D}isk(m_2, c_4)$ at halfway along the bottom side ($\mathcal{D}isk(m_2, c_4)$ intersects the bottom side at the bottom right corner, and midway along the bottom side).

BL-1.3 See figure 5.7. If $\angle ae_2e_1$ is acute, the triangulation is similar to BL-1.2a, as shown in subcase BL-1.3a.

If $\angle ae_2e_1$ is obtuse, triangulation follows BL-1.3b. Here, point p is located along the horizontal bisector to the right of the center. Thus, p is in acute position relative to c_2c_3. In order to ensure that triangle (b, c_2, p) is acute, p is placed outside $\mathcal{D}isk(b, c_2)$, but inside the right angle formed by extending a perpendicular to bc_2 out from b. That this can always be done is established by the following lemma.

Lemma 5.4.2 *The perimeter of* $\mathcal{D}\text{isk}(b, c_2)$ *intersects the horizontal bisector of the square (strictly) to the left of the right side.*

Proof: Because point b must be located in the region designated by corollary 5.4.1, the line segment c_2b reaches a maximum length in the degenerate case wherein b is coincident with the bottom midpoint. Thus, $|c_2b| \leq \sqrt{5}$ (assuming the square is 2×2, with origin at the lower-left corner). As well, the furthest to the right that the midpoint along c_2b can be located occurs in the same degenerate situation, where by similar triangles the center of the circle can be determined to be at coordinates $(1/2, 1)$. Hence, the intersection of the circle and the horizontal bisector of the square can be no further to the right than $\sqrt{5}/2 + 1/2$, which must be to the left of the right side of the 2×2 square.

Hence, by lemma 5.4.2 and the simple observation that a perpendicular to c_2b from b never intersects the horizontal bisector inside the square, it is certain p can be placed in acute position relative to both c_2c_3 and c_2b. The only triangle in doubt is (p, b, c_4). Corollary 5.4.1 implies that the height of b is bounded by $x(1-x)$ (within our 2×2 square), and thereby b always lies below the diagonals c_2c_4 and c_1c_3. Thus, b must lie within perpendiculars to pc_4 from p and from c_4. Furthermore, $\mathcal{D}isk(p, c_4)$ does not intersect the curve $x(1-x)$ within $0 < x < 1$, so b is in acute position to pc_4.

BL-1.4 See figure 5.7. Once again, if $\angle ae_2e_1$ is acute, the triangulation is similar to BL-1.2a, as shown in subcase BL-1.4a.

If $\angle ae_2e_1$ is obtuse, BL-1.4b is used instead. Point p here is located as the intersection of a perpendicular to c_2b at b and the vertical bisector; point p must be in acute position relative to c_3c_4. Since the location of b is constrained as per corollary 5.4.1, point p certainly can never be higher than the horizontal bisector, and so p is in acute position to c_2c_3. By construction, b must lie below p and above the bottom side, so angles $\angle pm_4b$ and $\angle bpm_4$ must be acute . As well, the placement of b implies that $\angle m_4bc_2$ must be no larger than π, and $\pi/2$ of that angle is "used up" by the right-angle $\angle pbc_2$. Hence $\angle m_4bp$ must be smaller than $\pi/2$.

BL-2.1 See figure 5.8. Again, if $\angle ae_2e_1$ is acute, the triangulation is similar to BL-1.2a, as shown in subcase BL-2.1a.

Otherwise subcase BL-2.1b applies; point p is located as the intersection of the horizontal bisector and a vertical extended up from b. Since e_2 is by definition left of the center, so will be p. The only triangle that is not a right-angle triangle is then (b, a, c_4). But since b is constrained as per corollary 5.4.1, b must be in acute position to ac_4: b must lie within perpendiculars to ac_4 extended out from a and c_4, and the center of $\mathcal{D}isk(a, c_4)$ maintains a distance of at least $1/\sqrt{2}$ (which is the same as its radius) away from any possible position of b.

BL-2.2 See figure 5.8. This situation is very similar to BL-1.1.

BL-2.3 See figure 5.8. This triangulation trivially follows.

BL-2.4 See figure 5.8. Since point b here is constrained to lie in the lower-left quadrant of the quad, b must be in acute position to m_3m_4, m_3c_4, and c_2m_3.

BL-2.5 See figure 5.8. Once more, if $\angle ae_2e_1$ is acute, the triangulation is similar to BL-1.2a, as shown in subcase BL-2.5a.

Alternatively, the triangulation is a trivial modification of the pattern shown in BL-1.4b, which is illustrated in BL-2.5b.

BL-2.6 See figure 5.8. Again, if $\angle e_1e_2a$ is acute, the triangulation is similar to BL-1.2a, as shown in subcase BL-2.6a.

Otherwise, assume a 2×2 square as in BL-2.6b; point e_1 is then constrained by corollary 5.4.1 to be such that if e_2 has x-coordinate x, then e_1 has y-coordinate no bigger than $x(1 - x)$. In particular, if $0.41877 < x < 0.58123$, then $0.25 \geq y \geq 0.2434$. Because of this, $\mathcal{D}isk(c_2, e_1)$ can have radius no bigger than $1 - 0.1217$, and hence intersects the horizontal bisector strictly to the left of the point $(0.8783, 1)$. Let p be placed as the intersection of a perpendicular to e_1e_2 extended from e_2 and the horizontal bisector, and let v be the horizontal difference between p and the center: $v = 1 - p.x$. If we show that within the limited range allowed for x (and hence y) v must remain outside $\mathcal{D}isk(c_2, e_1)$, then triangle (e_1, c_2, p) is surely acute.

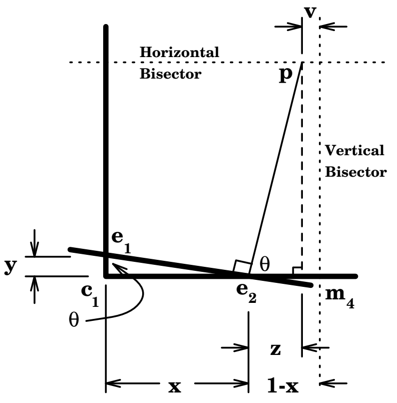


Figure 5.10: Subcase BL-2.6b detail.

Given x and y, we can calculate v as follows (see figure 5.10). Let θ be the angle $\angle c_1e_1e_2$; then $\tan(\theta) = x/y$. Let z be the length of the bottom side of a right-angled triangle formed from e_2, p and the intersection of a vertical through p and the bottom side, and note that $\theta = \angle(x+z, 0)e_2p$. From this we can conclude that $z = 1/\tan(\theta) = y/x$, and thus $v = 1 - x - z = 1 - x - y/x$. Distance v increases as y decreases, so for a y lower-bounded by 0.2343, v is maximal when $x = \sqrt{0.2343}$. Thus, v is maximal at about $v_{\max} \approx 0.013288$, which is well outside of $\mathcal{D}isk(c_2, e_1)$.

Also note that because p lies horizontally between e_2 and the center, p is in acute position to both c_2c_3 and e_2m_4, and m_4 is in acute position to pm_3.

Alternatively, if $y < 0.2343$ (which is certainly true if $x \leq 0.41877$ or $x \geq 0.58123$), then triangulation is as per BL-2.6c (see figure 5.9). Here p is located as the intersection of a perpendicular to c_2b at b and the vertical bisector. Note that because of corollary 5.4.1, $\angle m_4bc_2 < \pi$, and so if $\angle pbc_2 = \pi/2$ then $\angle m_4bp < \pi/2$. Point b is therefore in acute position to pm_4. Point p will also be in acute position to m_3c_4 if p does not rise vertically above the horizontal bisector. However, in order for $\angle m_3pc_2$ to be acute p must satisfy the more stringent requirement that p lie outside $\mathcal{D}isk(c_2, m_3)$.

We can bound the height of p. Let $\theta = \angle e_1c_2b$, and let q be the point $(1, y)$ (the intersection of a horizontal passing through e_1 and the vertical bisector). Let z be the y-coordinate of p. Because $\theta = \angle qbp$ and $\tan(\theta) = x/(2-y)$, we can determine $z = y + (1-x)x/(2-y)$, and because z increases as y increases, we can bound the size of z by considering only a maximal y for a given x; that is,

$y = x(1-x)$. By assumption, y is upper-bounded by 0.2343, and so z is upper-bounded by approximately 0.37. $\mathcal{D}isk(c_2, m_3)$ is fixed, and intersects the vertical bisector at y-coordinate $1.5 - \sqrt{5}/2 \approx 0.382$, hence p is outside $\mathcal{D}isk(c_2, m_3)$. Because p is of course lower-bounded by m_4, p must be in acute position to c_2m_3 and to m_3c_4.

BL-3.1 See figure 5.9. Point p is located as the intersection of a vertical extended up from e_2 and the horizontal bisector. Thus, point p is certain to be left of the center and in acute position relative to c_2m_2, and similarly m_2 will be in acute position to pm_3. Because b will be located left of the center, b will also be in acute position relative to m_3c_4. All the rest are right-angle.

BL-3.2 See figure 5.9. These results are trivial.

BL-3.3 See figure 5.9. These results are trivial.

BL-3.4 See figure 5.9. Here, point p is located as the intersection of a horizontal extended out from e_1 and the vertical bisector. Thus, p is in acute position to m_3c_4, and m_3 is in acute position relative to m_2p. Point b is constrained to lie in the lower left quadrant of the quad, and so point b is in acute position relative to c_2m_2.

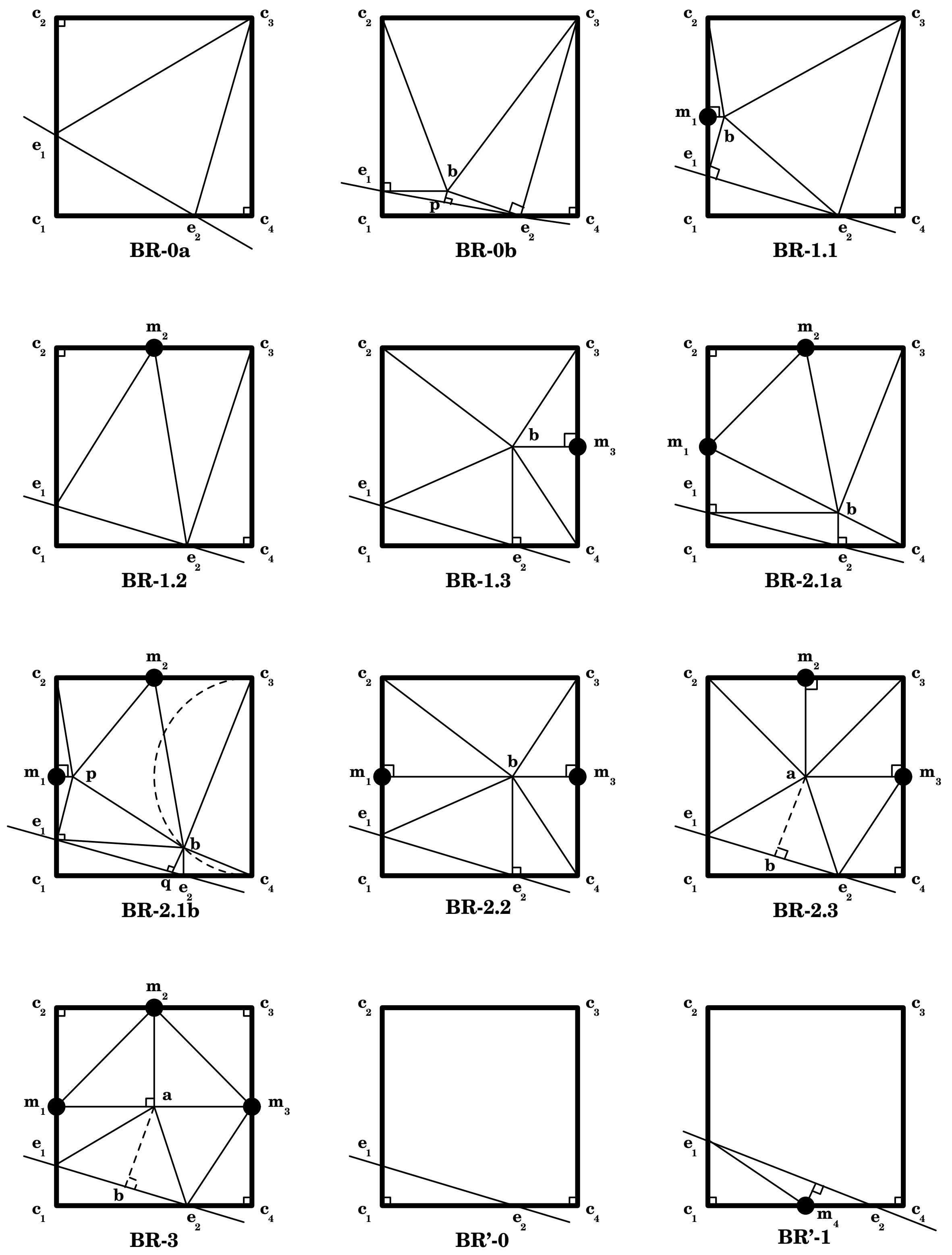
BL-4 See figure 5.9. These results are trivial.

BL-0' See figure 5.9. These results are trivial.

Below Right Intersections

When the input edge enters below the midpoint on the left, and right of the bottom midpoint, the triangulation follows one of the patterns below. There are only 10 subcases in total here, though—when triangulated one side of the input edge there are only three midpoints that may or may not be present. When triangulating the other side, only a single midpoint could exist.

Note that within these cases, the point b within the quad diagrams has no particular significance (*i.e.,* it is not always the intersection of a vertical from e_2 and a horizontal from e_1).



c_2 c_3 e_1 c_1 e_2 c_4
BR-0a
c_2 c_3 e_1 b p c_1 e_2 c_4
BR-0b
c_2 c_3 m_1 b e_1 c_1 e_2 c_4
BR-1.1
m_2 c_2 c_3 e_1 c_1 e_2 c_4
BR-1.2
c_2 c_3 b m_3 e_1 c_1 e_2 c_4
BR-1.3
m_2 c_2 c_3 m_1 e_1 b c_1 e_2 c_4
BR-2.1a
m_2 c_2 c_3 m_1 p e_1 b c_1 q e_2 c_4
BR-2.1b
c_2 c_3 b m_1 m_3 e_1 c_1 e_2 c_4
BR-2.2
m_2 c_2 c_3 a m_3 e_1 b c_1 e_2 c_4
BR-2.3
m_2 c_2 c_3 m_1 a m_3 e_1 b c_1 e_2 c_4
BR-3
c_2 c_3 e_1 c_1 e_2 c_4
BR'-0
c_2 c_3 e_1 c_1 m_4 e_2 c_4
BR'-1

Figure 5.11: Subcases for adjacent below-right intersections.

BR-0 See figure 5.11. If $\angle c_3e_2e_1$ is acute, triangulation is performed as in BR-0a. Point e_1 must lie outside $\mathcal{D}isk(e_2, c_3)$, and so is in acute position to e_2c_3.

If this is not the case, point b is located as the intersection of a perpendicular to c_3e_2 extended from e_2 and a horizontal extended from e_1. Since e_1 is certainly below the horizontal bisector, b will be in acute position to c_2c_3. Point p is then located as per note 5.4.1.

BR-1.1 See figure 5.11. Point b is located as the intersection of a perpendicular to e_1e_2 extended out from e_1 and the horizontal bisector. Thus, point b will always be in acute position relative to edge c_2c_3. By lemma 5.4.1, $|m_1b|$ is maximal when e_1 is midway between c_1 and m_1; assuming a 2×2 square, $|m_1b| \leq 1/(4x)$. Within the allowed range of x, this maximizes at $x = 1$, whereupon $|m_1b| = 1/4$. $\mathcal{D}isk(c_3, e_2)$ has radius no larger than $\sqrt{5}/2$, and its center has an x-coordinate no smaller than 1.5, so it intersects the horizontal bisector no closer to the left than about 0.382. Since $\angle e_2bc_3$ decreases as b moves to the left or e_2 moves to the right, and even when b is maximally to the right and e_2 maximally to the left $\angle e_2bc_3$ is acute, triangle (e_2, b, c_3) must be acute.

BR-1.2 See figure 5.11. Point e_2 is certainly in acute position to m_2c_3, and by the constraints on the positions of e_1 and e_2 (*i.e.,* being a below-right case) e_1 is in acute position to m_2e_2.

BR-1.3 See figure 5.11. Point b is located as the intersection of a vertical extended up from e_2, and the horizontal bisector. Since e_2 is certainly right of the vertical bisector and b is above e_1, b must be in acute position relative to e_1c_2, and since b is along the horizontal bisector, it is also in acute position to c_2c_3. Point e_1 must lie vertically between c_1 and m_3 so e_1 is in acute position to e_2b.

BR-2.1 See figure 5.11. Point b is located here as with the bottom left (BL) cases, as the intersection of a vertical extended from e_2 and a horizontal extended from e_1. If b lies outside $\mathcal{D}isk(c_3, c_4)$, then b is in acute position relative to c_3c_4, and triangulation is as per BR-2.1a. Because e_1 is never higher than the center, b is also in acute position relative to m_2c_3. Finally, because b must lie in the lower right quadrant, b must be in acute position to m_1m_2 as well.

If b would lie within $\mathcal{D}isk(c_3, c_4)$, then triangulation proceeds as in BR-2.1b. Let b lie on the intersection of a vertical up from e_2 and the perimeter of $\mathcal{D}isk(c_3, c_4)$, and position p as the intersection of a perpendicular to be_1 extended from e_1 and the horizontal bisector. Once again assuming a 2×2 square, we can express the equation describing $\mathcal{D}isk(c_3, c_4)$ as $y = 1 - \sqrt{-x^2 + 4x - 3}$.

Let x be the x-coordinate of e_2 and y be the y-coordinate of b. Let z be the y-coordinate of e_1, w be the x-coordinate of p, and let $\theta = \angle pe_1m_1$. Thus, it must also be that $\theta = \angle e_1b(0, y)$ and $\tan(\theta) = (z - y)/x$, and we can compute w from $w = (1 - z)\tan(\theta) = (1 - z)(z - y)/x$. In order to upper-bound w we note that if we assume a fixed x (and hence y), by lemma 5.4.1 w is maximized by placing e_1 precisely midway between y and m_1; $z = (y + 1)/2$. Thus, it must be that:

$$w \leq \frac{(1 - \frac{y+1}{2})(\frac{y+1}{2} - y}{x} = \frac{(\frac{1-y}{2})^2}{x} = \frac{-x^2 + 4x - 3}{4x}$$

This function is maximal within our allowed range of x values when $x = \sqrt{3}$, at which point $w_{\max} = 1 - \sqrt{3}/2$.

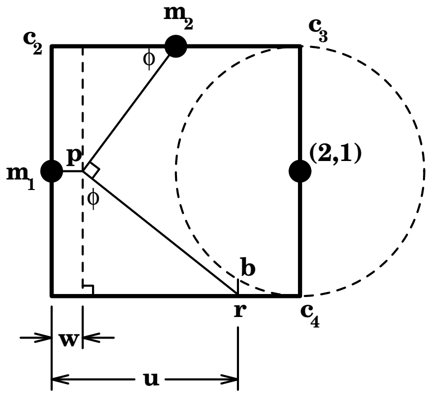


Figure 5.12: A necessarily acute angle in subcase BR-2.1b.

Now, consider a perpendicular to m_2p extended from p. It intersects the bottom side of the quad (the $y = 0$ line) at some point r (see figure 5.12). If we establish that pr never intersects $\mathcal{D}isk(c_3, c_4)$, then $\angle bpm_2$ must always be contained in $\angle rpm_2 = \pi/2$, and hence must be acute.

The position of r is dependent on p; as p moves to the right, r moves to the right. Thus, if pr does not intersect $\mathcal{D}isk(c_3, c_4)$ when w is maximal, neither does $p'r'$ for any $0 \leq p.x' < w$. We can calculate the x-coordinate of r by noting that if we let $\phi = \angle(w, 2)m_2p$, then $\tan(\phi) = 1/(1 - w)$, and $\phi = \angle(w, 0)pr$. Thus, if u is the x-coordinate of r, then $u = 1/(1-w) + w$. As mentioned, u is maximal when w is maximal, so:

$$u_{\max} = \frac{1}{1 - w_{\max}} + w_{\max} = \frac{1}{1-\sqrt{3}/2} + 1 - \sqrt{3}/2 = 1 + \frac{1}{2\sqrt{3}} \approx 1.288$$

The distance of the center of $\mathcal{D}isk(c_3, c_4)$ (which is $(2,1)$) from the line pr can then be determined to be $\sqrt{7}/2$, which is certainly larger than the unit radius of $\mathcal{D}isk(c_3, c_4)$.

Point b can never be located outside of the lower-right quadrant of the square and p must lie along the horizontal bisector left of the center, and so angles $\angle pm_2b$ and $\angle m_2bp$ must both be acute. For the same reasons, p is in acute position to c_2m_2, b is in acute position to m_2c_3, and by construction b is in acute position to c_3c_4. The angle $\angle e_1be_2$ is obtuse by assumption, so once we locate point q as per note 5.4.1 the remaining triangles are all right-angled.

BR-2.2 See figure 5.11. This case is a trivial modification of BR-1.3.

BR-2.3 See figure 5.11. Point a, being the center of the quad, is trivially in acute position to e_2m_3 and e_1c_2. Angle $\angle ae_2e_1$ cannot be obtuse since e_2 is right of a (and hence $\angle c_4e_2a$ is obtuse), and $\angle e_2e_1a$ cannot be obtuse since both $\angle ae_1c_2$ and $\angle c_1e_1e_2$ must both be at least $\pi/2$. The final angle, $\angle e_1ae_2$ may or may not be obtuse—if it is, then a point b is introduced as per note 5.4.1.

BR-3 See figure 5.11. This case is virtually identical to BR-2.3.

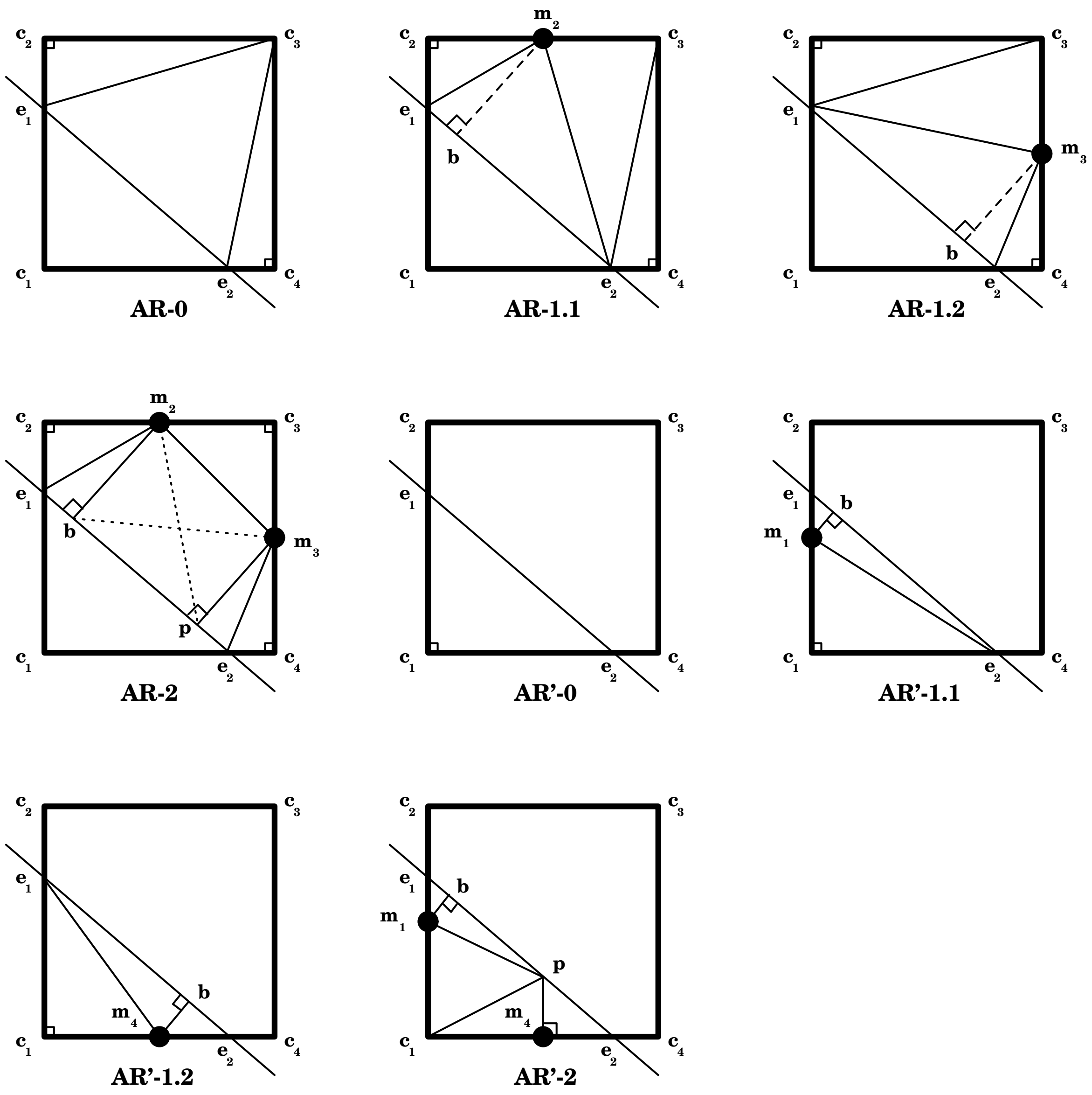
BR'-0 See figure 5.11. This case is trivial.

BR'-1 See figure 5.11. This case is trivial.

Above Right Intersections

When the input edge enters above the midpoint on the left, and right of the bottom midpoint, there are only 8 possible subcases—each side of the input edge has two possible midpoints, each of which may or may not be present.

AR-0 See figure 5.13. This case is trivial; the above right constraints on the input edge mean point c_3 must be in acute position to e_1e_2.



c_2 c_3 e_1 c_1 e_2 c_4
AR-0
m_2 c_2 c_3 e_1 b c_1 e_2 c_4
AR-1.1
c_2 c_3 e_1 m_3 b c_1 e_2 c_4
AR-1.2
m_2 c_2 c_3 e_1 b m_3 p c_1 e_2 c_4
AR-2
c_2 c_3 e_1 c_1 e_2 c_4
AR'-0
c_2 c_3 e_1 b m_1 c_1 e_2 c_4
AR'-1.1
c_2 c_3 e_1 b m_4 c_1 e_2 c_4
AR'-1.2
c_2 c_3 e_1 b m_1 p m_4 c_1 e_2 c_4
AR'-2

Figure 5.13: Subcases for adjacent above-right intersections.

AR-1.1 See figure 5.13. Point e_2 must be in acute position relative to m_2c_3. Angles $\angle e_2e_1m_2$ and $\angle m_2e_2e_1$ must be acute by the constraints on e_1 and e_2. And if $\angle e_1m_2e_2$ is not acute, then point b is added according to note 5.4.1.

AR-1.2 See figure 5.13. This case is actually a vertical reflection followed by a counter-clockwise rotation of $\pi/2$ of case AR-1.1.

AR-2 See figure 5.13. Points b and p can always be located along e_1e_2 as the intersection of a line coincident with e_1e_2 and a perpendicular to e_1e_2 passing through m_2 and m_3 respectively. Because of the slope constraints on e_1e_2 implied by this being an above-right case, the projection of m_2m_3 onto e_1e_2 has both a non-zero minimal length and is forced to lie entirely along e_1e_2; in other words, b and p necessarily lie along the segment e_1e_2, and bm_2 and pm_3 do not cross. The result is a quadrilateral, (b, m_2, m_3, p) containing two $\pi/2$-angles; thus, at most one of the remaining two angles can be obtuse—and even so it cannot be too obtuse, since the slope of e_1e_2 is constrained. If the quadrilateral is then divided into two triangles so as to split the obtuse angle, two acute triangles must result.

AR'-0 See figure 5.13. This case is trivial.

AR'-1.1 See figure 5.13. Point b is placed according to note 5.4.1.

AR'-1.2 See figure 5.13. This is a symmetric variant of AR'-1.1.

AR'-2 See figure 5.13. Point p is located as the intersection of the vertical bisector and e_1e_2. Since the center of the quad can never be included in the domain below e_1e_2, p must lie vertically between the horizontal bisector and m_4—in other words, in acute position relative to m_1c_1. Point b is then positioned as per note 5.4.1.

5.4.3 Case 1b: Opposing Intersections

An edge entering from one side and exiting from the other can be assumed without loss of generality to enter from the left side and exit from the right. There are then four possibilities: either the edge enters above or below the midpoint on the left, and it exits similarly on the right.

However, symmetry reduces the four cases to two. An above-above situation is a vertical reflection of below-below, and above-below is a rotation of below-above. Hence, only below-above and below-below actually need to be considered.

Below Above Intersections

When the edge enters from below and exits above, both sides of the input edge can be triangulated identically—one side is simply a π-rotation of the other. Thus, there are a mere 4 subcases to deal with here, corresponding to the two possible midpoints.

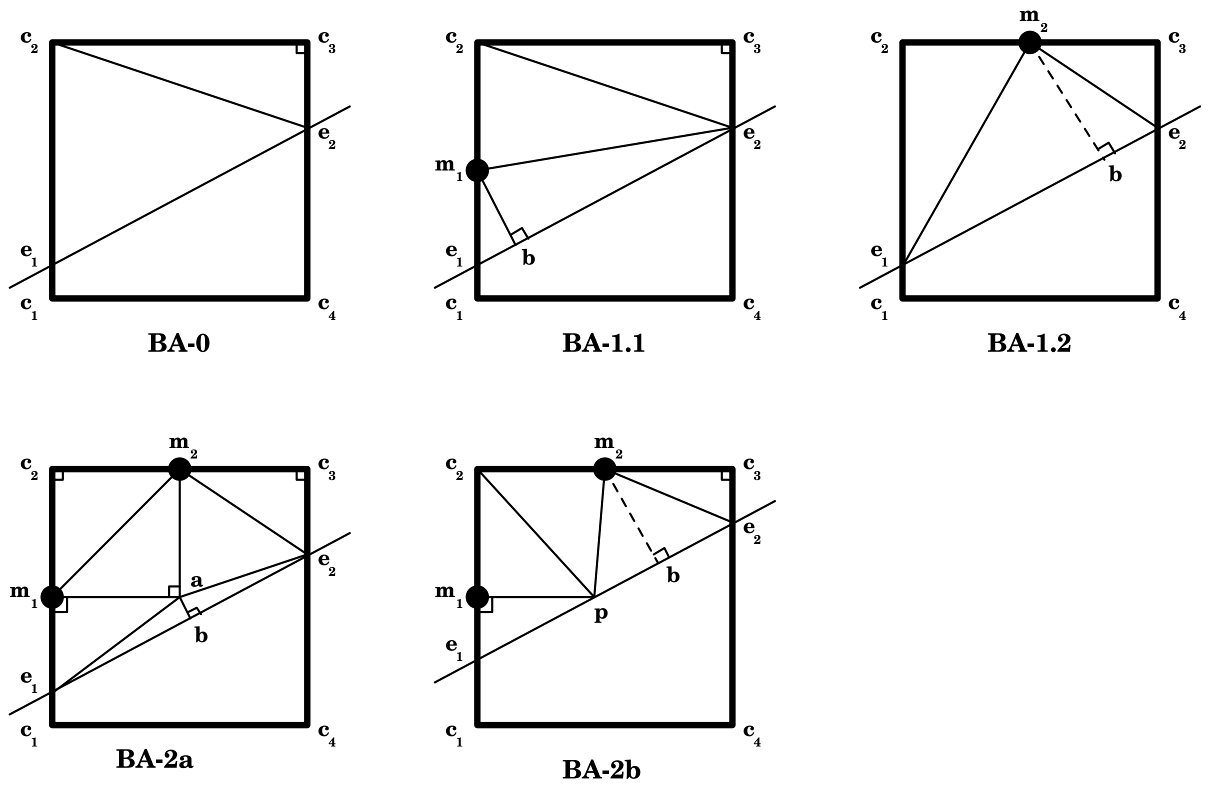


Figure 5.14: Subcases for opposing below-above intersections.

BA-0 See figure 5.14. This case is trivial; point e_2 must be located above the midpoint on the right, and so e_2 must be in acute position relative to c_2e_1.

BA-1.1 See figure 5.14. Again, point e_2 must be in acute position to c_2m_1. Point b is then located in accordance with note 5.4.1. Note that $\angle e_1m_1e_2$ must be obtuse, since e_2 is above m_1.

BA-1.2 See figure 5.14. Being a below-above case forces both $\angle e_2 e_1 m_2$ and $\angle m_2 e_2 e_1$ to be acute. The remaining angle, $\angle e_1 m_2 e_2$ may or may not be acute, but if it is not, point b is placed according to note 5.4.1.

BA-2 See figure 5.14. If the domain to be triangulated contains the center of the quad, then triangulation proceeds as in BA-2a. Point e_2 is in acute position to am_2, and point b is located to split the obtuse angle $\angle e_1 a e_2$, as described in note 5.4.1.

If, however, the domain does not contain a, case BA-2b applies. Here, point p is located as the intersection of the horizontal bisector and e_1e_2. Since the center is not contained in the domain, p must lie to the left of point m_2, and so p is in acute position to c_2m_2. As with case BA-1.2, $\angle m_2 e_2 e_1$ cannot be obtuse, and since $\angle m_2 p m_1$ is certainly obtuse, $\angle e_2 p m_2$ is certainly not ($\angle m_1 p e_1$, $\angle m_2 p m_1$ and $\angle e_2 p m_2$ must sum to π). Then, if $\angle p m_2 e_2$ is obtuse, point b is situated as in note 5.4.1.

Below Below Intersections

When an edge enters below the midpoint and exits below as well, the edge may or may not have a positive slope. However, if the edge does not have a positive slope, a horizontal reflection converts it to a case that does. This results in 10 cases: 8 for the side of the input edge that may have up to three midpoints, and two for the side that can have but one midpoint.

BB-0 See figure 5.15. This case is trivial; since the slope of e_1e_2 is positive, e_2 must be in acute position to c_2e_1.

BB-1.1 See figure 5.15. Since point e_2 is bound to be below m_1, m_1 must be in acute position to c_3e_2, and for the same reason e_2 is in acute position to m_1e_1.

BB-1.2 See figure 5.15. Since this is a below-below case, e_1 and e_2 are below the horizontal bisector. Hence, m_2 is in acute position to e_1e_2.

BB-1.3 See figure 5.15. Point a, being the center, must be in acute position relative to c_2e_1. Since e_1 and e_2 must both lie below the center, $\angle e_1 a e_2$ is certain to be at least $\pi/2$, which implies that $\angle e_1 e_2 a$ and $\angle a e_1 e_2$ are both acute. By note 5.4.1, point b can be placed to split the obtuse angle.

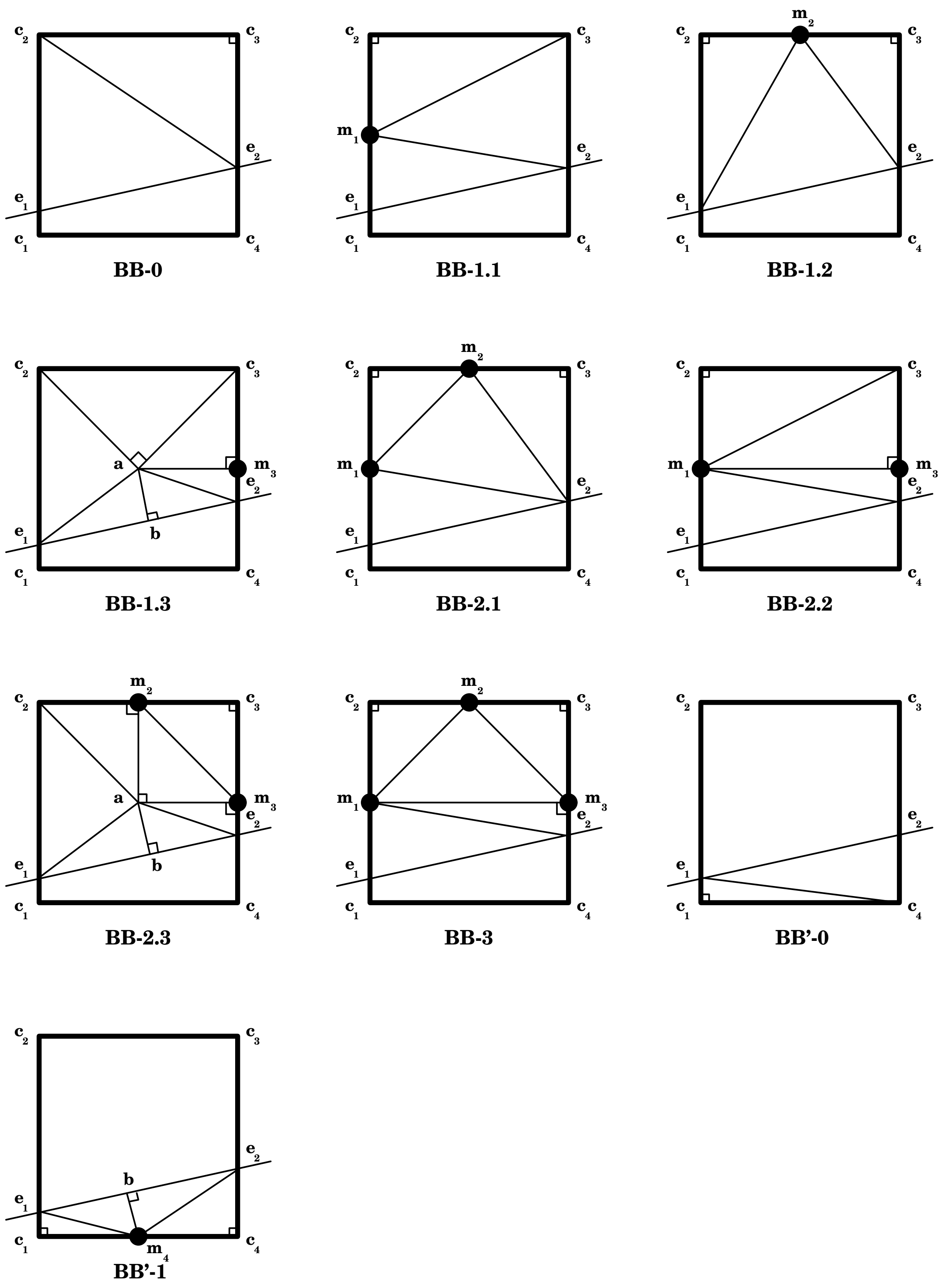


Figure 5.15: Subcases for opposing below-below intersections.

BB-2.1 See figure 5.15. Because point e_2 must lie below the center, $\angle m_1 m_2 e_2$ cannot be larger than $\pi/2$, and because e_2 is along the right side, $\angle e_2 m_1 m_2$ must be acute as well, putting e_2 in acute position to $m_1 m_2$. As with BB-1.1, e_2 is also in acute position relative to $e_1 m_1$.

BB-2.2 See figure 5.15. Again, as with BB-1.1, point e_2 is in acute position to $e_1 m_1$.

BB-2.3 See figure 5.15. This case is a trivial modification of BB-1.3.

BB-3 See figure 5.15. This case is a trivial modification of BB-2.2.

BB'-0 See figure 5.15. The slope of $e_1 e_2$ is positive, and so e_1 must be in acute position to $e_2 c_4$.

BB'-1 See figure 5.15. Since e_1 and e_2 are both bound to be below the horizontal bisector, $\angle e_2 m_4 e_1$ is necessarily obtuse. Thus, both $\angle m_4 e_1 e_2$ and $\angle e_1 e_2 m_4$ must be acute, and point b can be located as per note 5.4.1.

5.4.4 Case 2a: Two Input Edges With Non-Intersecting Domains

If two edges intersect the quad, and the domain sides of each edge do not intersect within the quad, then triangulation can proceed separately for each edge. Since only the domain sides will be triangulated, no conflict can result, and the independent triangulations of each edge can be (disjointly) merged.

5.4.5 Case 2b: Two Input Edges forming a Two-Edge Case

When two edges intersect a quad, their domain sides intersect within the quad, and they are not joined at a corner of the quad then the quad is recursively divided. This avoids considering many more cases, trying to triangulate the domain between two edges of more or less arbitrary orientation. Unfortunately, as can be seen in figure 5.1, recursively dividing such cases results in the quad actually containing the two edges joined at a corner having more than one “midpoint” along any or all of its sides.

Fortunately, a few observations and a simple scheme mentioned in Baker *et al*'s paper [BGR88] make this problem tractable. Although each side of the quad may have an arbitrary number of points, since only the interior of the domain is being triangulated only some of these points will have to be considered. As well, the region being triangulated will have two unconstrained edges—any number of points can be added to the boundary of the domain (the two input edges), and still the triangulation of such a quad will not affect its neighbouring quads.

Such a situation can have two different configurations; either both input edges intersect the same side of the quad, or they intersect adjacent sides. Without loss of generality the vertex shared by the two input edges can be assumed to be the upper left corner of the quad, and then the input edges either both intersect the bottom side, the right side, or one intersects the bottom and one intersects the right. Once again symmetry reduces the cases—if both edges intersect the right side, then a simple rotation by $\pi/2$ transforms the case so both intersect the bottom side.

Two-Edge Case: Both on the Bottom Side

When both edges intersect the same bottom side of the quad, the resulting figure is an obtuse triangle, with the leftmost edge forming an obtuse angle with the bottom side (see figure 5.16). Such a configuration can be triangulated by adding lines parallel to c_2e_2 at each of the points along the bottom side (*e.g.,* line p_1b_1). Each such line intersects input edge c_2e_1, and at each such intersection point a line perpendicular to c_2e_2 is projected out to edge c_2e_2. This decomposes the original obtuse triangle into a collection of rectangles, which can be trivially triangulated, and right-angle triangles.

Two-Edge Case: One on the Bottom Side, One on the Right Side

If the input edges do not intersect the same side, the result can be partially triangulated as shown in figure 5.17. Horizontal lines are extended from each point along the right side until they intersect the leftmost edge, c_2e_1. At each such intersection point, and from each point along the bottom side, a vertical line is extended up to the highest horizontal. This decomposes some of the region into rectangles and

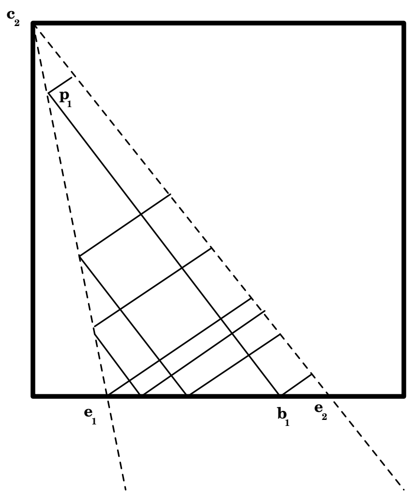


Figure 5.16: Triangulating the two-edge case when edges intersect the same side.

right-angle triangles. The remaining region is similar enough to figure 5.16 to be triangulated in the same manner.

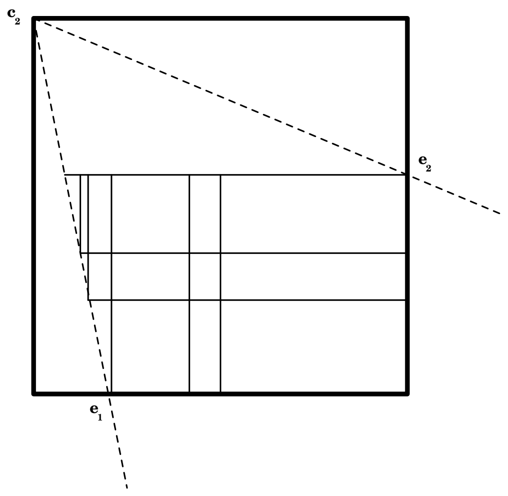


Figure 5.17: Triangulating the two-edge case when the edges intersect different sides.

5.5 Adaptivity

The above construction yields a complete triangulation of the domain computed from the leaves of the quadtree. This original structure constitutes a "base level," a minimum depth for each branch of the quadtree, enforced in order to permit independent triangulations of each of the leaves. However, it is possible to grow the quadtree deeper than needed if greater resolution (*i.e.,* more grid points) is required in a given region.

A leaf which is "deepened" after reaching the base level of triangulation may cause other leaves to require deepening, in accordance with the criteria discussed above. Fortunately, as long as the quad to be deepened is not a two-edge case, the base level construction ensures that no quad deepened beyond the base level will need to be deepened for any other reason that the balance condition. Once a quad contains only a single edge, for example, recursively dividing it can never make that quad contain more edges, and once all input vertices lie at corners of quads, deepening the quads further cannot move them.

5.5.1 Adapting On a Budget

An efficient adaptive algorithm needs to have limited external effect when a given region is adapted. Within the quadtree context, this means deepening a quad beyond the base level should not propagate deepenings throughout the grid, or at least should limit the number. This algorithm, "dampens" the propagation of the balance condition, to the point where the following lemma can be stated. Note that the proof of this lemma requires a fairly lengthy case analysis—more direct proofs certainly exist; however, the structure of this proof is used below to illustrate the partitioning method we will eventually apply to these grids.

Lemma 5.5.1 *If a non two-edge case leaf at depth d in the quadtree is deepened by one level (after the base level has been reached), then it will take at most $O(d)$ deepenings to restore the balance condition throughout the quadtree.*

Proof: The proof is inductive, relying on a recursive situation where only a single quad is "out of balance" with its neighbours. Note that it is not necessary to consider all possible arrangements of quads of varying depths—a quad at depth d just deepened to four quads of depth $d+1$ cannot propagate the imbalance to quads at a depth greater than $d-1$. In other words, the number of deepenings required to restore the balance condition will be maximized when quads neighbouring the unbalanced quad are at as shallow a depth as possible.

Also note that the inclusion of arbitrary boundaries or two-edge cases does not change the maximum effects of propagation. Although deepening the latter can generate many more quads (depending on the angle), the balance condition does not apply to either of these types of quad, and so any unbalanced quads adjacent to an exterior or two-edge quad will not cause more deepenings in the direction of the exterior/two-edge quad. The maximum number of deepenings will always occur in the absence of these special quads.

Since induction will be on the depth of the unbalanced quad, the base case, depth 0, is trivial—the root quad cannot be unbalanced with respect to its neighbours, since it does not have any.

Assume, then that there is a single quad, q at depth $d > 0$ that has just been deepened, and so is unbalanced with respect to its at least one of its neighbours. The discussion that follows demonstrates that after some constant number of deepenings, the quadtree will either again contain a single unbalanced quad with depth less than d, in which case the inductive argument holds, or will be completely balanced, or will contain two unbalanced quads of depth less than d. The latter situation will turn out to be closed—it either eventually produces a situation with just a single unbalanced quad, or it maintains exactly two.

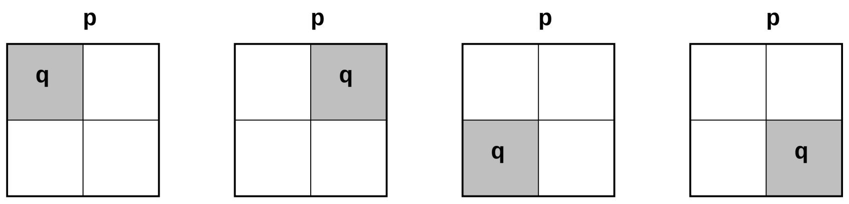


Figure 5.18: Possible positioning of q as a child.

Quad q must have a parent quad since $d > 0$. Call this parent quad p. Hence, q is an upper-left child, an upper-right child, a lower-left child, or a lower-right child (see Figure 5.18). Since these are all rotationally symmetric, without loss of generality q can be assumed to be an upper-left child.

The neighbours of p, must all satisfy the balance condition, since q is assumed to be the only unbalanced quad. In fact, because p has children, p must be surrounded by quads of depth no less than p itself. Of course p must also be the child of some quad (or the same degenerate truth used in the base case will apply), in one of the four possible child positions. Two situations can then arise—either the deepening of q will affect (unbalance) one or two neighbours of p. All the legal quadtree decompositions for the first case are illustrated in Figure 5.19[2] and for the second case in Figure 5.20. In each case p is shown as the inner square with a heavy outline.

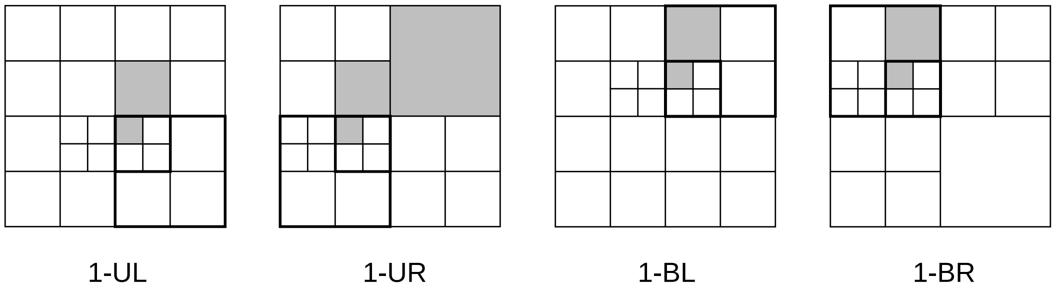


Figure 5.19: Legal quadtrees when p is a child and q unbalances only one neighbour.

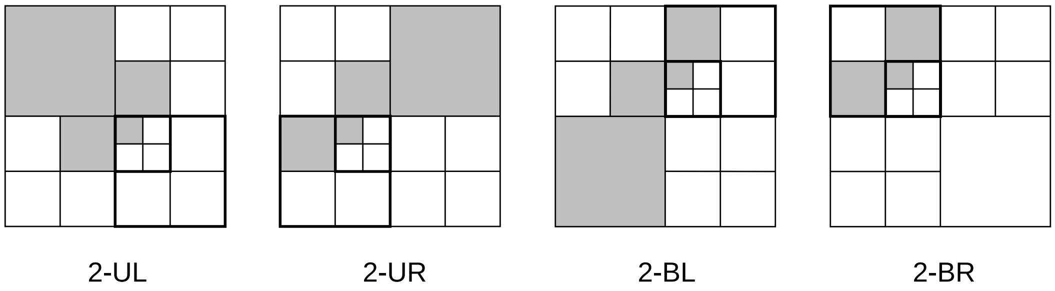


Figure 5.20: Legal quadtrees when p is a child and q unbalances two neighbours.

The first case in Figure 5.19, when p is an upper-left child (*1-UL*), is terminal—the quadtree is completely rebalanced. The inductive assumption applies to the other three cases, since they satisfy the initial requirement with a smaller depth—a single quad of depth $d-1$ or $d-2$ is left unbalanced in each situation, after causing either 1 or 2 quads to require deepening.

Unfortunately, it is not as simple when q unbalances two neighbouring quads. The first case in Figure 5.20 (*2-UL*) results in a single unbalanced quad of depth $d-2$, and so the inductive assumption applies. It is also apparent that *2-UR* is a symmetric variant of *2-BL*, leaving two cases, *2-BL* and *2-BR*, for which we must consider p's parent.

When p is assumed to be a below-right child, the situation is as shown in Figure 5.21. Three of the cases, (*2-BR-UL*, *2-BR-UR*, *2-BR-BL*), result in a

[2]The symmetric variation when the other quad adjacent to q is unbalanced is not shown.

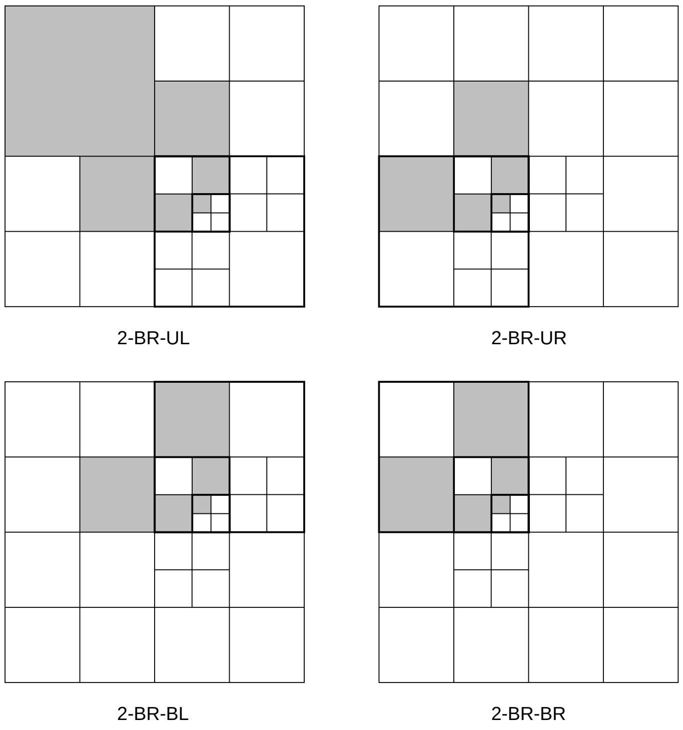


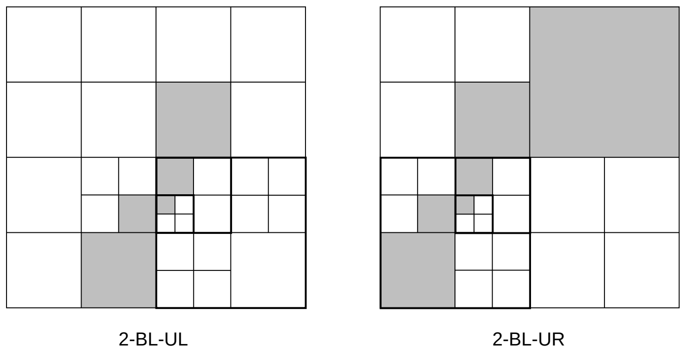
Figure 5.21: Further expansion of *2-BR*.

single unbalanced quad of either depth $d-2$ or $d-3$ after either 4 or 5 extra deepenings, and so the inductive argument can again be applied. The last case, (*2-BR-BR*) generates two quads at depth $d-2$.

The second possibility occurs when p is assumed to be a below-left child, and the four possibilities for its parent are shown in Figure 5.22. Two of these (*2-BL-UL* and *2-BL-BL*) result in the familiar recursive situation, and two result in two unbalanced quads.

Thus, of all the possibilities, only three result in a non-terminal, non-recursive situation: *2-BR-BR*, *2-BL-UR*, and *2-BL-BR*. The four further possibilities of *2-BL-UR* are shown in Figure 5.23; three of these result in a single unbalanced quad to which the inductive argument can be applied, and the fourth (*2-BL-UR-UR*) is terminal.

This leaves only two possibilities, *2-BR-BR* and *2-BL-BR*, that might create more than a constant number of deepenings per level of depth. These two cases, however, can be characterized by the pattern shown in Figure 5.24.



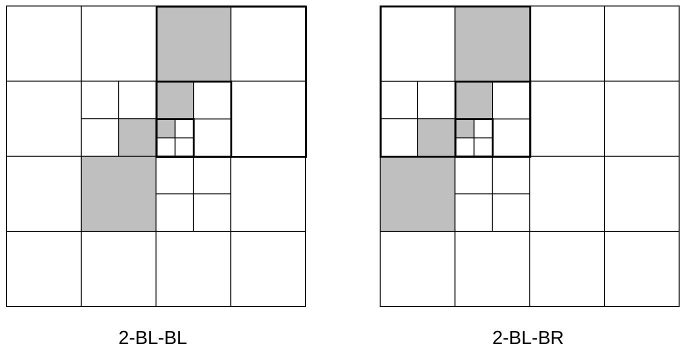
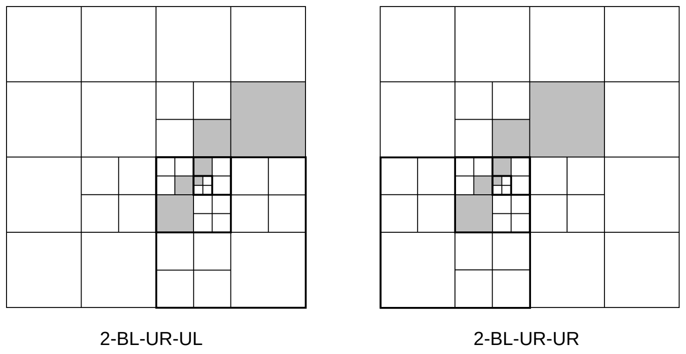


Figure 5.22: Further expansion of *2-BL*.



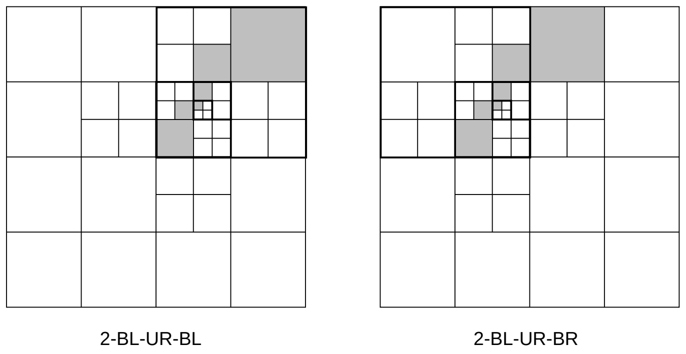


Figure 5.23: Further expansion of *2-BL-UR*.

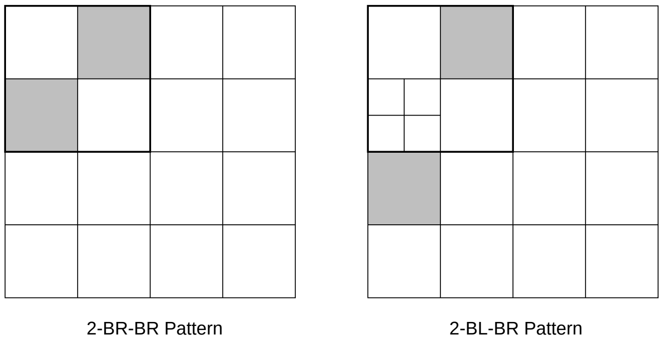


Figure 5.24: Patterns of growth when two quads are unbalanced.

In each case, four quads of depth d' containing two unbalanced quads of depth $d'+1$ are subdivided into quads of at least depth $d'+1$ around the perimeter of the depth $d'-1$ square (note that this outermost square may or may not form a quad). Internally, within the depth $d'-1$ square, all quads are balanced, and imbalance only occurs between the two indicated quads and quads external to the square. If these two patterns are then expanded one more level (Figures 5.25 and 5.26), it becomes apparent that both patterns either repeat exactly at a smaller depth, or are reduced to a single unbalanced quad at a smaller depth.

Thus, the deepening of a single quad can produce at most 7 further deepenings before a single quad of a strictly lesser depth remains as the only unbalanced quad, or two unbalanced quads remain in the aforementioned pattern. Since the latter case either repeats or devolves into the former, the inductive argument holds. □

From this we can extract an "exponential dampening" theorem, demonstrating that adapting has a guaranteed small bound on non-local effects of a local refinement:

Theorem 5.5.1 *Let s be a non-two-edge case at depth d in the quadtree and let s' be a leaf quad at depth $d' < d$ affected by the deepening of s (d, d' both greater than or equal to the base level). Then s' is no more than $O(d-d')$ quads distant from s.*

Proof: From the proof for lemma 5.5.1, it is clear that at most a constant number of quads can require deepening per level in the chain of deepenings connecting s to s'. Thus, if the difference in depth between s' and s is $d-d'$, then the chain of affected quads between s and s' is of length at most $c(d-d')$ for

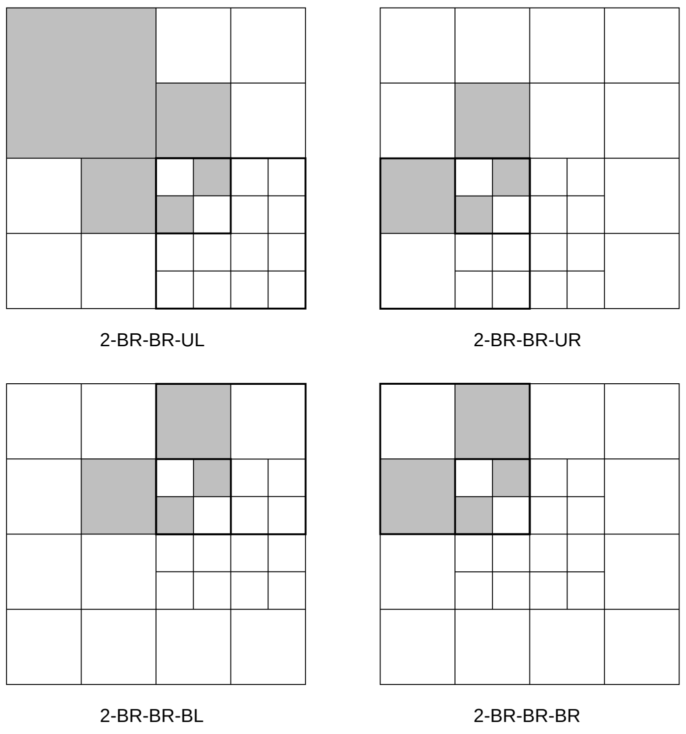


Figure 5.25: A schematic *2-BR-BR*, one step further.

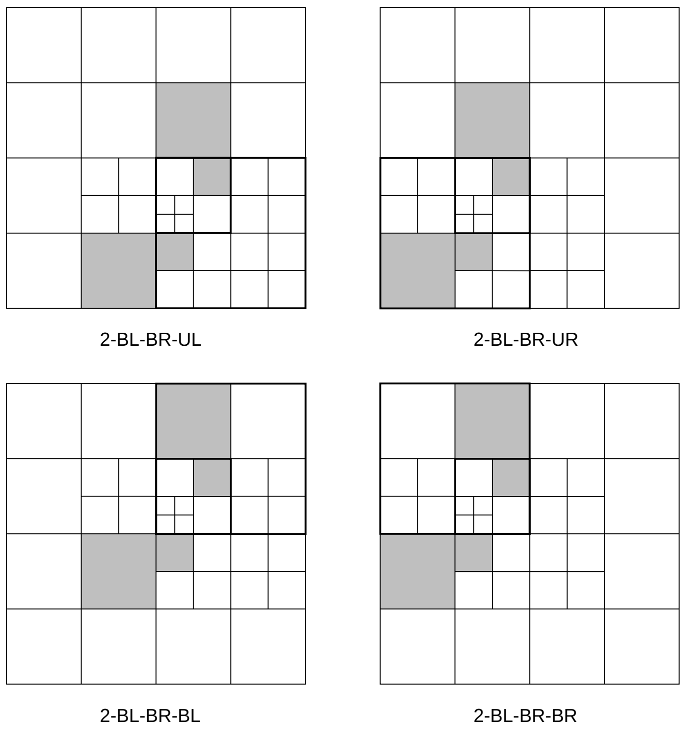


Figure 5.26: A schematic *2-BL-BR*, one step further.

some constant c. □

5.5.2 Unadapting

Once a quad has been deepened beyond the base level, it can be subsequently "undeepened" to reduce resolution of the grid. Note that these operations are complementary, but not inversive—even if all four children of a given deepened leaf are subsequently undeepened, it may not restore the quadtree to its original condition.

In order to undeepen a quad deepened below the base level, the only condition that needs to be checked is the balance condition, which may propagate undeepenings in a manner similar to the deepenings. Since an undeepening merely reverses the effect of a deepening, and one can can never undeepen higher than the base level, undeepenings will have the same upper bound on cost as deepenings.

5.6 Runtime Analysis

The exact number of triangles/nodes generated by this algorithm is geometry-dependent. While deepening any single node/square can generate at most a depth-bounded number of further deepenings, the number of nodes that will have to be deepened to satisfy the vertex and edge conditions is not so easily determined. The vertex condition, for example, demands that the depth of a node is lower-bounded by the length of the binary expression of its input coordinates (with respect to the bounding square). The edge condition merely implies that there must exist a grid of sufficient resolution to ensure no two (non two-edge case) edges with a non-empty intersection of interior domains intersect the same grid square. This is sufficient for termination, but since the resolution required will depend on the distance between edges and their orientation with respect to the bounding square, it is not possible to give very tight *a priori* bounds. We can, however, give an upper bound, for which the following definition is needed:

Definition 5.6.1 *The interior angle between two edges, e_1 and e_2 connected at a vertex v* straddles *the x-axis (respectively, the y-axis) if there exists an infinite ray r, parallel to the x-axis (y-axis) starting at v with some continuous segment of r including v entirely interior to the domain.*

Definition 5.6.2 *The* smallest feature *of a polygon is the smallest distance between any two distinct vertices, or between any two non-intersecting lines.*

Note that the smallest feature is always defined for every polygon, and since it is defined only on non-intersecting objects, is always larger than 0.

Lemma 5.6.1 *Let b be the maximum length of the binary expression of any input vertex (with respect to the bounding square), f' be the smallest feature,*

$$f = \left\lfloor \log_2 \left(\frac{f'}{\sqrt{2}} \right) \right\rfloor$$

and let θ be the smallest angle interior to the domain that straddles neither the x nor the y-axis. If θ does not exist, or equivalently if $\theta \geq \pi/2$, then the smallest quad has sides of length no smaller than $\min(2^{-b}, 2^{f})$.

Proof: The node condition is satisfied by the 2^{-b}; every vertex must lie on the corner of a quad, since no vertex has binary expansion larger than b. This also helps with the edge condition with respect to intersecting edges. Since each vertex is required to lie on a quad corner, either the interior angle is larger than $\pi/2$, or the two edges straddle an axis—in either case, intersecting edges must lie in different quads. Once quads are small enough that no two non-intersecting features can lie within the same quad, the edge condition must be satisfied. The balance condition, of course, cannot cause a quad to be deeper than the maximum depth. □

Alternatively, if $\theta < \pi/2$, then some two-edge case will exist in the triangulated domain, and can result in much smaller quads being generated. Let e_1, e_2 be a pair of intersecting lines with interior angle θ, and let c be the corresponding cone. Let y_1, y_2 be the points of intersections of the arms of c and a line parallel to the y-axis at x-distance 1 away from v, and let x_1, x_2 be similarly defined for the x-axis. Since $\theta < \pi/2$ and the edges do not lie on nor straddle either axis, these intersections are uniquely defined in both cases.

Lemma 5.6.2 *Let d' be the smaller of the width of c at x_1, x_2, y_1 and at y_2, and let*

$$d = \left\lfloor \log_2 \left(\frac{d'}{\sqrt{2}} \right) \right\rfloor$$

Then the smallest square will have sides no smaller than $\min(2^{d-b}, 2^{d+f})$.

Proof: The node condition, and all edge conditions concerning non-intersecting features are satisfied according to Lemma 5.6.1. However, even with such bounds it may happen that a two-edge case occurs in a quad of minimum size, in which case some of the surrounding quads may then contain two edges with intersecting domains. A minimum quad size of 2^d, though, is sufficient to ensure that the two edges exiting from a two-edge case do not lie within the same quad. Scaling this down to the minimum quad size for non-two-edge cases provides the given bounds. Note that two two-edge cases cannot interact; since the two cases do not intersect they are already separated by the minimum distance between features. $\square$

5.7 Experimental Results

Expected behaviour of our algorithm is difficult to determine without some model of expected input. Nevertheless, our initial attempts at quite complex domains are very encouraging, with the number of nodes being essentially linear in the depth of the quadtree. Here we present the results from three very different domains: an unsymmetric wrench (Figure 5.27), a hammer and sickle (Figure 5.28), and an almost-unit square[3] (Figure 5.29).

The wrench consists of 40 input nodes. The total number of nodes generated (and including the input) verses the maximum allowed binary expansion of input coordinates (*i.e.,* binary digits of input precision) is plotted in Figure 5.30. For the sickle, consisting of 35 nodes, the results are in Figure 5.31, and for the 4 node square in Figure 5.32. Neither the wrench nor the square contain any 2-edge cases. The sickle contains just one 2-edge case (at the upper tip of the hammer).

In each case, most or all of the coordinates have been chosen for their relatively long binary expansions, overconcentrating quadtree nodes (and hence triangles) around these points. Since there is usually some freedom in the specification of input vertices[4] it is straightforward to produce a much smaller and more balanced triangulation. Note that such coordinate selection does not change the domain—just the location along the domain of the selected vertices. The arch in Figure 5.33,

[3]A true unit square would be triangulated by just two triangles.

[4]Artificial domains are often selected with an eye to the perspicuity of their input coordinates, and actual domains are rarely so precise that some variation is not possible.

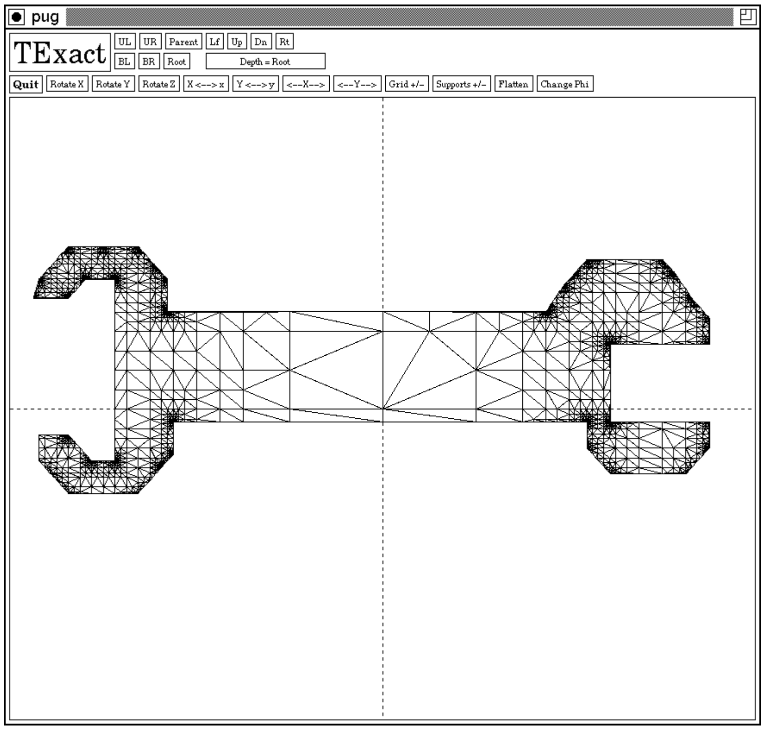


Figure 5.27: Triangulation of Wrench.

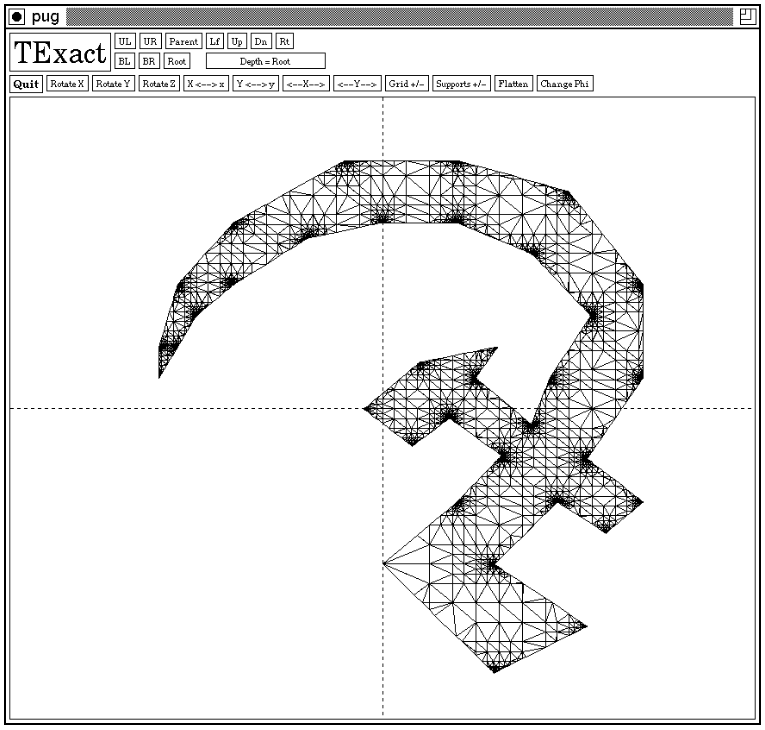


Figure 5.28: Triangulation of Hammer & Sickle.

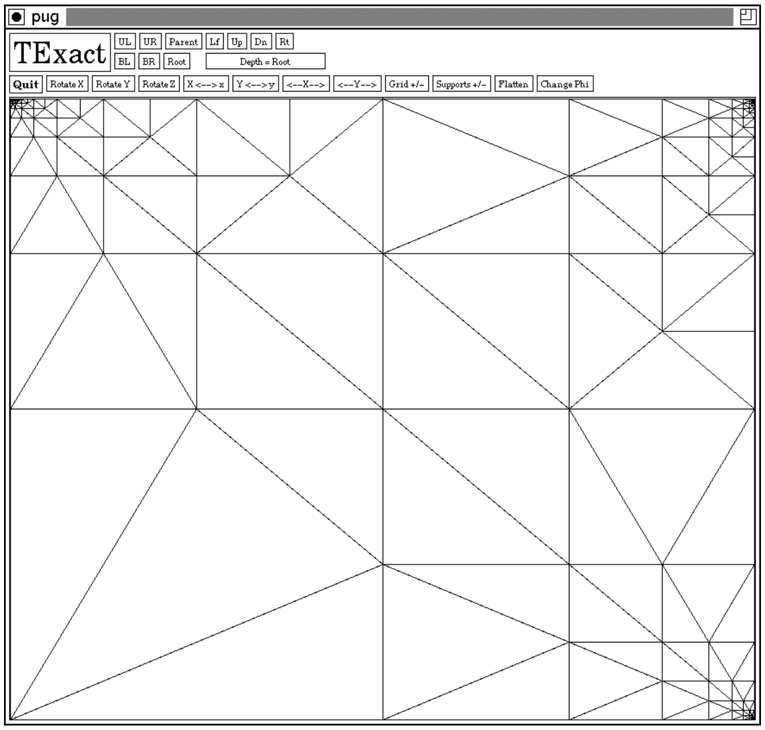


Figure 5.29: Triangulation of Square.

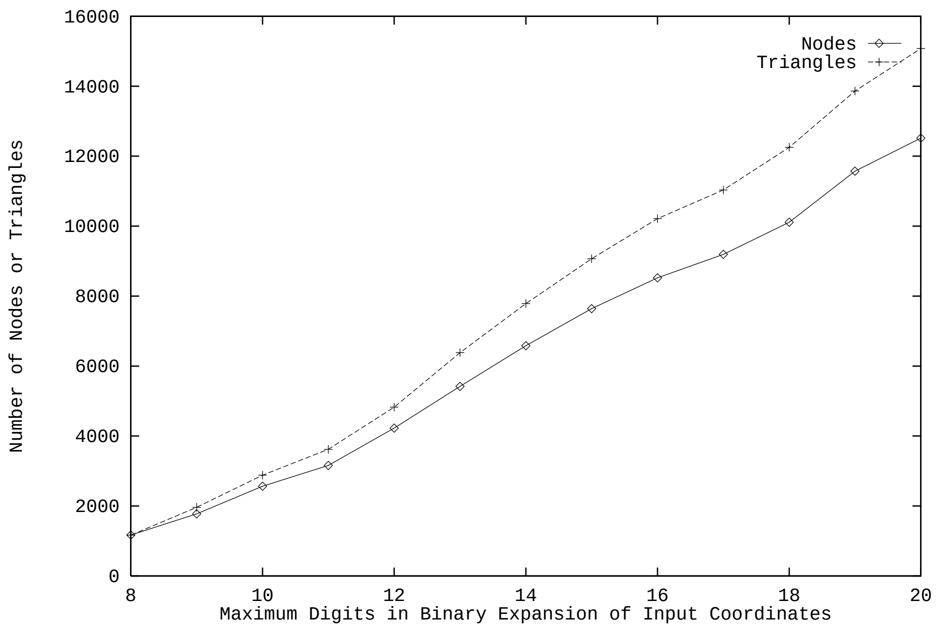


Figure 5.30: Vertices versus Precision for Wrench.

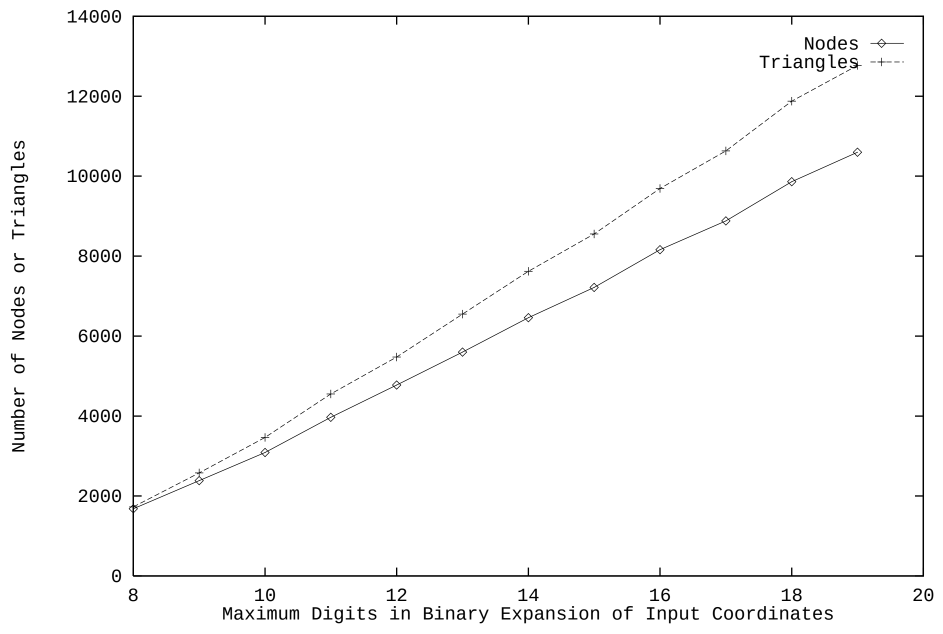


Figure 5.31: Vertices versus Precision for Hammer & Sickle.

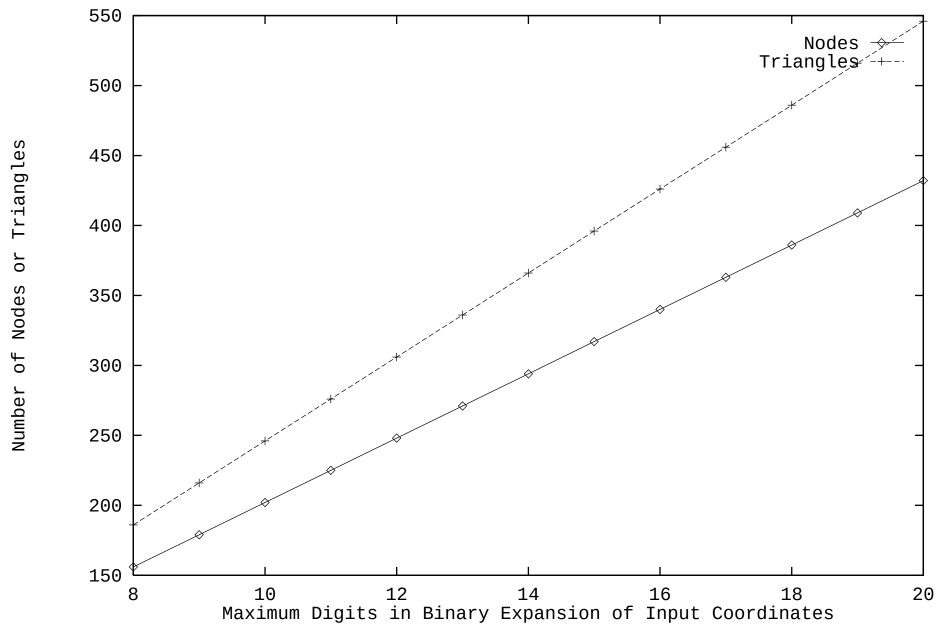


Figure 5.32: Vertices versus Precision for Square.

for instance, reaches its base level with a maximum quadtree depth of just 5 (*i.e.,* 5 digits of binary precision in input coordinates).

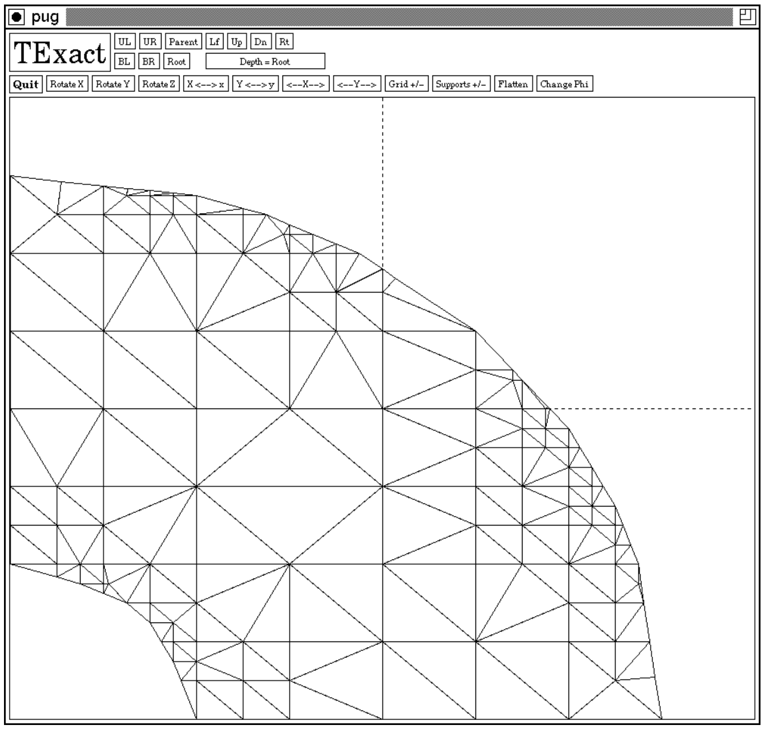
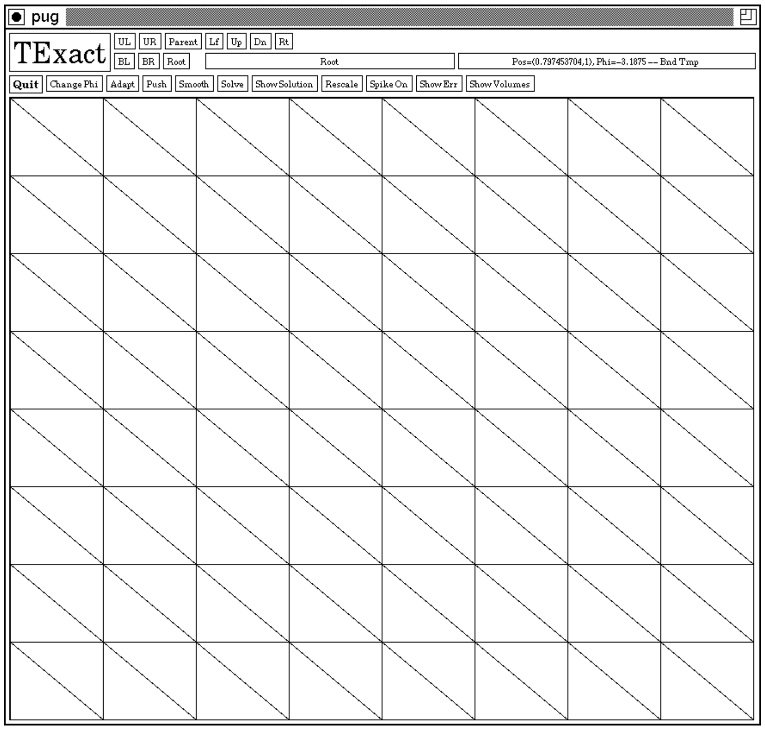


Figure 5.33: Triangulation of Arch.

The process and results of adapting a grid are shown in Figures 5.34, 5.35, 5.36 and 5.37. The first one, Figure 5.34, shows the initial grid, a full quadtree of depth 3 on the unit square. After solving this grid and adapting where the solution gradients are highest, the resulting grid is shown in Figure 5.35. This process is then repeated, producing grids 5.36 and 5.37. Note that the adaptation reflects the slightly asymmetric nature of our test problem, adapting more in the y-direction than in the x-direction.

5.8 Partitioning CVFEM

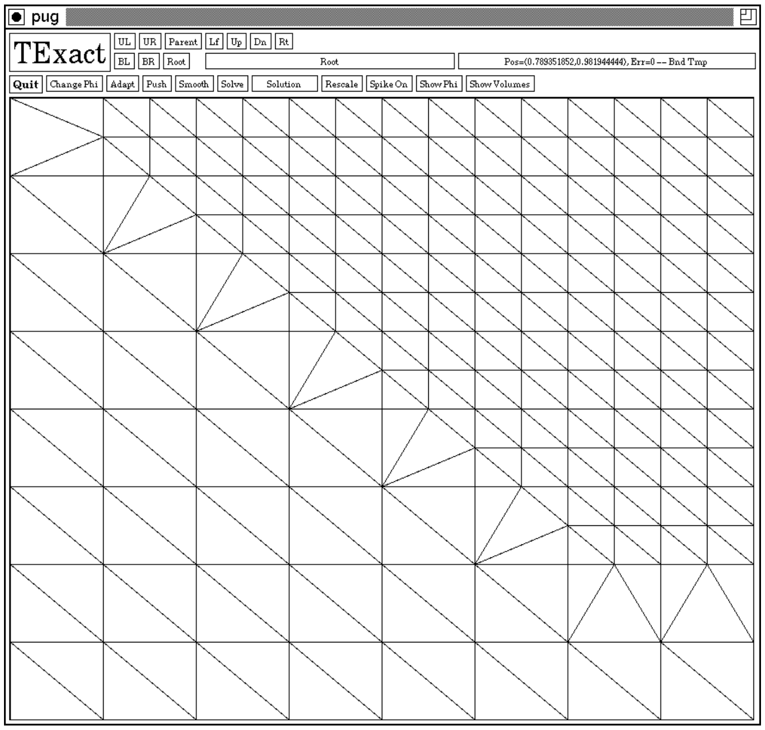
The solving phase of CVFEM consists of a repetitive update of each node in the grid based on the values stored in its immediate neighbours. This process is iterated until the node values reach convergence, which may require a great deal of computation. For this reason, in a parallel implementation of CVFEM, the usual approach is to coarsely partition the grid into independent sections, and have one process dedicated



Root

Pos=(0.797453704,1), Phi=-3.1875 -- Bnd Tmp

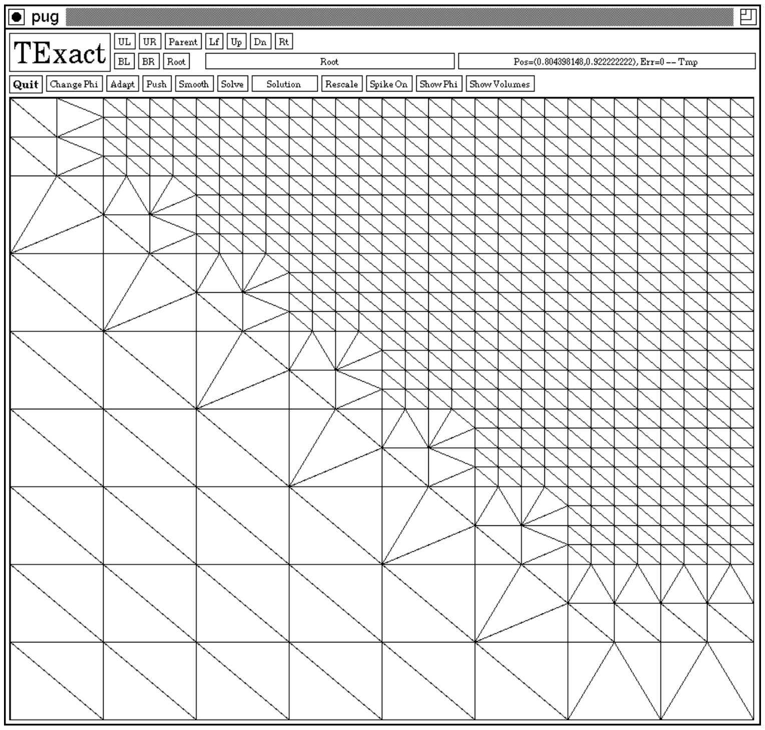
Figure 5.34: Initial triangulation of unit square.



Root

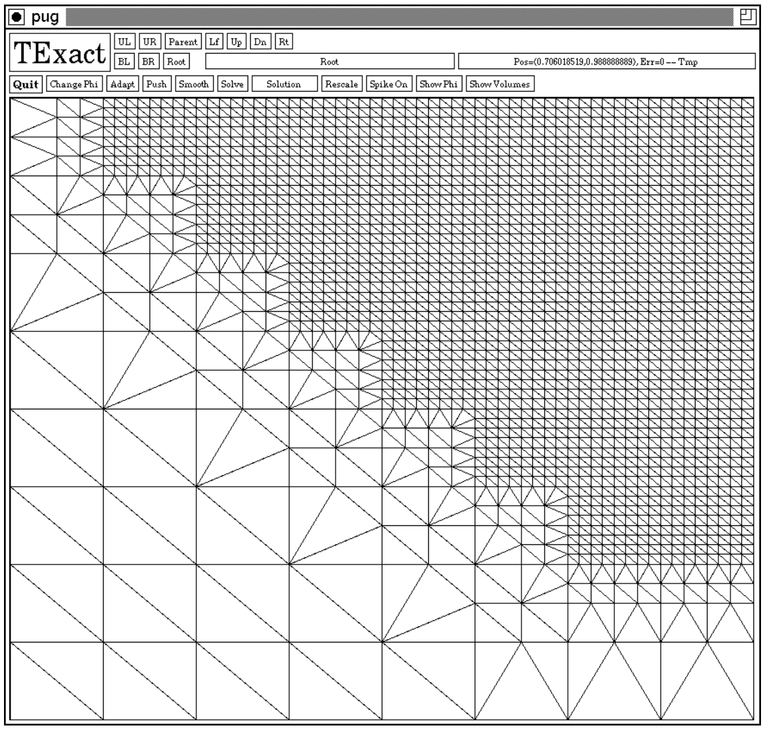
Pos=(0.789351852,0.981944444), Err=0 -- Bnd Tmp

Figure 5.35: First adaptation of unit square.



```
Pos=(0.804398148,0.922222222), Err=0 -- Tmp
```

Figure 5.36: Second adaptation of unit square.



```
Pos=(0.706018519,0.988888889), Err=0 -- Tmp
```

Figure 5.37: Third adaptation of unit square.

to each partition. Volume to surface area (or area to perimeter) arguments suggest that that this is a good strategy, requiring minimal interprocess communication.

However, partitioning irregular grids is quite hard in general, and one is usually forced to resort to heuristic techniques in these situations. This is particularly unfortunate in our case, since the shape of our grids is governed by the quadtree used to create and maintain the grid—*i.e.*, there is structure but it is unlikely any general heuristics will be able to make use of this information. Moreover, in order to support the adaptive refinements, the quadtree support structure should be partitioned as well, making the problem even harder.

The solution is of course to just partition the quadtree, and let the grid itself be partitioned thereby. All non-two-edge cases contain only a constant-bounded number of triangles, and actual two-edge cases have shown themselves rare, so this should be a very useable technique. Unfortunately, an efficient implementation of the quadtree data structure also includes "level-threading," or pointers connecting each quad (except 2-edge cases) to its 4 immediate neighbours at the same depth in the quadtree (fully-described in the next section). This allows for a constant-time check of the balance condition when deepening a quad, and for rapid navigation when inspecting the structure after or during construction. Thus the graph to be partitioned is neither planar, nor a quadtree—meaning most domain-specific heuristics will be of no help, and even direct methods for partitioning quadtrees will need to be tailored to manage the level-threading. It is also not clear what might be the impact of adaptivity. Still, these are not insurmountable problems, and an *ad hoc* partitioning method can be developed with some effort.

The problem, though, can be easily expressed in the path extended graph grammar formalism developed in Chapter 3. Such an expression mirrors the graph construction, and so the partitioning can be discovered as the graph is created, not as an additional post-processing step; this can have a large impact on partitioning in the presence of incremental changes, like adaptivity.

5.8.1 The Data Structure

The main data structure used to support our implementation of this algorithm is a "level-threaded" quadtree. This is a simple quadtree with pointers between neighbouring quads at the same level, at each level in the tree. However, due to our balance condition, we may have some leaf quads which do not have neighbours at the same level; in these cases the level-threading is directed upward one level in the tree—when or if the larger quad is expanded into 4 child quads, these upward-directed pointers can be just shifted down one level to the new children. A small example is shown in Figure 5.38. This upward-directed level-threading is only maintained when quads differ by one level (*i.e.,* within the interior of the quadtree, and not for a two-edge case); this design ensures a bounded number of pointers at each quadtree node.

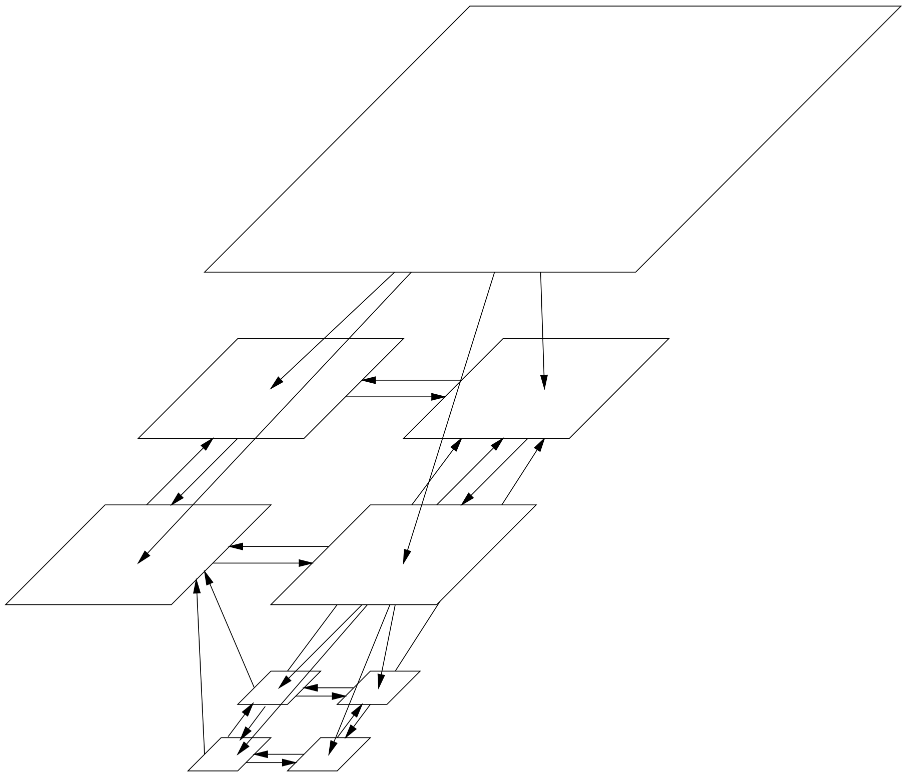


Figure 5.38: A level-threaded quadtree.

As mentioned, the level-threading is used to ensure constant-time access from one quad to its neighbours. To support our quadtree construction, this is only really critical for leaf quads. If unadapting is to be supported, though, retaining upper-level neighbour pointers can become useful once more (as interior nodes become leaves again), and even within the base-level of the quadtree these pointers can be used for fast and convenient navigation through the structure. For these reasons, level-threading is maintained at all levels of the quadtree.

5.8.2 A Dangling Graph Grammar for Partitioning

Each change in the graph as the grid is constructed can be seen as a deepening of one quadtree leaf node, followed by a chain of deepenings as required by the balance condition. Thus, if we can describe a grammar that follows this chain, deepening and updating level-threading pointers, we will be able to generate the entire tree using graph grammars, and then be able to show a partitioning with appropriately derived communication cost.

The idea for such a grammar is derived from the proof of the maximum costs of adaptivity used in Section 5.5: we will use path extended productions to mimic the update patterns seen in the proof of Lemma 5.5.1. For the purpose of clarity, we will only describe one of the symmetric aspects of the rule—the (constant number of) others can all be generated with simple changes to the labels of the parent-child connections. We assume leaf nodes and external quads are labelled differently from the internal nodes.

Consider Figure 5.18: q can be any child of p, so we need a rule which *or*s together 4 graphs, as in Figure 5.39, to express any one of the four possibilities. This way q may be connected by its parent 1/2-edge, either as an upper-left, upper-right, below-left or below-right child of p. A similar approach will apply each time we move up a level and are faced with four possible connections.

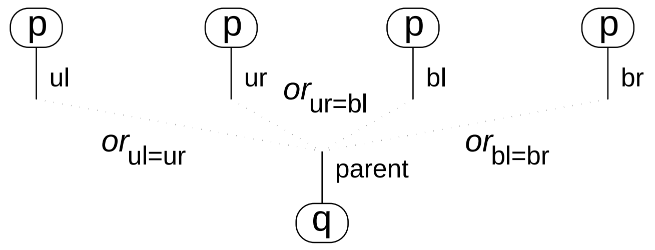


Figure 5.39: Possible positioning of q as a child, expressed as a rule.

We can describe each static situation in the same way, as some number of graphs *or*-ed together. The only difficulty then arises from the two inductive parts of the proof: we will not be able to just generate a rule of fixed size to describe the chain of quads in need of deepening. Fortunately, this is easily dealt with using the iteration operator of path extended productions. The complete expression is too large to include as a recognizable figure; however, a schematic portion of the expression is shown in Figure 5.40. Neither edge-labels nor the actual graphs at each point are shown, but note the use of the iteration operator to indicate that the whole situation may be repeated inductively, and that within any such repetition there may be a nested repetition of *2-BR-BR*. Also note that this expression corresponds to a worst-case scenario; all possible combinations where the chain of deepenings does not propagate further have to be factored in as well. This increases the size of the expression, but again only by a constant factor.

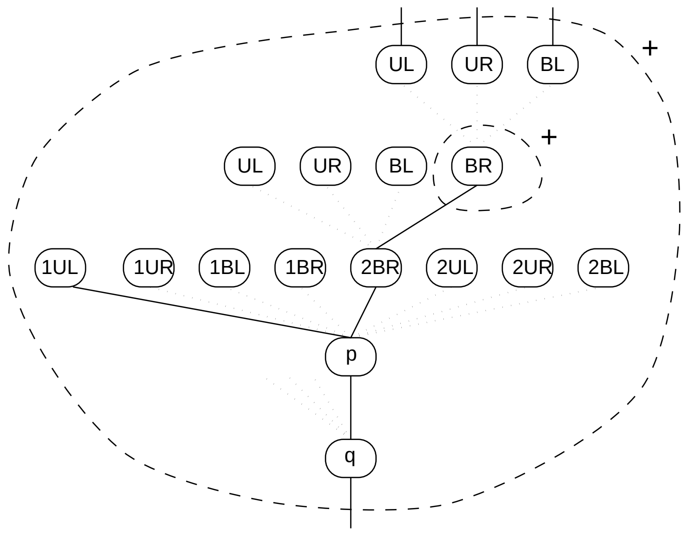


Figure 5.40: Schematic path expression for rebalancing.

A rule which maps this one large path expression (as a source graph) to a similar expression with each static graph having the appropriate nodes expanded (the target), generates the entire semi-balanced quadtree. Each time we deepen a node because it has not yet satisfied either the edge condition or the vertex condition we apply our rule, and the balance condition is then automatically satisfied, and our level threading can be maintained as well. A fragment of this rule, corresponding to the terminal *1-UL* situation, is shown in Figure 5.41. Most edges and connections are not shown, but note how both q and $q3$ are expanded simultaneously, and how a level-thread is established between one of the new descendants of $q3$ and the original q. Also note that most of this rule is non-rewritten *context;* only q and $q3$ are actually rewritten.

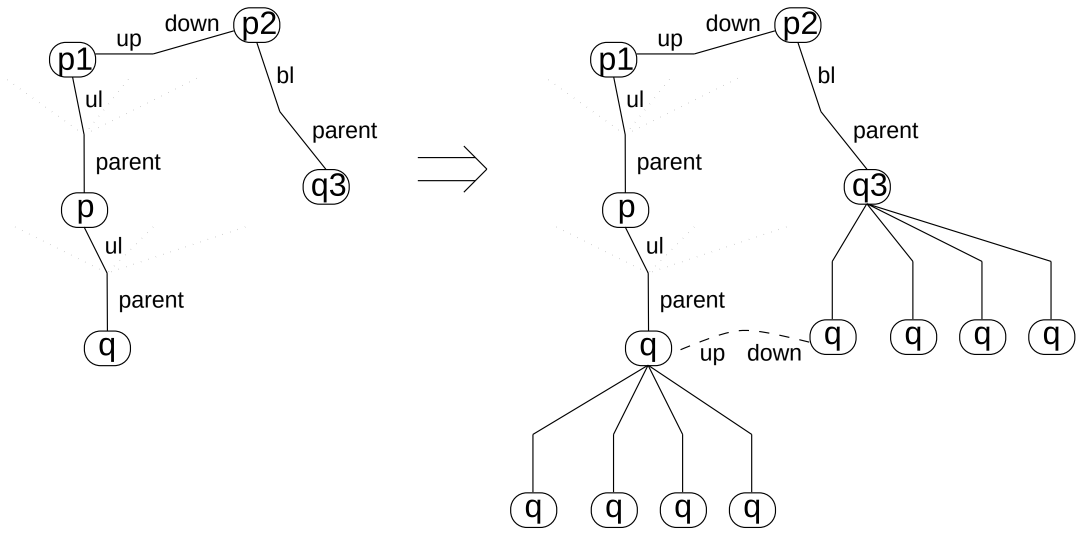


Figure 5.41: Fragment of Rebalancing Rule for *1-UL*.

5.8.3 Partitioning Bounds

The rule described above incrementally generates the quadtree by deepening one node, accounting for the balance condition and then repeating. This means that the rule must be applied multiple times in order to satisfy the vertex condition and edge conditions as they arise. In other words, any quad containing an input vertex having coordinates with maximum binary expansion d will have to be deepened at most[5] d times, and the edge condition implies a similar, geometry-dependent observation.

This is the basis for an upper bound on partitionability. If the entire quadtree is constructed by repeatedly deepening n input vertices at most d time each, then Theorem 3.6.2 allows us to derive an upper bound on partitionability, at least when the edge condition does not play a significant role.

Proposition 5.8.1 *Let Q be a level-threaded quadtree as produced by our grid generating method given an input domain of n nodes with maximum binary expansion d in any coordinate. Assume Q contains no two-edge cases, and that if any quad q deepened during the construction of Q failed the edge condition, then q also failed the vertex condition. Then Q is $O(nd^2\log(|V|))$-partitionable.*

Proof: Since the vertex condition, and the subsequent chain of rebalancings are the only reasons any quad is deepened, Q can be expressed by applying the

[5]Fewer deepenings are required if the containing quad was deepened as a result of the rebalancing due to some other deepening.

rule described in Subsection 5.8.2 at most d times to the quads containing each of the n input vertices. Each application is of a path-expression whose occurrence is formed of at most d concrete subexpressions. Thus, by Theorem 3.6.2, Q must be $O(sd\log(|V|))$-partitionable, where $s \leq dn$. $\square$

It is unlikely we would actually reach this worst-case complexity in practice. The bound on s we use to derive Proposition 5.8.1 requires a rather precise relationship between input nodes, which is certainly rare in practice, and may not even be physically realizable. Hence, in Table 5.1 we give experimental results, comparing the partitioning of the level-threaded quadtrees produced using our graph grammar-based method, and the partitionings produced by two "state-of-the-art" heuristic graph partitioners, Jostle and Metis. Both these partitioners use a combination of heuristic methods, and are described further in [WCE+95] and [KK95] respectively.

Three graphs were partitioned: the wrench of Figure 5.27 at maximum binary precision 12 and 18, resulting in graphs with 3677 and 8573 quadtree nodes, and the arch of Figure 5.33 with *minimum* depth 7, resulting in 12665 nodes. Thus, while the former two graphs are level-threaded quadtrees with a fairly irregular and sparse shape (quadtree depth is considerably greater at input node vertex sites than in the interior of the domain), the latter, owing to the minimum depth specification, is much closer to a full quadtree. In each case we show the total number of edges cut; in all cases partitions are very well balanced, usually within 10 vertices of a perfect split. As we might expect, our method does better for the sparser (relative to size) wrenches than for the arch, and does best for the larger wrench. This reflects the number of applications of a path extended production required by the different graphs; the wrenches require fewer deepenings than the arch, and the larger wrench requires fewer deepenings relative to the number of nodes in the tree than the smaller wrench. Note that while our partitionings do not tend to be as good as the ones produced by Metis, our method is competitive, and could certainly be considered as good as or better than Jostle on sparse structures when partition sizes are reasonably large.

Partitionings of the actual grids are described in Table 5.2, and a similar pattern of results is evident. Our grid partitionings are generated by following the quadtree partitioning, simply allocating grid nodes to the appropriate containing quadtree partition, whereas both Metis and Jostle are working directly on the grid itself. Because of this, we have an extra constant factor in the number of edges we cut.

Thus, while we sometimes do better than Jostle and Metis, generally there is a factor of between 1 and 2 difference in the total number of cut edges. Our partitions are also less balanced—the difference between partition sizes varies by up to 10%-15%, though typically less than 5%; again this is a consequence of our strategy, since we do not partition the grid *within* leaf quads, only between them. If more precise partitions are required, a simple node-swapping heuristic (like Kernighan-Lin [KL70]) that splits portions of the grid within leaf quads would enable greater balance and likely fewer cuts as well.

	Wrench Quadtree 3677 nodes, 9918 edges			Wrench Quadtree 8573 nodes, 23443 edges			Arch Quadtree 12665 nodes, 37096 edges		
Ptns	*Our Alg.*	*Jostle*	*Metis*	*Our Alg.*	*Jostle*	*Metis*	*Our Alg.*	*Jostle*	*Metis*
2	113	135	57	127	595	59	511	294	289
3	133	268	106	186	660	208	546	459	520
4	271	126	129	280	502	164	977	713	663
5	308	240	231	350	506	270	998	868	827
6	340	285	204	399	495	255	1202	952	1002
7	410	246	209	487	556	390	1255	977	1015
8	452	565	255	538	826	344	1331	1136	1159
9	529	285	288	571	673	386	1380	1281	1242
10	618	377	331	577	939	390	1557	1275	1328
20	945	634	511	1014	1292	726	2238	2025	2065
30	1304	762	719	1441	1638	899	2808	2613	2485
50	1743	1155	1108	2086	1375	1333	3673	3402	3240
100	2415	1937	1943	3186	2344	2387	5183	4858	4831

Table 5.1: Comparison of total number of edges cut by our algorithm, Jostle and Metis in level-threaded quadtree partitioning.

For adaptivity, or in cases when the graph changes only slightly, our method has a distinct advantage over Metis and Jostle. Whereas post-processing methods require the entire graph as input, and the partitioning of one graph may not be easily associated with the partitioning of its adapted descendant, our method is quite well-suited to incremental approaches. Since our grammars follow the transformations of the quadtree, we can derive one partitioning from another based on the way the data structure changes as it adapts. Some results of this process for a binary partitioning on the adapting unit square of Figures 5.34 through 5.37 and on an adapting wrench are illustrated in Tables 5.3 and 5.4 using our method, and in Tables 5.5 and 5.6 using Jostle[6]. Each row is a different quadtree, generated from the specified pattern (unit square or wrench) by successive adaptations (*adapt* in the *Process* column),

[6]Jostle does perform "repartitioning," but only in the case when the number of nodes remains the same. Needless to say, this does not apply to our repartitioning problem.

	Wrench 3040 nodes, 8310 edges			Wrench 7195 nodes, 19805 edges			Arch 9451 nodes, 27962 edges		
Ptns	*Our Alg.*	*Jostle*	*Metis*	*Our Alg.*	*Jostle*	*Metis*	*Our Alg.*	*Jostle*	*Metis*
2	96	123	33	103	307	40	512	283	262
3	127	85	64	164	418	89	540	325	326
4	231	112	70	246	114	86	956	446	487
5	285	106	144	312	278	160	957	544	576
6	318	178	143	358	374	163	1191	609	653
7	375	217	168	437	345	194	1204	698	761
8	400	189	175	482	428	214	1285	812	806
9	488	226	173	528	321	235	1359	883	880
10	546	201	215	503	638	281	1479	880	965
20	826	451	410	903	913	541	2118	1407	1481
30	1180	578	527	1271	937	655	2595	1761	1872
50	1553	864	839	1871	1045	1078	3366	2379	2453
100	2096	1389	1383	2871	1866	1897	4768	3423	3504

Table 5.2: Comparison of total number of edges cut by our algorithm, Jostle and Metis in grid partitioning.

or by expanding all quadtree leaves interior to the domain one level (*push* in the *Process* column), and so we show the number of nodes, edges and total edge cuts to make the binary partition in each case. The remaining columns show the work involved in repartitioning the tree once it has been adapted or pushed; our method is integrated into tree generation, so we can show exactly how many tree nodes must be moved from one partition to the other to reflect the new partitioning. Thus, in Tables 5.3 and 5.4 we give the absolute number of nodes moved, as well as the percentage of nodes in the quadtree to move. Jostle does not directly repartition a changed graph, but we can relate the nodes in the smaller graph to the nodes in the adapted graph by preserving node identification numbers, and examining the two partitionings to calculate how many original nodes have been moved to make the new partition. Depending on how the new nodes in the adapted quadtree have been allocated, we end up with a minimum number of nodes to move, which is the total number of original nodes moved to a different partition, (*min shifted* in Tables 5.5 and 5.6) and a maximum, which is the total number of originals moved plus the number of new nodes added (*max shifted* in the tables). Actual repartitioning costs will of course be somewhere inbetween.

Our repartitioning costs are dramatically less than with Jostle. In most instances we move below 10% of the nodes; this represents a significant saving over the cost of following a new partitioning generated from scratch, which can require moving

Process	*Nodes*	*Edges*	*Cuts*	*Nodes Shifted*	
				#	%
base	85	224	23	0	0.0
adapt	193	548	40	39	20.2
adapt	657	1938	77	71	10.8
adapt	2493	7444	189	137	5.5
push	9973	29848	390	922	9.2
push	39893	119536	803	3776	9.5
push	159573	478432	1608	15168	9.5

Table 5.3: Our repartitioning costs for unit square when adapting.

Process	*Nodes*	*Edges*	*Cuts*	*Nodes Shifted*	
				#	%
base	3677	9918	113	0	0.0
adapt	8465	23672	137	4208	49.7
adapt	18301	52963	240	1878	10.3
adapt	30277	88569	229	534	1.8
adapt	43741	128154	443	413	0.9
push	167137	495744	953	1116	0.7

Table 5.4: Our repartitioning costs for wrench when adapting.

Process	*Nodes*	*Edges*	*Cuts*	*Nodes Shifted*			
				Min	%	*Max*	%
base	85	224	24	0	0.0	0	0.0
adapt	193	548	45	35	18.1	143	74.1
adapt	657	1938	63	119	18.1	583	88.8
adapt	2493	7444	132	636	25.5	2472	99.2
push	9973	29848	306	2191	22.0	9671	97.0
push	39893	119536	594	9033	22.6	38953	97.6
push	159573	478432	1315	38372	24.0	158052	99.0

Table 5.5: Jostle repartitioning costs for unit square when adapting.

Process	*Nodes*	*Edges*	*Cuts*	*Nodes Shifted*			
				Min	%	*Max*	%
base	3677	9918	135	0	0.0	0	0.0
adapt	8465	23672	212	535	6.3	5323	62.9
adapt	18301	52963	144	2970	16.2	12806	70.0
adapt	30277	88569	229	337	1.1	12313	40.7
adapt	43741	128154	325	904	2.1	14368	32.8
push	167137	495744	1213	37017	22.1	160413	96.0

Table 5.6: Jostle repartitioning costs for wrench when adapting.

upward of 90% of the nodes. In fact, our repartitionings are usually better than the *minimum* cost if we were to follow the ones produced by Jostle; only for the first adaptation of the wrench is Jostle likely (though not certainly) to result in moving fewer nodes, and this savings would be quickly offset by future adaptations. Also note how our repartitioning cost mirrors the regularity of changes to the quadtree; the first adaptation for both the square and the wrench results in large changes to the quadtree structure, and our repartitioning cost is correspondingly high. Subsequent changes, and in particular "pushes" which make the tree bigger without altering its fundamental "shape" (*i.e.,* the heaviest branches in the tree partition scheme tend to stay the heaviest after a push), result in low repartitioning costs. This permits our repartitionings to have costs less than 1% (wrench) or 10% (square) after a few adaptations, whereas costs engendered by Jostle vary wildly, though they do seem to be converging to somewhere between 20%-99% for the square.

It is therefore evident that our partitioning method is both practical and effective. While we do not consistently produce partitions of better quality than state-of-the-art heuristic approaches, our partitions are definitely competitive; even in the worst cases our partitions are roughly balanced, and communication cost is within a small factor of the costs of the ones produced by Jostle or Metis. Our method is also very fast; the production of the tree partition scheme is integrated into the graph generation process itself, meaning that the bulk of work in partitioning is amortized within the cost of data structure creation. Partitions are then generated by simple $O(n)$ tree traversals[7] of the tree partition scheme. This is an extremely fast procedure, requiring only a fraction of a second even for large graphs; Jostle and Metis, comparatively, take up to several seconds for the same graphs. For example, an arch with minimum depth 8 has 49761 nodes and 147480 edges in the quadtree structure; we partition this in 0.7 seconds, whereas Jostle takes 10.2 seconds, and Metis 7.0 seconds.

The miscegenation of the partitioning procedure and the data structure generation has another clear advantage: the partitioning of one data structure can be easily related to the partitioning of another data structure derived from the first. This allows us to partition a graph incrementally, following the incremental changes in the graph structure as it is updated (or adapted in this example). Partitioners

[7]Postorder search is linear in tree size, as is the subtree weight calculation. We do need to sort children by subtree weight for our partitioning, but since each node in our tree partition scheme has bounded degree, this is performed in constant time.

that require the complete graph as input and do not relate the partitioning of one graph to another are not able to use this information, and this can make the cost of repartitioning very expensive. When dynamic data structures are being used, this is clearly an important criterion, which has not been very well addressed by current methods.

5.9 Conclusions

As an example of the utility of our grammar-based partitioning technique, we have developed and implemented an algorithm that efficiently constructs two-dimensional triangular grids conforming to polygonal boundaries, not necessarily simply connected, with the guarantee of nonobtuseness of the triangulation. This algorithm is a non-trivial solution to the problem of grid generation for irregular domains, allowing for the construction of conforming triangulations and permitting adaptive refinements. The propagation of local changes to the grid also "decays exponentially" making our algorithm a good candidate for parallelism—small changes in the graph result in a small amount of communication.

A primary advantage of our grid generation algorithm is its ability to adapt with reasonable locality, and its relative simplicity. It requires data structures no more sophisticated than augmented quadtrees and lists, and algorithms no more complex than tree traversals. Further, and despite the large theoretical bounds on size, the algorithm has in practice been very fast and consistently produced triangulations comparable in size to other methods. The major factor dominating the size of the quadtree seems to be the vertex condition; input polygons having vertices with small binary expansions result in quite small triangulations. Given the intended application domain (grids for the finite element method in fluid dynamics), even grids containing vertices with long binary expansions can be quite useful—the physics of fluid movement suggests that placing many nodes around corners is very often desirable.

The hierarchical representation has a number of other advantages too. Checking convergence of the solution as well as computing the criterion for determining the locations of adaptations both require gathering information from all graph nodes into a global value. The quadtree provides a convenient structure for such algorithms. As well, the quadtree provides an efficient point location structure, which can be quite useful for attendant grid problems. User input, for example, is often desireable in order to inspect solution progress or to guide grid creation/management, but locating the point of a mouse click within a large unstructured grid can require other non-trivial data structures as well as the ones for grid creation and storage. The tree structure of a quadtree, coupled with the balance condition, can allow for efficient point location without any extra structure. Finally, there is some indication that the quadtree can be used for either algebraic of geometric multigrid methods on irregular domains; this remains a topic for further investigation.

To support the construction and manipulation of the grids, we use a "level-threaded" quadtree; this is an unusual variation of a well-known data structure, specifically created for our algorithm. The dynamic quality of the structure, in particular the application of adaptivity, coupled with the custom design means that a suitable partitioning strategy will be hard to find, and under normal circumstances we would be forced to develop an *ad hoc* partitioning technique for this specific problem. However, we have been able to design a dangling graph grammar mimicking the changes in the structure, and this grammar follows quite naturally from the design of the algorithm. By using this grammar in conjunction with the partitioning method developed in Chapter 3, we have demonstrated that our approach produces good quality partitions, roughly comparable to those produced by more powerful methods. Moreover, the partitionings we have evinced carry over to the actual generated grid, and are also of reasonably good quality.

To support adaptivity the data structure, and hence the partitioning, must periodically change. External partitioning techniques, ones applied after the fact with little or no knowledge of how the data structure changes, perform poorly in these situations; since they do not relate one partitioning to another, they can require drastic changes to the way the data structure has been divided up. The grammar-based approach we use is considerably more effective in these cases; small changes, and even dramatic changes that do not alter the fundamental "shape" of the structure, result in a correspondingly limited repartitioning cost. Depending on the manner in which a data structure changes, being able to repartition without having to move too many nodes around can be very important, and must be factored into the selection of a partitioning strategy. An exact partitioning, with minimum communication cost and perfect load-balancing is of little use in a dynamic situation if each time the data structure mutates a completely new partitioning must be generated, possibly requiring most or all nodes to be shifted from one processor to another. The speed of the partitioning also becomes critical in such situations; even if communication is quite cheap, an expensive partitioning algorithm can result in a great deal of overhead if the data structure changes. Our algorithm is very fast, requiring only a linear-size tour through the nodes to produce partitions. Even the extra cost of modelling the grammar as the data structure is manipulated is quite small—a constant-bounded amount of work each time the data structure is altered. Our method is therefore well-suited to the combination of constraints on dynamic, irregular problems: it produces partitions competitive in quality to more direct methods, it can efficiently deal with local updates, and it is fast.

Chapter 6

Related Work

Of course the problem of using graph partitioning to facilitate parallelism has been investigated before, and many aspects predate parallel computing. The following sections contextualize the work presented in the previous chapters; first we discuss general efforts at graph partitioning. The next section expounds on other "linguistic" characterizations of graphs, and situates our particular form of graph grammar in relation to existing classes of graph grammars. The language **eL** is a new parallel language based on these grammars, so we also compare **eL** to other parallel and graph grammar-based languages. Finally, in Section 6.4, we describe various non-obtuse grid generation algorithms.

6.1 Partitionability

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 [LT79] 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 [SV93] and other variations have been explored [BP92, DDSV93] 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 [Luk74]. 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 [KM77]. While these algorithms are interesting and have some similarity to our tree partitioning algorithm of Section 3.4, we do not necessarily require connectivity, and we do require there to be a given number of partitions. Other types of tree partitioning have been considered which are less relevant to our goals. Tarjan and Misra [MT75] give an algorithm for finding an optimal *chain partition* of a tree, where a chain partition is a collection of edges with the property that no node n has more than one child c with (n, c) in the collection. Frederickson gives optimal algorithms for removing a given number of edges in order to minimize the maximum, or maximize the minimum component weight [Fre91].

Heuristic attacks on partitioning problems abound. Perhaps the most well-known is the Kernighan-Lin heuristic [KL70]: 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 [FM82]. Feo and Khellaf have developed a heuristic based on a recursive pair-wise grouping of nodes for k-partitioning when k is large [FK90]. More recent methods include Spectral Bisection [HL93], Simulated Annealing [JAMS89], and others; heuristic combinations of such methods have also been quite successful [DLMS95, KK95, WCE$^+$95]. We of course are interested in deterministic methods, though we have compared our partitionings to some heuristic ones and found our results roughly comparable. 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 [SV95]. While polynomial ($O(n^p)$ for some p), direct methods such as these are still too expensive for our purposes.

Data partitioning explicity for irregular problems has also been explored. Nakanishi *et al.* [NJS$^+$94], for instance, develop a "Heirarchical Data Partitioning" graph, incorporating a hierarchical representation of control flow and control-flow dependencies. This is a general model for partitioning that does not attempt to utilize any structure-specific information, and does not include cost bounds. Naturally, better results can be obtained if even general characteristics of such algorithms are known; this is the approach taken by Gautier, Roch and Villard [GRV95]. They identify several programming paradigms for dealing with irregular problems in order to produce a classification scheme. Their efforts are meant to facilitate either manual or automatic load-balancing and not to actually specify such algorithms. Das, Moser and Melliar-Smith [DMMS95] also give a generic approach through the presentation of hardware designed to support irregular data accesses, called the "Intersecting Broadcast Machine." By distributed data randomly, and maintaining multiple copies of data through broadcasts, they can show (experimentally) very good load-balancing and processor utilization; however, their method is stochastic, rather than deterministic. Alternatively, there are many algorithms for tackling specific problem areas: backtracking search trees [San95], Finite Element Methods on irregular domains [BK95, DMM95, GWZ95, WCE$^+$95], particle systems [MP95], *etc.* Most of these make use of heuristics or randomized techniques, such as greedy graph clustering [Far88], simulated annealing and recursive bisection.

6.2 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 have argued 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, Hendren and Nicolau [HHN92] 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 [Gup92], 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 [KS93a] 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.

6.3 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 [No.90] 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 [BEHL86, Ehr87, EBHL88, EKL90, LE91], 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 [Cou90a, Cou90b] 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 and Rozenberg [JR80a]. 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 [JR80b], and have used this as the basis for an expressiveness heirarchy [JRV82, JRV83].

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 [JR82a, JR82b]. 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: $\mathrm{NLC} = 1 - \mathrm{NCE} \subset \mathrm{NCE}$ [JR82b]. 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 [ER90]. "dNCE" extends NCE to directed graphs [JR81], and "eNCE" includes edge-labels into NCE [Bra86, ELW90]; the combination, "edNCE", having both. The most useful extension seems to be "C-edNCE", or *Confluent* edNCE grammars [Eng89, Eng90]. 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 [Cou87]). 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 [Eng90], "Separated Handle-Rewriting Hypergraph Grammars" (S-HH) [CER90], 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[1] to (hyper)graphs. (Hyper)edge rewriting grammars have been primarily investigated by Kreowski [HK86, DK90], Lautemann [Lau90b, Lau90a] and Courcelle [Cou90a, Cou90b], 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."

There have been attempts to reconcile the two approaches. For instance, the schematic formalization of graph grammars developed by Kreowski and Rozenberg [KR90a, KR90b] 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

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

grammars are expressively equivalent to C-edNCE [CER90, CER93], 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 [EH94].

Our grammars of Chapter 3 were designed for two reasons; 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 [Sli82]. However, there are many important differences, such as the use of 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 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 [Lin68, PH92].

6.3.1 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 [OH91], solid modeling for CAD/CAM systems [Fit86], compiler generation [Hof82], and as a syntax for visual representations [Rek94].

There have also been several languages 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.* [NEGS82]. 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 [Sch89, Sch90c]. 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.* [BvEG+87] 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 [AE93, AP91, FV82], but again these contain constructs which make partitioning difficult.

Specifically for parallel applications, Janssens and Rozenberg [JR90] 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" [BS93], and Glauert, Kennaway and Sleep have developed "Dactl" as a graph grammar-based common target language for a number of other languages [GKS90]. 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.

In Chapter 4 we developed a visual environment for editing and viewing our graph grammars. Our environment is unique in the incorporation of the control structure; however, there are many other environments for editing graphs and graph grammars, and the number of interactive graph drawing tools is of course legion. Bailey and Cuny, for instance, have developed graphical editing methods based on graph grammars for designing parallel communication structures [BC86], and also the "ParaGraph" editor for specifying very large graphs [BC90]. "Graph$^{\mathrm{Ed}}$" is another graph grammar editor/viewer designed by Himsolt [Him90], which allows for the interactive visual editing of graphs and also specification and execution and of some types of NLC grammars. The IPSEN project, and PROGRESS as well, both specify an editing environment, though entirely textual in nature [NS90, Sch90a, Sch90b].

Although the number of graph grammar languages and environments is not large, there are still remarkably few complexity analyses of implementations of graph grammars systems. Because of the NP-complete subgraph isomorphism problem usually associated with matching the rules to the (iterated) axiom, graph grammars have typically been regarded as entirely theoretical, and practical considerations have not been addressed. Bunke, Glauser and Tran [BGT90] provide one of the few attempts to optimize the matching process, by noting that most of the graph, and hence possible locations for rule applications, does not change after a production is applied. This does not give them a better worst-case complexity, but it does give a general improvement, and experimental evidence is given. A more generic approach is given by Dörr [Dör94, Dör95], where a general, efficient *Graph Rewriting Abstract Machine* (GRAM) is defined. Dörr specifies conditions under which efficient graph rewriting can occur; if the rewrite system can match subgraphs in constant time, then the grammar can be efficienty executed by GRAM.

6.4 CVFEM Grid Generation

Till recently the bulk of the work on generation of unstructured grids has been either heuristic, or has concentrated on generating grids satisfying the Delaunay criterion. A paper by Bern and Eppstein [BE92] provides an exhaustive survey of the field, particularly as it relates to the finite element method in fluid dynamics.

Techniques for generating bounded-size grids over arbitrary polygons (and without obtuse angles) have only begun to appear in the last few years; Baker, Grosse and Rafferty [BGR88] being perhaps the very first to provide a provably correct algorithm. Their efforts were focussed on establishing the existence of an algorithm rather than on demonstrating its usefulness or feasibility in practice. They did not establish any bounds on the size of the triangulations that they generate nor did they implement their algorithm. Despite these criticisms this paper clearly opened up a new set of possibilities and demonstrated the existence of triangulations based on subtle ideas from modern computational geometry.

The papers by Bern, Eppstein and Gilbert [BEG90, BEG94] describe a family of related algorithms that cover a variety of cases. In three of them they work with given point sets rather than with given regions. This means that the boundary of the region is the convex hull of the point set and is thus much more tractable than the regions we work with. In one case they guarantee no small angles, but some of the angles could be obtuse; in the other two cases they avoid obtuse angles.

The basic tradeoff is between the number of triangles and the minimum angle. They also give an algorithm that triangulates polygonal regions (but which does not guarantee no obtuse angles), and some higher dimensional algorithms with similar guarantees. In sum, while these are very interesting algorithms from the point of view of computational geometry, they do not have all the requirements that one needs in practice.

In another paper by Bern and Eppstein [BE91] they develop an $O(n^2)$ triangulation which is guaranteed to contain no obtuse angles. The regions could have holes and certainly need not be convex so this is general enough for most applications. It is not clear, however, how one could modify the algorithm to make it adaptive.

Several authors have also investigated the use of quadtrees in mesh generations, both heuristically (Yerry and Shephard [YS83]), and deterministically (Bern, Eppstein and Gilbert [BEG94]). The former use quadtrees to essentially tile the domain with some (fixed) number of patterns which can then be triangulated, while the latter extend an (other) algorithm presented in [BGR88] to triangulate point sets with no obtuse angles.

To triangulate polygons (with holes) with no small angles and no obtuse angles, Melissaratos and Souvaine [MS92] have developed an algorithm which first locates a rectilinear polygon inscribed inside the original domain, triangulates this region and the remainder of the polygon separately, and then patches the triangulation to make the two tilings consistent. They also use quadtrees to reduce the size of the resulting triangulation in a manner similar to Bern, Eppstein and Gilbert's 1990 algorithm for non-obtuse triangulation of point sets [BEG90], reaching $O(nM)$ triangles where M depends on the input geometry.

Bern, Dobkin and Eppstein [BDE92] describe several other algorithms for quality triangulations. They maximize the minimum height of triangles within a polygon using $O(n)$ triangles, ensure the largest angle is no larger than $150°$ using $O(n \log n)$ triangles (simple polygon) and $O(n^{3/2})$ triangles (polygon with holes) and also generate a nonobtuse triangulation of convex polygons with $O(n^{1.85})$ triangles.

Very recently Bern, Mitchell and Ruppert [BMR94] announced a linear-size nonobtuse triangulation of polygons, finally satisfying the longstanding problem. This algorithm uses a very ingenious circle-packing scheme and is recursive in character. Unfortunately, it is not clear how feasible this algorithm would be for an adaptive grid—the effects of replacing a given circle with some number of smaller ones does not have an easily determined effect. As well, the computation of the "generalized Voronoï polygon" is required as substep; efficient algorithms exist for this structure, but the programming effort is daunting, and even they resort to a heuristic instead. Thus, despite being a major accomplishment in grid generation, their algorithm is not practically suitable for our purposes.

Chapter 7

Conclusions

We have illustrated a general method for partitioning irregular data structures for parallelism. By translating the data structure updates into graph grammar rules in our formalism, the resulting structure is necessarily partitionable. This allows a parallel version of the algorithm in question to be executed with guaranteed bounds on communication cost, and with well-balanced partitions. The production of these partitions is also quite fast, and can follow incremental changes in the structure; these are especially important qualities when the structure is dynamically changing.

Our method applies to only a subset of all possible graphs. This is intended; the general partitioning problem applied to arbitrary graphs is NP-complete, and so we have no hope of finding efficient solutions for all inputs. However, by restricting our input to the graphs typically employed in applications using dynamic data structures, we are able to show reasonable bounds on communication cost. Moreover, our method does not constrain the input precisely—small variations in structure can be easily accomodated: a tree remains fundamentally a tree even if a few edges between siblings are added, or if recursive threading is introduced, and such changes can be directly incorporated into the corresponding grammar, and hence into our partitionings. A partitioning method specific to trees will obviously be stymied by such alterations, and heuristic methods will usually fail to identify the underlying structure and default to more general methods. This makes our approach quite viable, particularly when the data structure is a slightly altered version of a more common data structure.

Such flexibility allows us to design a complete programming language around the graph grammar structure. In this way both partitioning and computation can coexist in the same formalism, making the development of irregular applications for parallelism considerably more straightforward. Our current implementations are naturally prototypical, but they do indicate that any practical problems to the implementation of our ideas are surmountable. They also give an effective context for exploring graph grammars and the data structures and algorithms they can represent; our graphical representation (Tuna) constitutes a novel representation of computation. Tuna incorporates control flow into the visual paradigm of computation given by graph grammars, and this allows us to provide a seamless graphical environment for the generation and manipulation of data structures, eliminating much of the tedious and error-prone aspects of a textual description of graphs and graph updates. Our language also has the quality that graph rewrites represent atomic data structure changes; whereas procedural languages like C must move and create pointers over several sequential steps, which can cause the data structure to be momentarily in an inconsistent state, we can mutate the data structure and update pointers in a single indivisible step. This quality can be very important to analyses trying to determine more about the nature of the data structure being used.

The utility and efficacy of our method is established by example. We have shown small, but indicative, grammars illustrating how a number of data structures can be expressed in our formalism. The considerably larger example of adaptive grid generation for the finite element method on irregular domains provides a more practical explication of the process; we have shown how to take an irregular problem, including a pertinent (to the problem) though non-standard data structure, and use our system to determine the partitionings. Experimental measurements verify the quality of the resultant partitionings: the partitionings we generate are reasonably-good, and are certainly competetive with state-of-the-art heuristic approaches.

Our method has a distinct advantage over a post-processing approach to graph partitioning. The grammar formalism is integrated with the data structure manipulations themselves; this means the partitioning we generate for a given data structure can be easily related to the partitioning we generate once the structure has been updated in some local way. An "external" partitioning strategy does not make this kind of association, and produces each partitioning with no reference to any of its previous work. As we have demonstrated, this can result in quite drastic changes in the partitioning, and may cause the entire data structure to be shuffled between processors. Our approach, on the other hand, directly accommodates incremental changes to the data structure—the number of data structure nodes we move between partitions is generally quite small, and in proportion to the severity of the changes in the structure itself. This is clearly a very important quality for dynamic problems.

The time required to produce the partitioning can also be a critical factor when working with a dynamic algorithm. The heuristic techniques we have compared our work against are very fast, usually requiring only a few seconds, even for graphs with upward of 10^5 vertices. Even so, our system is easily an order of magnitude faster, requiring less than a second for the same large graphs. Of course, some of our cost is amortized within the data structure construction itself, with each such step adding (at most) a constant amount of extra work to the construction. Altogether, though, this extra effort still only constitutes as much work as another linear traversal of the associated tree partition scheme. When the graph changes dynamically, the speed of our technique, combined with a reduced repartitioning cost makes our approach especially efficient.

We have therefore developed a complete general system for working with irregular, dynamic problems. The partitions we produce are deterministically generated by the actions of the grammar or data structure updates; this means we can immediately produce partitions without recourse to iterative-converging or randomized methods. This makes our method quite effective, and not prone to the occassional bad partition which heuristics can sometimes produce. At the same time, our method is quite flexible, accommodating a great deal of variation in structure—we can easily use a custom-designed structure derived from a more well-known one, without being forced to resort to more general methods. This carries over to data structure alterations; while other methods are applied *post facto,* and so cannot take advantage of the nature of the changes applied to a data structure, the partitionings produced by our method reflect the way the data structure was constructed. Partitioning each graph—the original data structure and its derivation by updates—separately can result in large data movements, whereas our method has a very low repartitioning cost. Similar arguments apply to the time required for partitioning; the integration of our method with data structure construction, together with the simple tree traversals for generating actual partitions results in a very fast partitioning strategy. It is, however, the combination of all these qualities that makes our methodology so appropriate; irregular problems often require specialized data structures which change dynamically. By illustrating a method which incorporates the ability to deal with a wide variety of irregular structures, and which efficiently manages dynamic updates with a fast partitioning algorithm, we have shown a general solution to this very difficult problem.

7.1 Future Work

The partitioning techniques described here are the basis for producing an efficient parallel implementation of an algorithm. It would be interesting to see how this is actually born out in practice. An implementation of the grid generation algorithm on parallel hardware, where work is divided up according to our partitioning scheme would provide this sort of information; the speed-up with respect to the number of processors could be measured and compared with the speed-up using other partitioning techniques.

The language **eL** has only been defined using our basic graph grammars. The incorporation of path-extended grammars remains to be done, as well expanding the language to include an input/output mechanism. At the moment I/O is performed by exploiting the fact that **eL** is compiled to C; a more rational framework might include special "input" and "output" nodes which produce or consume attached data nodes.

The dangling graph grammars and associated overlap properties we have defined indicate several interesting areas for further research. For instance, we require the grammar be such that the resulting derivation maps onto a tree, with appropriate disconnection properties; similar results using much the same techniques could ensure that the derived graphs map onto other well-known structures, such as rectangular grids. This might provide a means to partition more complex graphs without the generality of using path-expressions. Our formalism could also be extended to handle graphs with nodes without fixed degree, or to graphs with weighted edges.

Appendix A

eL Programs

The following is an example of **eL**-code. This program receives a sequence of integers as input, sorts them and outputs the sorted list. The algorithm works by constructing a binary search tree from the input data.

A.1 Sorting Program

```
/* Sort numbers using a parallel threaded binary search tree.
   The algorithm is as follows:  A root node received inputs,
   and passes them down to its children.  As the value descends
   the tree, it branches appropriately based on its value.  When
   it reaches a terminal (external) node, that terminal is expanded
   to an internal node and two terminals, with the value stored at
   the new internal node.  Input comes in every other time step,
   and is followed by an end-of-input marker.

   The end-of-input marker is propagated down all branches, and
   converts terminal nodes to TerminalDone nodes.  Once all such
   nodes have been converted, all of the initial input to the tree
   must have finished its journey down the tree, and the sorted
   output can be read off. */

// --- Node Types -------------------------------------------
```

```
node Input {       // The main input node
    link e;        // Link to binary search tree
}

node InputIn {     // Input node carrying a value
    node Input;
    int value;     // Last input value
}

node Wait {        // For waiting to flush the tree after all input
    node Input;    //  has been completed
}

node Elt {         // Nodes within the tree
    int value;     // Value held in node
    link p,l,r;    // Links to parent, left and right children
    link left,right; // inorder threading
}

node EltDown {     // For descending values
    node Elt;      // Same as an Elt
    int down;      // Except for having a descending value
}

node Terminal {    // External nodes
    link p;        // Link to parent
    link left,right;  // For threading
}

node TerminalDone { // External node that's done
    node Terminal;
}

node Begin {       // Beginning of the threading
```

```
    link right;
}

node End {          // End of the threading
    link left;
}

node Potato {       // A 'hot-potato' used to sequentially output
    link left,      //  the sorted list
         right,
         p,l,r;
}

// --- Graph Types --------------------------------------------

graph Initial {     // Axiom
    Input i;        // Just an input node
    Terminal t;     // connected to an external node
    Begin b;        // and threaded between a begin and end node
    End e;
    link i.e,t.p;
    link t.left,b.right;
    link t.right,e.left;
}

graph TerminalRead { // For the very first read
    InputIn i;
    Terminal t;
    link i.e,t.p;
}

graph RootRead {    // For the root of the tree to read from input
    InputIn i;
    Elt e;
    link i.e,e.p;
}

graph OneElt {      // An element and two terminals from a terminal
    Elt e;
    Terminal tl,tr;
    link e.l,tl.p;
    link e.r,tr.p;
}

graph TinyTree {    // An element and two terminals from a terminal
    Elt e;
    Terminal tl,tr;
    link e.l,tl.p;
    link e.r,tr.p;
    link tl.right,e.left; // threading
    link e.right,tr.left;
}

graph G_EltDown {   // An Element with a descending node
    EltDown e;
}

graph G_Elt {       // An Element
    Elt e;
}

graph G_Input {     // An input node
    Input i;
}

graph G_InputIn {   // An input node with a value
    InputIn i;
}
```

```
graph G_Wait {     // An input node waiting to flush the network
    Wait w;
}

graph G_Terminal { // Just a terminal node
    Terminal t;
}

graph G_TerminalDone { // A terminal node received the flush signal
    TerminalDone t;
}

graph G_Potato {   // hot potato, for printing list results
    Potato p;
}



// --- Blocks -------------------------------------------------

// A block just to print a carriage return
block PrintCR : -> {
    @printf("\n");@ ;
}

// Printing out the sorted list.  Terminates when the hot-potato
// reaches the end of the threading
block PrintList : graph {
                     Potato p;
                     End e;
                     link p.right,e.left;
                  } -> {

    /* Begin process */
    rule : graph {
              Begin b;
           } by b -> G_Potato (p.right=b.right) ;

    rule : graph {
              Elt e;
              Potato p;
              link e.left,p.right;
           } by e -> G_Potato (p.link=e.link)
           @ printf("%d ",elf_src->e->value); @ ;

    rule : graph {
              Terminal t;
              Potato p;
              link t.left,p.right;
           } by t -> G_Potato (p.link=t.link) ;
}

// The actual printing routine.  Only action is to revert
// TerminalDone nodes (which have been fully-flushed) to
// Terminal nodes.  Then outputs the ordered list, followed
// by a carriage-return
block Print : -> PrintCR PrintList {

    // Convert TerminalDone's to Terminals
    rule : G_TerminalDone by t -> G_Terminal (t=t) ;
}

// Actual sorting algorithm.  Builds the binary search tree until
// all input has been received, and fully-filtered through the tree
// (ie, the tree has been flushed).
block Sort : !graph { Terminal t; } -> {

    // Rule to begin the tree
```

```
rule : TerminalRead by t @elf_src->i->value>0@ ->
       TinyTree (e.=i.) (e.p=t.p) (tl.left=t.left)
                (tr.right=t.right);

// Letting the root read
rule : RootRead by e ->
       G_EltDown (e=e)
       @elf_tar->e->down = elf_src->i->value;@ ;

// Discarding the descending value
rule : G_EltDown by e ->
       G_Elt (e=e) ;

// Rules to inherit descending value from parent
// From an Elt to an Elt
rule : graph {
          EltDown d;
          Elt e;
          link d.l,e.p;
       } by e @elf_src->d->down==0 ||
               elf_src->d->down<=elf_src->d->value@ ->
       G_EltDown (e=e)
       @elf_tar->e->down = elf_src->d->down;@ ;

rule : graph {
          EltDown d;
          Elt e;
          link d.r,e.p;
       } by e @elf_src->d->down==0 ||
               elf_src->d->down>elf_src->d->value@ ->
       G_EltDown (e=e)
       @elf_tar->e->down = elf_src->d->down;@ ;

// From an Elt to a Terminal
```

```
rule : graph {
          EltDown e;
          Terminal t;
          link e.l,t.p;
       } by t @elf_src->e->down!=0 &&
               elf_src->e->down<=elf_src->e->value@ ->
       TinyTree (e.p=t.p) (tl.left=t.left) (tr.right=t.right)
       @elf_tar->e->value = elf_src->e->down;@ ;

rule : graph {
          EltDown e;
          Terminal t;
          link e.r,t.p;
       } by t @elf_src->e->down!=0 &&
               elf_src->e->down>elf_src->e->value@ ->
       TinyTree (e.p=t.p) (tl.left=t.left) (tr.right=t.right)
       @elf_tar->e->value = elf_src->e->down;@ ;

// Rules to read input
rule : G_InputIn by i @elf_src->i->value>0@ -> G_Input (i=i) ;
rule : G_InputIn by i @elf_src->i->value==0@ -> G_Wait (w=i) ;

rule : G_Input by i -> G_InputIn (i.link=i.link)
       @if(!scanf("%d",&(elf_tar->i->value)))
           elf_tar->i->value = 0; @ ;

// Rules for flushing the network
// From an Elt to a Terminal
rule : graph {
          EltDown e;
          Terminal t;
          link e.l,t.p;
       } by t @elf_src->e->down==0@ ->
       G_TerminalDone (t=t) ;
```

```
    rule : graph {
              EltDown e;
              Terminal t;
              link e.r,t.p;
           } by t @elf_src->e->down==0@ ->
           G_TerminalDone (t=t) ;
}

// Axiom declaration
axiom Initial;

// Declaration for which block begins the computation
start Sort;
```

Bibliography

[AE93] Marc Andries and Gregor Engels. Syntax and semantics of hybrid database languages. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science: Proceedings of the International Workshop*, number 776 in Lecture Notes in Computer Science, pages 19–36, Dagstuhl Castle, Germany, January 1993. Springer-Verlag.

[ALS88] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Problems easy for tree-decomposable graphs. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, pages 38–51, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[AP91] Marc Andries and Jan Paredaens. A language for generic graph-transformations. In G. Schmidt and R. Berghammer, editors, *Graph-Theoretic Concepts in Computer Science: Proceedings of the 17th International Workshop, WG '91*, number 570 in Lecture Notes in Computer Science, pages 63–74, Fischbachau, Germany, June 17–19 1991. Springer-Verlag.

[Arn85] Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT*, 25(1):2–23, 1985.

[AST94] Noga Alon, Paul Seymour, and Robin Thomas. Planar separators. *SIAM Journal on Discrete Mathematics*, 7(2):184–193, May 1994.

[Aur91] Franz Aurenhammer. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, San Francisco, California, January 17–19, 1990. ACM SIGACT and SIGPLAN.

[BC86] Duane A. Bailey and Janice E. Cuny. Graph grammar based specification of interconnection structures for massively parallel computation. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 73–85, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[BC90] Duane A. Bailey and Janice E. Cuny. Programming with very large graphs. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 84–97, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[BDE92] Marshall Bern, David Dobkin, and David Eppstein. Triangulating polygons without large angles. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, pages 222–231, June 1992.

[BE91] Marshall Bern and David Eppstein. Polynomial-size nonobtuse triangulation of polygons. In *Proceedings of the 7th ACM Symposium on Computational Geometry*, pages 342–350, 1991.

[BE92] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. In F.K. Huang, editor, *Computing in Euclidian Geometry.* World Scientific, 1992.

[BEG90] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, pages 231–241, 1990.

[BEG94] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. *J. Comput. System Sci.*, 48:384–409, 1994.

[BEHL86] Paul Boehm, Hartmut Ehrig, Udo Hummert, and Michael Löwe. Towards bistributed graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 86–98, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[BGR88] Brenda S. Baker, Eric Grosse, and Conor S. Rafferty. Nonobtuse triangulation of polygons. *Discrete Comput. Geom.*, 3:147–168, 1988.

[BGT90] H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 174–189, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[BH86] Joshua E. Barnes and Piet Hut. A heirarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[BK95] George Horatiu Botorog and Herbert Kuchen. Algorithmic skeletons for adaptive multigrid methods. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 27–41, Lyon, France, September 4–6 1995. Springer-Verlag.

[BMR94] Marshall Bern, Scott Mitchell, and Jim Ruppert. Linear-size nonobtuse triangulation of polygons. In *Proceedings of the 10th ACM Conference on Computational Geometry*, pages 221–230, 1994.

[Bod88] Hans L. Bodlaender. NC-algorithms for graphs with small treewidth. In J. van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science: Proceedings of the International Workshop, WG '88*, number 344 in Lecture Notes in Computer Science, pages 1–10, Amsterdam, The Netherlands, June 15–17 1988. Springer-Verlag.

[Bow81] A. Bowyer. Computing Dirichlet tesselations. *Comput. J.*, 24(2):162–166, 1981.

[BP92] Thang Nguyen Bui and Andrew Peck. Partitioning planar graphs. *SIAM Journal on Computing*, 21(2):203–215, April 1992.

[Bra83] Franz J. Brandenburg. The computational complexity of certain graph grammars. In A. B. Cremers and H. P. Kriegel, editors, *Theoretical Computer Science: 6th GI-Conference*, number 145 in Lecture Notes in Computer Science, pages 91–99, Dortmund, West Germany, January 1983. Springer-Verlag.

[Bra86] Franz J. Brandenburg. On partially ordered graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 99–111, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[BS93] Klaus Barthelmann and Georg Schied. Graph-grammar semantics of a higher-order programming language for distributed systems. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science: Proceedings of the International Workshop*, number 776 in Lecture Notes in Computer Science, pages 71–85, Dagstuhl Castle, Germany, January 1993. Springer-Verlag.

[BvEG$^+$87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards and intermediate language based on graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *Proceedings of PARLE – Parallel Architectures and Languages Europe*, volume I of *Lecture Notes in Computer*

Science 258–259, pages 159–174, Eindhoven, The Netherlands, June 15–19, 1987. Springer-Verlag.

[CER90] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Context-free handle-rewriting hypergraph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 253–268, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[CER93] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-rewriting hypergraph grammars. *Journal of Computer and System Sciences*, 46:218–270, 1993.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course.* MIT PRess, Cambridge, MA, 1990.

[CGL86] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT and SIGPLAN.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169, October 1988.

[CL93] Bruno Courcelle and Jens Lagergren. Recognizable sets of graphs of bounded tree-width. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science: Proceedings of the International Workshop*, number 776 in Lecture Notes in Computer Science, pages 138–152, Dagstuhl Castle, Germany, January 1993. Springer-Verlag.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press; McGraw-Hill Book Company, Cambridge, Massachusetts; New York, New York, 1990.

[Cou87] Bruno Courcelle. An axiomatic definition of context-free rewriting and its application to NLC graph grammars. *Theoretical Computer Science*, 55:141–181, 1987.

[Cou90a] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 195–241. Elsevier, Amsterdam, 1990.

[Cou90b] Bruno Courcelle. The logical expression of graph properties (abstract). In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 38–40, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[DDSV93] Krzystof Diks, Hristo N. Djidjev, Ondrej Sýkora, and Imrich Vrťo. Edge separators of planar and outerplanar graphs with applications. *Journal of Algorithms*, 14:258–279, 1993.

[DH73] W. E. Donat and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, September 1973.

[DK90] Frank Drewes and Hans-Jörg Kreowski. A note on hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 1–11, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[DLMS95] R. Diekmann, R. Lüling, B. Monien, and C. Spräner. A parallel local-search algorithm for the k-partitioning problem. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS '95)*, volume 2, pages 41–50, 1995.

[DMM95] Ralf Diekmann, Derk Meyer, and Burkhard Monien. Parallel decomposition of unstructured fem-meshes. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems:*

Proceedings of the Second International Workshop, IRREGULAR '95, number 980 in Lecture Notes in Computer Science, pages 199–215, Lyon, France, September 4–6 1995. Springer-Verlag.

[DMMS95] A. Das, L.E. Moser, and P.M. Melliar-Smith. A parallel processing paradigm for irregular applications. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 249–254, Lyon, France, September 4–6 1995. Springer-Verlag.

[Dör94] Heiko Dörr. An abstract machine for the execution of graph grammars. Technical Report B-94-07, Freie Universität Berlin, Takustraße 9, D-14195 Berlin, 1994.

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation.* Number 922 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[EBHL88] Hartmut Ehrig, Paul Boehm, Udo Hummert, and Michael Löwe. Distributed parallelism of graph transformations. In H. Gottler and H. J. Schneider, editors, *Proceedings of the 13th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '87)*, number 314 in Lecture Notes in Computer Science, pages 1–19. Springer-Verlag, July 1988.

[EH94] Joost Engelfriet and Linda Heyker. Hypergraph languages of bounded degree. *Journal of Computer and System Sciences*, 48:58–89, 1994.

[Ehr87] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rozenberg, editors, *Proceedings of the 3rdInternational Workshop on Graph-Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 3–14. Springer-Verlag, December 1987.

[EKL90] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 24–37, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[EKR90] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[ELW90] J. Engelfriet, G. Leih, and E. Welzl. Boundary graph grammars with dynamic edge relabeling. *Journal of Computer and System Sciences*, 40:307–345, 1990.

[EMR82] Hartmut Ehrig, Manfred Magl, and Grzegorz Rozenberg, editors. *Proceedings of the 2nd International Workshop on Graph Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, Haus Ohrbeck, West Germany, October 4–8, 1982. Springer-Verlag.

[Eng89] Joost Engelfriet. Context-free NCE graph grammars. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT '89)*, number 380 in Lecture Notes in Computer Science, pages 148–161, Szeged, Hungary, August 1989. Springer-Verlag.

[Eng90] Joost Engelfriet. A characterization of context-free NCE graph languages by monadic second-order logic on trees. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 311–327, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[ENRR86] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in

Computer Science, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[ER90] Joost Engelfriet and Grzegorz Rozenberg. Graph grammars based on node rewriting: An introduction to NLC graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 12–21, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[Far88] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers & Structures*, 28(5):579–602, 1988.

[Fit86] Patrick Fitzhorn. A linguistic formalism for engineering solid modeling. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 202–215, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[FK90] Thomas A. Feo and Mallek Khellaf. A class of bounded approximation algorithms for graph partitioning. *Networks*, 20:181–195, 1990.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[FR95] Afonso Ferreira and José Rolim, editors. *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, Lyon, France, September 4–6 1995. Springer-Verlag.

[Fre91] Greg N. Frederickson. Optimal algorithms for tree partitioning. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 168–177, San Francisco, California, January 1991.

[FV82] A.L. Furtado and P.A.S. Veloso. Specification of data bases through rewriting rules. In Hartmut Ehrig, Manfred Magl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph*

Grammars and Their Application to Computer Science, number 153 in Lecture Notes in Computer Science, pages 102–114, Haus Ohrbeck, West Germany, October 4–8, 1982. Springer-Verlag.

[Geo91] P. L. George. *Automatic Mesh Generation: Application to Finite Element Methods*. John Wiley & Sons, Inc., 1991.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freemann and Co., New York, New York, 1979.

[GKS90] J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep. Dactl: An experimental graph rewriting language. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 378–395, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[GRV95] T. Gautier, J.L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 1–25, Lyon, France, September 4–6 1995. Springer-Verlag.

[Gup92] Rajiv Gupta. SPMD execution of programs with dynamic data structures on distributed memory machines. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 232–241, Oakland, California, April 20–23, 1992. IEEE Computer Society Press.

[GWZ95] P.W. Grant, M.F. Webster, and X. Zhang. Solving computational fluid dynamics problems on unstructured grids with distributed parallel processing. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 187–197, Lyon, France, September 4–6 1995. Springer-Verlag.

[HHN92] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[Him90] Michael Himsolt. Graph$^{\text{Ed}}$: An interactive tool for developing graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 61–65, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[HK86] Annegret Habel and Hans-Jörg Kreowksi. May we introduce to you: Hyperedge replacement. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 15–26, Warrenton, Virginia, December 2–6, 1986. Springer-Verlag.

[HL93] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical Report SAND93-0074, Sandia National Laboratory, January 1993.

[Hof82] Berthold Hoffmann. Modelling compiler generation by graph grammars. In Hartmut Ehrig, Manfred Magl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, pages 159–171, Haus Ohrbeck, West Germany, October 4–8, 1982. Springer-Verlag.

[Jac86] Manfred Jackel. ADA concurrency specified by graph grammars. In Gottfried Tinhofer and Gunther Schmidt, editors, *Proceedings of the 12th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '86)*, number 246 in Lecture Notes in Computer Science, pages 41–57, Bernried, West Germany, June 17–19, 1986. Springer-Verlag.

[JAMS89] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part 1: Graph partitioning. *Operations Research*, 37(6):865–893, 1989.

[JR80a] D. Janssens and G. Rozenberg. On the structure of node-label-controlled graph languages. *Information Sciences*, 20:191–216, 1980.

[JR80b] D. Janssens and G. Rozenberg. Restrictions, extensions and variations of NLC grammars. *Information Sciences*, 20:217–244, 1980.

[JR81] D. Janssens and G. Rozenberg. A characterization of context-free string languages by directed nodel-label controlled graph grammars. *Acta Informatica*, 16:63–85, 1981.

[JR82a] D. Janssens and G. Rozenberg. Graph grammars with neighbourhood controlled embedding. *Theoretical Computer Science*, 21:55–74, 1982.

[JR82b] D. Janssens and G. Rozenberg. Graph grammars with node-label controlled rewriting and embedding. In Hartmut Ehrig, Manfred Magl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, pages 186–203, Haus Ohrbeck, West Germany, October 4–8, 1982. Springer-Verlag.

[JR83] D. Janssens and G. Rozenberg. Hypergraph systems generating graph languages. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, pages 172–185. Springer-Verlag, October 1983.

[JR90] D. Janssens and G. Rozenberg. Structured transformations and computation graphs for actor grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 446–460, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[JRV82] D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars. *Computer Graphics and Image Processing*, 18:279–304, 1982.

[JRV83] D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars, II. *Computer Vision, Graphics, and Image Processing*, 23:295–312, 1983.

[Ken87] Richard Kennaway. On "On graph rewritings". *Theoretical Computer Science*, 52:37–58, 1987.

[KK95] George Karypis and Vipin 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.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[Klo94] Tom Kloks. *Treewidth: Computations and Approximations.* Number 842 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[KM77] Sukhamay Kundu and Jayadev Misra. A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1):151–154, March 1977.

[KR90a] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars I. *Information Sciences*, 52:185–210, 1990.

[KR90b] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars II. *Information Sciences*, 52:221–246, 1990.

[KS93a] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, January 10–13, 1993.

[KS93b] Patrick Knupp and Stanly Steinberg. *Fundamentals of Grid Generation.* CRC Press, 1993.

[Lau88a] Clemens Lautemann. Decomposition trees: Structured graph representation and efficient algorithms. In M. Dauchet and N. Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, number 299 in Lecture Notes in Computer Science, pages 28–39. Springer-Verlag, March 1988.

[Lau88b] Clemens Lautemann. Efficient algorithms on context-free graph languages. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, pages 362–378, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[Lau90a] Clemens Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.

[Lau90b] Clemens Lautemann. Tree automata, tree decomposition and hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 520–537, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[LE91] M. Löwe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In R. H. Mohring, editor, *Proceedings of the 16th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '90)*, number 484 in Lecture Notes in Computer Science, pages 338–353. Springer-Verlag, June 1991.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Lin92] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, February 1992.

[LMZ92] Igor Litovsky, Yves Métivier, and Wieslaw Zielonka. The power and the limitations of local computations on graphs. In E. W. Mayr, editor, *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '92)*, number 657 in Lecture Notes in

Computer Science, pages 333–345, Wiesbaden-Naurod, Germany, June 18–20, 1992. Springer-Verlag.

[LS88] Timo Lepistö and Arto Salomaa, editors. *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, Tampere, Finland, July 11–15, 1988. Springer-Verlag.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[Luk74] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, May 1974.

[Mon70] Ugo G. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control*, 16:243–267, 1970.

[MP95] Serge Miguet and Jean-Marc Pierson. Load balancing strategies for a parallel system of particles. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 255–260, Lyon, France, September 4–6 1995. Springer-Verlag.

[MS92] Elefterios A. Melissaratos and Diane L. Souvaine. Coping with inconsistencies: A new approach to produce quality triangulations of polygonal domains with holes. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, pages 202–211, June 1992.

[MT75] Jayadev Misra and R. Endre Tarjan. Optimal chain partitioning of trees. *Information Processing Letters*, 4(1):24–26, September 1975.

[Nag77] Manfred Nagl. On the relation between graph grammars and graph l-systems. In Marek Karpinski, editor, *Fundamentals of Computation Theory: Proceedings of the 1977 International FCT-Conference*, number 56 in Lecture Notes in Computer Science, pages 142–151, Poznan-Kornik, Poland, September 1977. Springer-Verlag.

[NEGS82] M. Nagl, G. Engels, R. Gall, and W. Schäfer. Software specification by graph grammars. In Hartmut Ehrig, Manfred Magl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph Grammars and Their Application to Computer Science*, number 153 in Lecture Notes in Computer Science, pages 267–287, Haus Ohrbeck, West Germany, October 4–8, 1982. Springer-Verlag.

[NJS$^+$94] Tsuneo Nakanishi, Kazuki Joe, Hideki Saito, Constantine D. Polychronopoulos, Akira Fukuda, and Keijiro Araki. The data partitionining graph: Extending data and control dependencies for data partitioning. In K. Pingali, U. Banerjee, D. Gelertner, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, number 892 in Lecture Notes in Computer Science, pages 170–185, Ithaca, New York, USA, August 8–10 1994. Springer-Verlag.

[No.90] ESPRIT Basic Research Working Group No.3299. Computing by graph transformation: Overal aims and new results. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 688–703, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[NS90] M. Nagl and Andy Schürr. A specification environment for graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 599–609, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[OH91] Yasuyoshi Okada and Masahiro Hayashi. Graph rewriting systems and their application to network reliability analysis. In G. Schmidt and R. Berghammer, editors, *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '91)*, number 570 in Lecture Notes in Computer Science, pages 36–47, Fischbachau, Germany, June 17–19, 1991. Springer-Verlag.

[PH92] Przemyslaw Prusinkiewicz and James Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, pages 193–211. Springer-Verlag, Berlin, 1992.

[PR91] M. Pourazady and M. Radhakrishnan. Optimization of a triangular mesh. *Comput. & Structures*, 40(3):795–804, 1991.

[Rao84] Jean-Claude Raoult. On graph rewritings. *Theoretical Computer Science*, 32:1–24, 1984.

[Rei91] Rüdiger Reischuk. Graph theoretical methods for the design of parallel algorithms. In L. Budach, editor, *Proceedings of the 8th International Conference on Fundamentals of Computation Theory (FCT '91)*, number 529 in Lecture Notes in Computer Science, pages 61–67, Gosen, Germany, September 1991. Springer-Verlag.

[Rek94] 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`.

[RS86] N. Robertson and P. D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Journal of Algorithms*, 7:309–322, 1986.

[San95] Peter Sanders. Better algorithms for parallel backtracking. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 333–347, Lyon, France, September 4–6 1995. Springer-Verlag.

[Sch89] Andy Schürr. Introduction to PROGRESS, an attribute graph grammar based specification language. In M. Nagl, editor, *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '89)*, number 411 in Lecture Notes in Computer

Science, pages 151–165, Castle Rolduc, The Netherlands, June 1989. Springer-Verlag.

[Sch90a] Andy Schürr. Presentation of the IPSEN-environment: An integrated and incremental project support environment. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 66–66, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[Sch90b] Andy Schürr. Presentation of the PROGRESS-editor: A text-oriented hybrid editor for PROgrammed Graph REwriting SyStems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 67–67, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[Sch90c] Andy Schürr. PROGRESS: A VHL-language based on graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 4th International Workshop on Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 641–659, Bremen, Germany, March 5–9, 1990. Springer-Verlag.

[SE93] Hans Jürgen Schneider and Hartmut Ehrig, editors. *Graph Transformations in Computer Science: Proceedings of the International Workshop*, number 776 in Lecture Notes in Computer Science, Dagstuhl Castle, Germany, January 1993. Springer-Verlag.

[Sli82] A. O. Slisenko. Context-free grammars as a tool for describing polynomial-time subclasses of hard problems. *Information Processing Letters*, 14(2):52–56, April 1982.

[SV93] Ondrej Sýkora and Imrich Vrťo. Edge separators for graphs of bounded genus with applications. *Theoretical Computer Science*, 112(2):419–429, 1993.

[SV95] Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, February 1995.

[SW90] John E. Savage and Markus G. Wloka. On parallelizing graph-partitioning heuristics. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium On Automata, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 476–489, Warwick University, England, July 16–20, 1990. Springer-Verlag.

[Wat81] D.F. Watson. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Comput. J.*, 24(2):167–172, 1981.

[WCE$^+$95] C. Walshaw, M. Cross, M.G. Everett, S. Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: Proceedings of the Second International Workshop, IRREGULAR '95*, number 980 in Lecture Notes in Computer Science, pages 121–126, Lyon, France, September 4–6 1995. Springer-Verlag.

[WCJE94] C. Walshaw, M. Cross, S. Johnson, and M. Everett. A parallelisable alorithm for partitioning unstructured meshes. In *Proceedings of Irregular '94: Parallel Algorithms for Irregularly Structured Problems*, 1994.

[Wea90] N. P. Weatherill. Numerical grid generation. Lecture Series 1990-06, von Karman Institute for Fluid Dynamics, 1990.

[YS83] M.A. Yerry and M.S. Shephard. A modified quadtree approarch to finite element mesh generation. *IEEE Computing Applications*, 3:39–46, 1983.