Abstract of "Techniques and Tools for Graph Drawing" by Stina Bridgeman, Ph.D., Brown University, May 2002.

The visualization of information structured as a graph or network has applications in a variety of areas including programming and software development, database design, VLSI design, web navigation, network administration, modelling protein interactions and molecular structures, modelling social interactions between people and organizations, and illustrating train timetables. The problem of producing clear and readable drawings of graphs has received a great deal of attention from researchers.

We present work on several problems related to graph drawing, ranging from techniques for drawing algorithms to tools for making existing drawing algorithms easier to use.

In addressing compaction of orthogonal representations, we present a characterization of a class of orthogonal representations for which orthogonal drawings with minimum area or minimum area and total edge length can be produced efficiently. We also present heuristics for the general case which result in improved performance over previous compaction heuristics. These techniques can be used to improve existing orthogonal drawing algorithms based on the topology-shape-metrics approach.

The problem of measuring the similarity of two drawings of the same or nearly the same graph has received little attention. The need for such a measure arises in applications in which the user is working with a graph which changes over time and must be periodically redrawn. Ideally, if the graph structure only changes in a small way, the new drawing will only be a little different from the previous one. Designing effective drawing algorithms for this scenario requires an understanding of what "a little different" means. We begin the task of finding a suitable similarity measure by defining and evaluating several potential measures.

Finally, we present tools to facilitate the use of graph drawing technology by experts and novice users alike, in order to bring graph drawing to a wider audience. The Graph Drawing Server makes graph drawing technology easily available over the Internet, while GeomNet extends the idea to computational geometry algorithms. Both the Graph Drawing Server and GeomNet provide interfaces which can be used interactively or called as a subroutine by a program. PILOT is a learning tool based on these systems.

Abstract of "Techniques and Tools for Graph Drawing" by Stina Bridgeman, Ph.D., Brown University, May 2002.

The visualization of information structured as a graph or network has applications in a variety of areas including programming and software development, database design, VLSI design, web navigation, network administration, modelling protein interactions and molecular structures, modelling social interactions between people and organizations, and illustrating train timetables. The problem of producing clear and readable drawings of graphs has received a great deal of attention from researchers.

We present work on several problems related to graph drawing, ranging from techniques for drawing algorithms to tools for making existing drawing algorithms easier to use.

In addressing compaction of orthogonal representations, we present a characterization of a class of orthogonal representations for which orthogonal drawings with minimum area or minimum area and total edge length can be produced efficiently. We also present heuristics for the general case which result in improved performance over previous compaction heuristics. These techniques can be used to improve existing orthogonal drawing algorithms based on the topology-shape-metrics approach.

The problem of measuring the similarity of two drawings of the same or nearly the same graph has received little attention. The need for such a measure arises in applications in which the user is working with a graph which changes over time and must be periodically redrawn. Ideally, if the graph structure only changes in a small way, the new drawing will only be a little different from the previous one. Designing effective drawing algorithms for this scenario requires an understanding of what "a little different" means. We begin the task of finding a suitable similarity measure by defining and evaluating several potential measures.

Finally, we present tools to facilitate the use of graph drawing technology by experts and novice users alike, in order to bring graph drawing to a wider audience. The Graph Drawing Server makes graph drawing technology easily available over the Internet, while GeomNet extends the idea to computational geometry algorithms. Both the Graph Drawing Server and GeomNet provide interfaces which can be used interactively or called as a subroutine by a program. PILOT is a learning tool based on these systems.

Techniques and Tools for Graph Drawing

by
Stina Bridgeman
B.A., Williams College, 1995
Sc.M., Brown University, 1999

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2002

This dissertation by Stina Bridgeman is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Roberto Tamassia, Director

Recommended to the Graduate Council

Date _____ _____

Franco Preparata, Reader

Date _____ _____

Michael Goodrich, Reader
(University of California, Irvine)

Approved by the Graduate Council

Date _____ _____

Peder J. Estrup
Dean of the Graduate School and Research

Vita

Personal

Born in Roanoke, VA on October 18, 1974

Education

1999 Sc.M. in Computer Science, Brown University, Providence, RI

1995 B.A. with Highest Honors in Computer Science, Williams College, Williamstown, MA
Thesis Topic: "Finding Hamiltonian Cycles in Grid Graphs Without Holes"

Teaching and Professional Experience

Fall 2001- Instructor in Computer Science
Colgate University, Hamilton, NY

Spring 2001 Visiting Lecturer in Computer Science, CX 214 (Data Structures)
Middlebury College, Middlebury, VT

Fall 2000 Digital Image Design, Inc., New York, NY
Consulted on graph drawing for a visualization project.

Spring 1999 Graduate Teaching Assistant, CS 16 (Algorithms and Data Structures)
Brown University, Providence, RI
Wrote homeworks, exams, and their solutions; conducted homework help sessions and held office hours; delivered several lectures.

Summer 1996 Instructor, CPS1 (Theoretical Foundations of Computer Science)
Center for Talented Youth, Lancaster, PA
(One three-week session.) Lectured 3–4 hours per day; worked with students one-on-one; evaluated student work (written problems and programming tasks).

Summer 1995 Teaching Assistant, CPS1 (Theoretical Foundations of Computer Science)
Center for Talented Youth, Lancaster, PA
(Two three-week sessions.) Worked with students one-on-one; evaluated student work.

Spring 1995 Teaching Assistant, CS 108 (Artificial Intelligence: Image and Reality)
Williams College, Williamstown, MA
Graded student programs; worked with students one-on-one in labs.

Fall 1994 Teaching Assistant, CS 134 (Introduction to Computer Science)
Williams College, Williamstown, MA
Graded student programs; worked with students one-on-one in labs.

Awards and Honors

2000–01 Brown Dissertation Fellowship

1995–98 NSF Graduate Fellowship

Invited Talks

October 2000 Middlebury College, Middlebury, VT

April 1999 Williams College, Williamstown, MA

Conference Presentations

September 2001 Graph Drawing '01, Vienna, Austria (poster)

September 2000 Graph Drawing '00, Williamsburg, VA

March 2000 ACM Technical Symposium on Computer Science Education (SIGCSE '00), Austin, TX

October 1998 CGC Workshop on Geometric Computing, Providence, RI

August 1998 Graph Drawing '98, Montreal, Quebec, Canada

June 1997 13th Annual ACM Symposium on Computational Geometry, Nice, France (poster)

September 1996 Graph Drawing '96, Berkeley, CA

Publications

Journals

1. S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. *Journal of Graph Algorithms and Applications*. Invited submission.

2. S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *Journal of Graph Algorithms and Applications*. 4(3):47–74, 2000.

3. S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Computational Geometry: Theory and Applications*, 16(1):53–93, 2000.

4. G. Barequet, S. Bridgeman, C. A. Duncan, M. T. Goodrich, and R. Tamassia. Geometric computing over the Internet. *IEEE Internet Computing*, 3(2):21–29, 1999.

5. S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. *International Journal of Computational Geometry and Applications*, 9(4/5):419–446, 1999.

Conferences

6. S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. In Joe Marks, editor, *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2001.

7. S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 139–143. 2000.

8. S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. SAIL: A system for generating, archiving, and retrieving specialized assignments using LaTeX. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 300–304. 2000.

9. S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-Regularity and Planar Orthogonal Drawings. In *Graph Drawing (Proceedings of GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 8–26. Springer-Verlag, 1999.

10. S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal drawings of orthogonal representations. In *Abstracts of the 15th European Workshop on Computational Geometry*, pages 161–164. INRIA Sophia-Antipolis, 1999.

11. S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In *Graph Drawing (Proceedings of GD '98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 1998.

12. S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Optimal compaction of orthogonal representations. *CGC Workshop on Geometric Computing*. 1999.

13. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. InteractiveGiotto: An algorithm for interactive orthogonal graph drawing. In *Graph Drawing (Proceedings of GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 303–308. Springer-Verlag, 1997.

14. G. Barequet, S. Bridgeman, C. A. Duncan, M. T. Goodrich, and R. Tamassia. Classical computational geometry in GeomNet. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 412–414, 1997.

15. S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In *Graph Drawing (Proceedings of GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 45–52. Springer-Verlag, 1997.

Acknowledgements

First and foremost, I would like to thank my advisor, Roberto Tamassia, for his advice and support. I would also like to thank the other members of my committee, Franco P. Preparata and especially Michael T. Goodrich, with whom I have worked on several projects.

In addition, I wish to thank all those with whom I have worked, whether it be on a paper or just a conversation around the office: Ryan Baker, Gill Barequet, Ulrik Brandes, Giuseppe Di Battista, Walter Didimo, Christian Duncan, Jody Fanto, Ashim Garg, Stephen Kobourov, Giuseppe Liotta, Galina Shubina, and Luca Vismara.

Finally, I would like to thank my parents for providing support through all the years and my partner Elizabeth for typing in the text of chapter 6 when I could not find an electronic copy in the proper format, for cooking dinner for the last eight months, and most of all, for being there.

Contents

⋆ Chapters 2-7 have been previously published. See each chapter for the relevant citations.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Graphs and Graph Drawing

A graph in the mathematical sense consists of a set of vertices, commonly called V, and a set of edges E connecting pairs of vertices. Graphs are suitable for modeling many types of information; a few possibilities include computer network topology, interconnections between software components in a large program, molecular structures, airline flight databases, handwriting samples, and interactions between people and organizations in a community.

The goal of a graph drawing algorithm is to assign geometry to the graph so as to produce a drawing or layout (see Figure 1.1). As might be expected from the range of areas in which graph- or network-structured information arises, graph drawing has been applied to many domains, including software engineering [50, 76, 77, 100, 101, 129, 133]), program debugging [98, 121] and evaluation [141], visual programming languages [139], database design [49, 68, 87], and VLSI design. Other visualization applications include WWW navigation [89, 118], search engines (a previous version of the AltaVista search engine used a graph to illustrate the results of a query), and network administration [45]. In other fields, graph drawing techniques have been applied to molecular structures [23], protein interactions [12], social networks [29], telecommunications networks [64], and train timetables [30, 32].

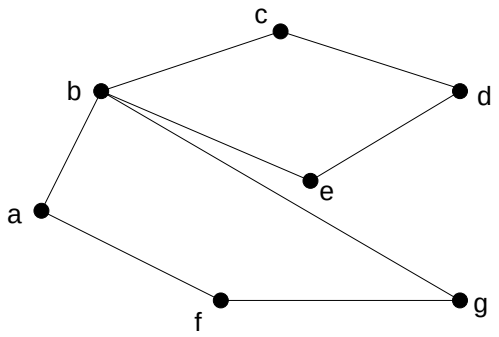
The problem of producing nice drawings of graphs has therefore received a lot of attention from researchers. Di Battista, Eades, Tamassia, and Tollis have produced a bibliography of graph drawing algorithms [53], and a more recent book [54] provides a detailed discussion of many graph drawing techniques. It is not sufficient for a graph drawing algorithm to simply produce any drawing of a graph; Figure 1.2 shows three drawings of the same graph, but the leftmost drawing would not be considered "nice" for most applications. What constitutes "nice" depends on the application — for applications such as VLSI design, a nice layout is easy and cost-effective to print on a chip. For applications in which the drawing is intended for a human audience, the drawing should be easy to read and should convey the important structural information contained in the graph.

V = { a, b, c, d, e, f, g }
E = { (a,b), (a,f), (b,c), (b,e), (b,g), (c,d), (d,e), (f,g) }

(a)

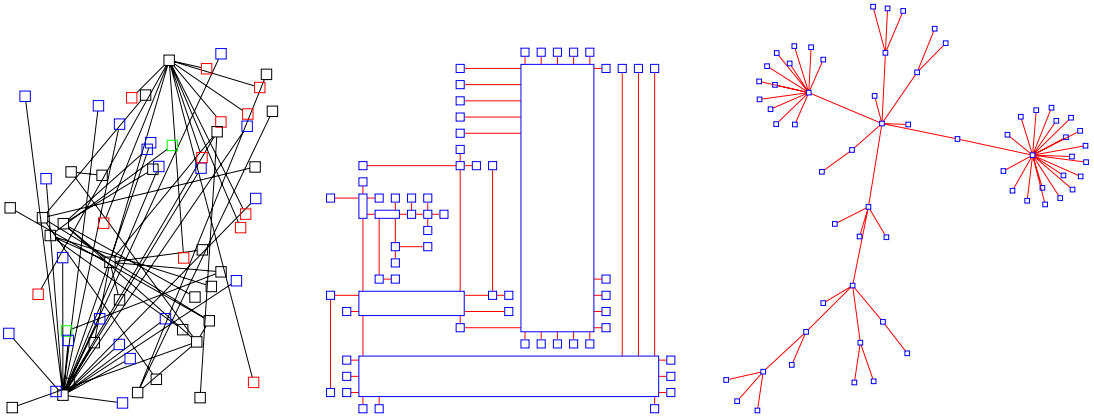
a:	(0,3)	e:	(9,4)
b:	(2,7)	f:	(6,0)
c:	(8,9)	g:	(14,0)
d:	(14,7)		

(b)



(c)

Figure 1.1: The role of a graph drawing algorithm. The algorithm takes (a) a graph description as input and (b) assigns geometry. (c) shows a graphical rendering of the geometry-enhanced graph description, where each vertex has the assigned coordinates. This is in some sense the simplest kind of layout, where only vertex positions are assigned. In general the drawing algorithm may place vertices, edge bends, vertex and edge labels, and other elements.



(a) (b) (c)

Figure 1.2: Three drawings of the same graph: (a) a jumble, (b) an orthogonal drawing, and (c) a force-directed drawing. All three are technically drawings; however (a) is not likely to be considered nice for any application.

For a graph drawing algorithm, the idea of "nice" is traditionally expressed by a combination of drawing conventions and aesthetic criteria. Drawing conventions are general constraints on the geometric representation of the drawing, such as requiring that the graph be laid out on a grid, that all directed edges point upward, or that no edge crossings be present. The drawing convention defines a set of rules which must be satisfied in the resulting drawing in order for the drawing to

be considered valid. Aesthetic criteria are optimization goals intended to improve the quality of the drawing, since a drawing convention alone is not sufficient to guarantee a nice result. (See, for example, Figure 1.2(a) — the drawing obeys the straight-line drawing convention, but it is nearly impossible to discern the graph's structure.) Common aesthetic criteria include minimizing the area, the number of edge bends, and the number of edge crossings, and maximizing the angular resolution and the display of symmetries. Tradeoffs exist, since satisfying one optimization goal typically worsens another. Furthermore, many of the optimization problems are NP-hard.

1.2 Organization

The following chapters will present work on several topics in the field of graph drawing, ranging from techniques used in graph drawing algorithms (compaction of orthogonal representations and similarity of graph drawings) to tools for making graph drawing algorithms easier to use (the Graph Drawing Server and GeomNet). The last chapter describes an application of the Graph Drawing Server and GeomNet. Each chapter is self-contained.

Compaction of Orthogonal Representations

A number of graph drawing conventions have emerged for different applications. *Orthogonal drawings* are drawings in which vertices are placed on a grid and edges are composed of horizontal and vertical segments routed along gridlines. Such drawings have applications in VLSI design and architectural floorplan layout, and are commonly used for entity-relationship diagrams, data-flow diagrams, and industrial schematics.

An *orthogonal representation* describes a class of orthogonal drawings with the same shape. Formally, an orthogonal representation consists of a description of the bends along each edge and the angles between consecutive edges around each vertex, but no geometric information such as edge lengths or vertex coordinates. The process of assigning edge lengths and vertex coordinates to obtain an orthogonal drawing is known as *compaction*, since the usual goal is the minimization of area and/or total edge length.

Compaction is a key step in orthogonal drawing algorithms using the *topology-shape-metrics* approach. Topology-shape-metrics algorithms typically have three steps: *planarization*, which fixes the embedding of the graph and replaces edge crossings with dummy vertices (thus determining the topology of the drawing); *orthogonalization*, which computes an orthogonal representation (determining the shape of the drawing); and *compaction* (determining the metrics of the drawing). See Tamassia [144, 145, 146], Fößmeier and Kaufman [72, 74], and Tamassia and Tollis [148] for several topology-shape-metrics algorithms for orthogonal drawings, and Klau and Mutzel [104] for an algorithm for quasi-orthogonal drawing (where edges are allowed to deviate from the grid near their endpoints).

Compaction can also be applied in a post-processing step added to any orthogonal drawing algorithm or drawing in order to remove extra space. In this case the orthogonal representation is

derived from the existing drawing. See Biedl and Kant [17] and Biedl, Madden, and Tollis [21] for two examples.

Chapter 2 presents a characterization of a class of orthogonal representations which can be optimally compacted with respect to area or total edge length in polynomial time. Heuristics for compacting general orthogonal representations are also presented, along with experimental results showing that these heuristics result in an improvement over previous compaction techniques. This work has been presented at the CGC Workshop on Geometric Computing (1998) [40], the 15th European Workshop on Computational Geometry (1999) [39], and Graph Drawing '99 [41]. The text of chapter 2 has been published in *Computational Geometry: Theory and Applications* [33].

Similarity of Graph Drawings

Graph drawing algorithms have typically been developed using a batch model, where the graph is redrawn from scratch every time a drawing is needed. Such algorithms are not well suited for interactive applications, where the user may repeatedly make small changes to the graph, requesting a new drawing after each change. These changes may be made because the user is explicitly editing the graph or because the graph represents some structure that is being updated, such as a map showing the user's navigation through a collection of Web pages or a data structure being manipulated by a running program. Other applications include the visualization of very large graphs, where a part of the graph condensed into a single node is expanded for further examination. Since the user is likely to be familiar with the existing drawing at each step and because the changes are small with respect to the whole graph, it is desirable to have this reflected in the new drawing — namely, the new drawing should look as much like the old drawing as possible, with the greatest variation in the immediate vicinity of the modified sections of the graph. Redrawing the graph from scratch often causes large changes in the drawing, destroying the user's "mental map" and forcing her to spend considerable time refamiliarizing herself with the layout.

Work on interactive graph drawing algorithms has been motivated by the need to preserve the user's mental map, but much of the evaluation of the algorithms has so far focused on traditional optimization criteria such as the area and the number of bends and crossings (see, for example, the analysis in Biedl and Kaufmann [18], Fößmeier [73], Papakostas, Six, and Tollis [126], and Papakostas and Tollis [125]). The interactive drawing scenarios "No Change" and "Relative Coordinates" [125] provide a model for interactive drawing algorithms in which the previous drawing is changed little if at all, but these models may be overly restrictive — they certainly do not allow for models of drawing stability such as those used by Brandes and Wagner [31] or in InteractiveGiotto [43]. These scenarios also do not provide a good basis for comparing different algorithms on the basis of mental map preservation, because they give only a guarantee that the drawings do not change "too much" and do not provide for a measurement of the degree of change.

Little work has been done on formally defining and evaluating potential similarity measures for use in interactive applications. Chapters 3 and 4 address this issue — chapter 3 defines several possible measures and gives a preliminary analysis of their suitability, while chapter 4 presents refined

definitions for several measures and the results of a more extensive user study. This work has been presented at Graph Drawing '98 [37] and '00 [42]. The text of chapter 3 has been published in the *Journal of Graph Algorithms and Applications* [38], and the text of chapter 4 has been submitted to the same journal.

Graph Drawing Server and GeomNet

As has been mentioned, a large number of graph drawing algorithms exist for a variety of applications. However, there are significant obstacles to overcome for someone wishing to make use of an algorithm — she must first find an implementation, install it, and convert her data to the proper format for the program to read. The last step is necessary because there is no single universally accepted format for describing graphs and their drawings, and researchers typically define their own formats when implementing an algorithm. To further complicate matters, things can go wrong at each step, since it may be difficult to find source code, the program may not compile or run on the user's systems, and the data conversion may be non-trivial.

Similar problems exist for geometric problems — geometric data typically involves a non-trivial combination of numerical and combinatorial relationships, and much care must be taken to develop formats for representing geometric data and for performing robust and reliable computations using such representations. There is a rich, growing literature directed at developing techniques for dealing with such robustness issues [5, 44, 65, 69, 70, 84, 85, 86, 92, 113, 156] and fundamental geometric algorithms are being redesigned taking into account robustness and arithmetic precision issues [69, 147]. The net result is a large number of implementations of non-trivial algorithms, but which may be difficult for a user, particularly a casual user, to take full advantage of.

GeomNet [9] is a service available over the World Wide Web (`http://loki.cs.brown.edu:8081/geomNet/`) which seeks to solve these problems by providing a collection of algorithms in one location, with a common, easy-to-use interface and a set of translators to convert between data formats. This allows users to employ a large number of algorithms while knowing only a few formats. The Graph Drawing Server [35] is a component of GeomNet providing access to graph drawing algorithms and common graph formats. The Geometric Algorithm Server and Mocha [7] provide access to services related to geometric computing, including geometric algorithm execution and animation [6], consistency checking of topological and geometric structures, and experimental study and comparison of geometric algorithms.

Chapters 5 and 6 describe the Graph Drawing Server and GeomNet, respectively. The Graph Drawing Server has been presented at Graph Drawing '96 [34] and GeomNet has been presented at the 13th Annual ACM Symposium on Computational Geometry (1997) [10]. The text of chapter 5 has been published in the *International Journal of Computational Geometry and Applications* [35] and the text of chapter 6 has been published in *IEEE Internet Computing* [9].

PILOT

PILOT is a tool designed to enhance student learning by providing random instances of problems which the student can solve and have graded online. PILOT can be used in three modes: an interactive learning mode, where the system provides immediate feedback for the student's actions; a practice mode, where the student solves the problem independently and then asks the system to check and grade the work, with the option of having the system solve the problem if she gets stuck; and an exam mode, where the user is only allowed a single attempt at solving the problem and cannot ask the system for a solution. The PILOT system acts as a proof-of-concept of GeomNet and the Graph Drawing Server: PILOT is built using the GeomNet architecture and additionally makes use of algorithms provided by the Graph Drawing Server.

PILOT is discussed in chapter 7. The system was presented at SIGCSE 2000 [36].

Chapter 2

Turn-Regularity and Optimal Area Drawings of Orthogonal Representations

2.1 Introduction

Orthogonal drawings are drawings of graphs in which every edge is represented by a chain of horizontal and vertical segments. An orthogonal representation is an equivalence class of orthogonal drawings that have the same "shape" (see Fig. 2.1). This class is formally described by specifying the bends along each edge and the angles between consecutive edges around each vertex. In this paper we consider planar orthogonal representations, that is, equivalence classes of orthogonal drawings for which at least one of the drawings is planar. Given a planar orthogonal representation H, the problem of finding a planar orthogonal grid drawing of H with small area is usually referred to as the *compaction* of H.

Orthogonal representations and planar orthogonal drawings have been extensively investigated (see, e.g., [16, 61, 62, 72, 79, 80, 81, 93, 145, 148, 149, 152]) because of their direct applications to the development of practical graph drawing techniques for information visualization [54]. In particular, it has been experimentally shown that drawing algorithms for general graphs based on the compaction of orthogonal representations with minimum number of bends perform better in practice than other known orthogonal drawing algorithms [57, 138]. Orthogonal representations and related concepts, such as slicing floorplans, are also widely used in VLSI layout compaction algorithms (see, e.g., [96, 107, 122, 142, 155]).

Previously published as Stina Bridgeman, Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Roberto Tamassia, and Luca Vismara. Turn-regularity and optimal drawings of orthogonal representations. *Computational Geometry: Theory and Applications.* 16(1):53–93, 2000.

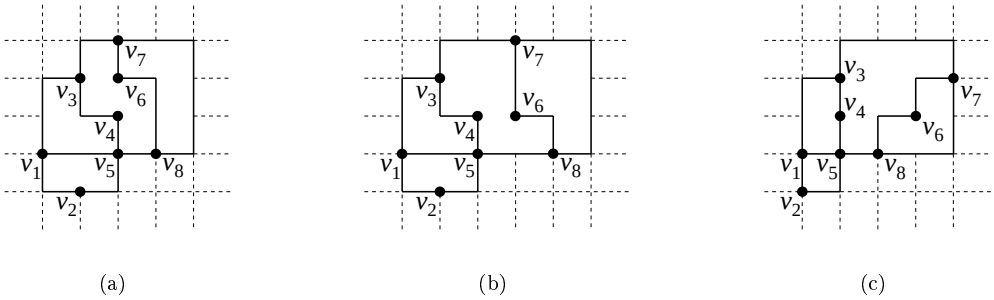


Figure 2.1: Three planar orthogonal drawings of a graph. Two drawings of the same orthogonal representation are shown in (a) and (b), while a drawing of a different orthogonal representation is shown in (c). The drawing in (a) has optimal area among all planar drawings of that orthogonal representation.

Despite the significant body of research on orthogonal representations, the development of effective compaction techniques remains a challenging task. It has been conjectured for a long time [152], and recently proved [128], that the optimal compaction of planar orthogonal representations, i.e., computing a minimum area planar orthogonal grid drawing of a given planar orthogonal representation, is NP-complete. The only class of planar orthogonal representations for which a polynomial-time optimal compaction algorithm is known is the trivial class of orthogonal representations whose faces are all rectangular [54].

From a practical perspective, the compaction algorithms used by current graph drawing libraries and systems, such as *AGD*[1], *GDToolkit*[2], and the *Graph Drawing Server*[3], are all variations of the compaction technique presented in [93, 145], which is based on the idea of splitting faces into rectangles. Since the splitting imposes unnecessary constraints on the geometry, the resulting drawings may have substantially suboptimal area.

The importance of compaction techniques for graph visualization applications is confirmed by a recent work of Klau and Mutzel [102, 103]. They consider the problem of assigning coordinates to vertices and edge bends of an orthogonal representation so that the total edge length is minimized. The problem is formulated as an integer linear program, whose practical performance is fairly good. Also, they show that the problem can be solved in polynomial time for those orthogonal representations in which there is only one possible relative position of any two vertices that results in a planar drawing; in this case, the inequalities of the corresponding ILP formulation form a totally unimodular matrix. The problem of minimizing the area of the drawing is not considered.

The main results of this paper can be summarized as follows.

- Given a planar orthogonal representation H, we define the concept of *turn-regularity* of a face of H, which is based on the structure of the sequence of left and right turns encountered when

[1] `http://www.mpi-sb.mpg.de/AGD/`

[2] `http://www.dia.uniroma3.it/~gdt/`

[3] `http://www.cs.brown.edu/cgc/graphserver/`

traversing the face. We show that the turn-regularity of a face can be tested in linear time.

- We relate turn-regularity to the concept of *switch-regularity* [59]. Namely, we characterize the turn-regularity of a face f in terms of the switch-regularity of two upward orientations of f.
- We introduce the concept of *orthogonal relation* between two vertices of H. This relation establishes the relative position of the two vertices in any planar orthogonal drawing of H. We show that an orthogonal relation is defined between every two vertices of H if and only if all the faces of H are turn-regular.
- We show that if H is turn-regular (i.e., all its faces are turn-regular), then any orthogonal drawing of H such that the orthogonal relations between every two vertices are satisfied is planar.
- We show that if H is turn-regular, then a planar orthogonal drawing of H with optimal area can be computed in $O(n)$ time and space, where n is the number of vertices and bends of H. Furthermore, a planar orthogonal drawing of H with optimal area and minimum total edge length within that area can be computed in $O(n^{7/4} \log n)$ time.
- We present the results of an experimental study on a test suite of planar orthogonal representations of randomly generated biconnected 4-planar graphs. The experiments show that the percentage of regular faces is quite high (95%). Motivated by this result, we have designed compaction heuristics based on the idea of "face regularization." Namely, we decompose non-regular faces into regular ones, and then perform an optimal compaction of the resulting planar orthogonal representation. We implemented our compaction algorithms and experimentally observed that the improvement in area is substantial when compared to the compaction algorithms available in state-of-the-art graph drawing libraries.

The paper is organized as follows. We recall some basic definitions, the notion of switch-regularity and its basic properties in Section 2.2. In Section 2.3, we define the orthogonal relations and the concept of turn-regularity, and relate the latter to switch-regularity. Two partial orientations of a turn-regular orthogonal representation and their properties are described in Section 2.4. In Section 2.5, we prove the existence of an orthogonal relation between every two vertices of a turn-regular orthogonal representation. The recognition algorithm and the two compaction algorithms are described in Section 2.6. In Section 2.7, we present the results of the experimental study. Section 2.8 contains the conclusions and some open problems. Finally, a table listing the various symbols used throughout the paper is given on page 50.

2.2 Preliminaries

2.2.1 Basic Definitions

We assume familiarity with graph terminology and basic properties of planar graphs (see, e.g., [24, 88]). The graphs we consider are assumed to be connected. For background on graph drawing, see [54].

A *drawing* of a graph G maps each vertex of G to a distinct point of the plane and each edge (u, v) of G to a simple Jordan curve with endpoints u and v. In an *orthogonal* drawing, each edge is represented as a polygonal chain of alternating horizontal and vertical segments. A drawing is *planar* if no two edges intersect, except, possibly, at common endpoints. A graph is planar if it has a planar drawing. A planar graph whose vertices have degree at most four is said to be 4-*planar*. Two planar drawings of a planar graph G are *equivalent* if, for each vertex v, they have the same clockwise circular sequence of edges incident on v. Hence, the planar drawings of G are partitioned into equivalence classes. Each of those classes is called an *embedding* of G. A planar drawing divides the plane into topologically connected regions, called *faces*. The *external* face is the unbounded region; all other faces are *internal*. Two equivalent planar drawings have the same faces. An *embedded planar* graph is a planar graph with a prescribed embedding.

An *st-digraph* is an acyclic digraph with a single source s (vertex with no incoming edges) and a single sink t (vertex with no outgoing edges). A *planar st-digraph* is an st-digraph that is planar and embedded with vertices s and t on the boundary of the external face. An important property of planar st-digraphs is that the incoming edges of each vertex v appear consecutively around v, as do the outgoing edges. Also, the boundary of each face f consists of two directed paths enclosing f, with common origin and destination.

Let G be an embedded 4-planar graph and let f be a face of G. In the following, we always traverse the boundary of f so that f is on the left, i.e., counterclockwise if f is internal and clockwise if f is external. The boundary of f consists of an alternating circular sequence of vertices and edges. Note that if G is not biconnected, there may be two occurrences of the same edge and multiple occurrences of the same vertex on the boundary of f. We denote by a_f the number of vertices (or edges) of f, each counted with its multiplicity.

Informally speaking, an *orthogonal representation* of an embedded 4-planar graph G describes an equivalence class of orthogonal drawings of G with "similar shape." It consists of a "decorated" version of G where each pair of consecutive edges around a vertex v is assigned an angle multiple of $\pi/2$ and each edge (u, v) is assigned a sequence of bends going from u to v, each a left or right turn. In this paper we consider *planar orthogonal representations*, that is, equivalence classes of orthogonal drawings for which at least one of the drawings is planar; in the rest of the paper, we omit the word planar when referring to orthogonal representations.

An orthogonal representation of G is formally defined as follows. Let v be a vertex of G. We assign an $\alpha \cdot \pi/2$ angle, $1 \leq \alpha \leq 4$, to each pair of consecutive edges around v (note that if v has degree one, the two consecutive edges around v coincide). We refer to these angles as *vertex-angles*.

Let e be an edge of G with end-vertices u and v. We assign to e two sequences of $\pi/2$ and $-\pi/2$ angles; one contains the angles on the left of the bends along e when going from u to v, and the other the angles on the left of the bends along e when going from v to u. A $\pi/2$ angle corresponds to a left turn, while a $-\pi/2$ angle corresponds to a right turn. We refer to these angles as *bend-angles*. Note that one of the two sequences associated with e can be obtained from the other by reversing the order and changing the signs of its elements. The following properties must be satisfied by the above assignments:

Property 1 *The sum of the vertex-angles around each vertex is* 2π.

Property 2 *The sum of the vertex-angles minus the sum of the bend-angles along the boundary of each face* f *is*

$$\begin{cases} (2a_f - 4) \cdot \pi/2 & \text{if } f \text{ is an internal face,} \\ (2a_f + 4) \cdot \pi/2 & \text{if } f \text{ is the external face.} \end{cases}$$

Since each bend can be replaced by a dummy vertex of degree 2, in the rest of the paper we assume, for the sake of simplicity, that orthogonal representations have no bends. We also assume that different drawings of the same orthogonal representation are iso-oriented, i.e., each edge has the same direction and its end-vertices are in the same relative position.

2.2.2 Switch-Regularity

We recall some terminology and results from [13, 15, 59]. A drawing of a digraph is said to be *upward* if edges are mapped to curves monotonically increasing in a common direction, for instance the vertical one. A digraph is *upward planar* if it admits an upward planar drawing. As we are going to show in the next section, upward planar drawings and orthogonal representations are strictly related. We recall here some notations and results that will be useful in the rest of the paper.

A vertex v of an embedded planar digraph G is said to be *bimodal* if all the incoming/outgoing edges of v appear consecutively around v in the embedding. If all the vertices of G are bimodal then G and its embedding are called *bimodal*. Let f be a face of a bimodal embedded digraph G. A vertex v of f with incident edges e_1 and e_2 is a *switch* if e_1 and e_2 are both incoming or both outgoing edges (note that e_1 and e_2 may coincide if the digraph is not biconnected). In the former case v is a *sink switch* of f, in the latter a *source switch* of f. We denote by $2n_f$ the number of switches of f.

Assign S and L labels to the switches of each face f such that (see Fig. 2.2(a)): (i) each source or sink of G has exactly one L label; (ii) for each face f, the number of L-labeled switches assigned to f is equal to $n_f - 1$ if f is an internal face, and to $n_f + 1$ if f is the external face. The S-labeled (L-labeled) source switches are called s_S-switches (s_L-switches) and the S-labeled (L-labeled) sink switches are called t_S-switches (t_L-switches). The circular sequence of labels of f so obtained is a *labeling* of f and is denoted by σ_f. Also, S_{σ_f} (L_{σ_f}) denotes the number of S-labels (L-labels) of σ_f. A face f of G labeled in this manner is *upward consistent*.

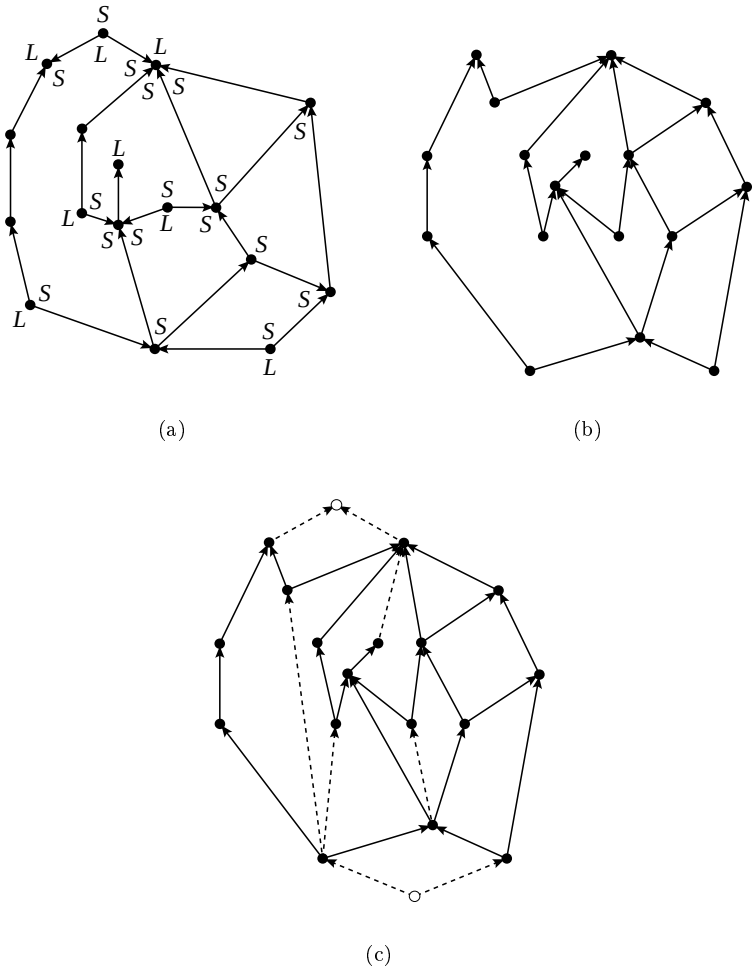


Figure 2.2: (a) A bimodal embedded planar digraph G with an upward consistent labeling of its faces. (b) An upward planar drawing of G corresponding to the upward consistent labeling in (a). (c) A complete saturator of G; s and t are represented as white circles, and saturating edges are represented as dashed segments.

Property 3 **[15]** *For each upward consistent face f,*

$$S_{\sigma_f} - L_{\sigma_f} = \begin{cases} 2 & \text{if } f \text{ is an internal face,} \\ -2 & \text{if } f \text{ is the external face.} \end{cases}$$

Theorem 1 **[15]** *A bimodal embedded digraph is upward planar if and only if all its faces have an upward consistent labeling.*

Let G be a bimodal embedded digraph such that all its faces have an upward consistent labeling. For each face of G, the S-label (L-label) assigned to a switch intuitively indicates that the angle formed by the two edges identifying the switch is smaller (larger) than π in an upward planar drawing

of G. Any such drawing is said to *correspond* to the upward consistent labeling of the faces of G. On the other hand, given an upward planar drawing of a bimodal embedded digraph G, an upward consistent labeling for each face of G can be obtained by simply checking whether the angle formed by each pair of edges identifying a switch is smaller or larger than π. Fig. 2.2(b) shows an upward planar drawing (corresponding to the upward consistent labeling) of the embedded planar digraph in Fig. 2.2(a).

Given an embedded upward planar digraph G, a *saturator* of G is a set of edges (each edge a *saturating edge*) plus two vertices s and t connected by edge (s,t). A saturating edge (u,v) is such that:

- Vertices u and v are switches of the same face, or u is a t_L-switch of the external face and $v = t$, or $u = s$ and v is an s_L-switch of the external face.
- If, $u, v \neq s, t$, either u is an s_S-switch and v is an s_L-switch or u is a t_L-switch and v is a t_S-switch. In the former case we say that u *saturates* v and in the latter case we say that v *saturates* u.
- The faces obtained with the insertion of a saturating edge are upward consistent.

A saturator of G is said to be *complete* if for every face f and for every switch u of f labeled L, u is an end-vertex of an edge of the saturator (see Fig. 2.2(c)). Clearly, adding to G a complete saturator yields a planar st-digraph.

Lemma 1 [**59**] *Every upward planar embedding admits a complete saturator.*

An upward planar embedding can have, in general, several complete saturators. The class of embedded upward planar digraphs for which there exists a unique complete saturator has been characterized in [59]. The characterization is based on a certain type of labeling. Namely, let G be an embedded upward planar digraph. An internal face f of G has a *switch-regular labeling* if σ_f does not contain two distinct maximal subsequences σ_1 and σ_2 of S-labels such that $S_{\sigma_1} > 1$ and $S_{\sigma_2} > 1$. An external face f of G has a switch-regular labeling if σ_f does not contain two consecutive S-labels. (Note that a switch-regular labeling is called just regular labeling in [59].) A face of G with a switch-regular labeling is a *switch-regular* face. For example all faces of Fig. 2.2(a) are switch-regular. An embedded upward planar digraph is *switch-regular* if all its faces have a switch-regular labeling. The corresponding embedding is also called *switch-regular*.

Theorem 2 [**59**] *An upward planar embedding has a unique complete saturator if and only if it is switch-regular.*

2.3 Turn-Regularity and Switch-Regularity

2.3.1 Orthogonal Relations

Let G be an embedded 4-planar graph, H be an orthogonal representation of G, Γ be a planar drawing of H, and v be a vertex of G. We denote by $x(v)$ and $y(v)$ the x- and y-coordinates of the point representing v in Γ. We define four binary relations on the vertex set of G: for each pair $\{u, v\}$ of vertices of G, these relations determine the relative position of u and v in *all planar drawings* of H.

- $u <_x v$ if $x(u) < x(v)$ for all planar drawings of H; in this case, we say that u is *left* of v and that v is *right* of u.
- $u =_x v$ if $x(u) = x(v)$ for all planar drawings of H; in this case, we say that u is *x-aligned* with v.
- $u <_y v$ if $y(u) < y(v)$ for all planar drawings of H; in this case, we say that u is *below* v and that v is *above* u.
- $u =_y v$ if $y(u) = y(v)$ for all planar drawings of H; in this case, we say that u is *y-aligned* with v.

We refer to the first two binary relations as x-relations and to the second two binary relations as y-relations. As an example, in the orthogonal representation in Fig. 2.1 $v_2 <_x v_8$, $v_6 =_x v_7$, $v_2 <_y v_3$, and $v_1 =_y v_5$.

We define three new binary relations on the vertex set of G, obtained by combining an x-relation and a y-relation: $=_x \wedge <_y$, $<_x \wedge =_y$, and $<_x \wedge <_y$. These three binary relations together with the binary relations $<_x$ and $<_y$ are collectively referred to as *orthogonal relations*. As an example, in the orthogonal representation in Fig. 2.1 $v_5 =_x v_4 \wedge v_5 <_y v_4$, $v_1 <_x v_8 \wedge v_1 =_y v_8$, and $v_1 <_x v_7 \wedge v_1 <_y v_7$, while no orthogonal relation holds for $\{v_4, v_6\}$.

2.3.2 Turn-Regularity

To characterize those orthogonal representations that have an orthogonal relation for each pair of vertices, we introduce the notion of *turn-regularity*.

Let G be an embedded 4-planar graph, H be an orthogonal representation of G, and f be a face of G. For each occurrence of vertex v on the boundary of f, let $prev(v)$ and $next(v)$ be the edges preceding and following v, respectively, on the boundary of f. Note that $prev(v)$ and $next(v)$ may coincide if the graph is not biconnected. We associate with each occurrence of v one or two *corners*. Namely:

- If the angle internal to f between $prev(v)$ and $next(v)$ is $\pi/2$ in H, we associate with v one *convex* corner, and say that v corresponds to a *left turn*.

- If the angle internal to f between $prev(v)$ and $next(v)$ is π in H, we associate with v one *flat* corner, and say that v corresponds to a *flat turn*.
- If the angle internal to f between $prev(v)$ and $next(v)$ is $3\pi/2$ in H, we associate with v one *reflex* corner, and say that v corresponds to a *right turn*.
- If the angle internal to f between $prev(v)$ and $next(v)$ is 2π in H, we associate with v an ordered pair of *reflex* corners, and say that v corresponds to a *U-turn*.

Hence, a circular sequence of corners can be associated with the boundary of f. For each corner c of f, let $turn(c) = 1$ if c is convex, $turn(c) = 0$ if c is flat, and $turn(c) = -1$ if c is reflex. As an example, in Fig. 2.3 a convex corner is associated with v_1, a flat corner with v_2, a reflex corner with v_3, and an ordered pair of reflex corners with v_4. The grey portion of Fig. 2.3 (and of other figures of the paper) represents the rest of the graph.

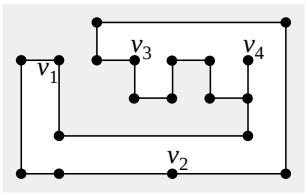


Figure 2.3: A face of an orthogonal representation. The grey portion represents the rest of the graph.

For each ordered pair $\{c_i, c_j\}$ of corners associated with vertices of f, let $rotation(c_i, c_j) = \sum_c turn(c)$ for all corners c along the boundary of f from c_i (included) to c_j (excluded). If c_i and c_j are associated with distinct vertices v_i and v_j, respectively, $rotation(c_i, c_j) \cdot \pi/2$ is the net angle turned between $prev(v_i)$ and $prev(v_j)$. As an example, in Fig. 2.3 let c_1, c_2, and c_3 be the corners associated with v_1, v_2, and v_3, respectively, and let $\{c'_4, c''_4\}$ be the ordered pair of corners associated with v_4; $rotation(c_1, c_2) = 3$, $rotation(c_3, c'_4) = 1$, $rotation(c_3, c''_4) = 0$, and $rotation(c_3, c_1) = -3$.

The following property is a direct consequence of the results in [145].

Property 4 *For each face f,*

$$rotation(c_i, c_i) = \begin{cases} 4 & \text{if } f \text{ is an internal face,} \\ -4 & \text{if } f \text{ is the external face.} \end{cases}$$

From the definition of $rotation(c_i, c_j)$ and from Property 4, Properties 5 and 6 easily follow.

Property 5 *For each ordered triplet of corners $\{c_i, c_j, c_k\}$ on the boundary of a face,*

$$rotation(c_i, c_j) = rotation(c_i, c_k) + rotation(c_k, c_j).$$

Property 6 *For each face f, $rotation(c_i, c_j) = 2$ if and only if*

$$rotation(c_j, c_i) = \begin{cases} 2 & \text{if } f \text{ is an internal face,} \\ -6 & \text{if } f \text{ is the external face.} \end{cases}$$

Two reflex corners c_i and c_j are called *kitty corners* if $rotation(c_i, c_j) = 2$ or $rotation(c_j, c_i) = 2$. In Fig. 2.1, the corners associated with vertices v_4 and v_6 are kitty corners. A face of an orthogonal representation is *turn-regular* if it has no kitty corners. An orthogonal representation is *turn-regular* if all its faces are turn-regular.

The reflex corners of a face of an orthogonal representation can be partitioned into four classes. Let f be a face of an orthogonal representation, and c be a reflex corner associated with vertex v of f.

- *NE-corners*: (*i*) if v is the left end-vertex of $prev(v)$ and the bottom end-vertex of $next(v)$; (*ii*) if v is the left end-vertex of both $prev(v)$ and $next(v)$, and c is the first corner associated with v; (*iii*) if v is the bottom end-vertex of both $prev(v)$ and $next(v)$, and c is the second corner associated with v.

- *NW-corners*: (*i*) if v is the bottom end-vertex of $prev(v)$ and the right end-vertex of $next(v)$; (*ii*) if v is the bottom end-vertex of both $prev(v)$ and $next(v)$, and c is the first corner associated with v; (*iii*) if v is the right end-vertex of both $prev(v)$ and $next(v)$, and c is the second corner associated with v.

- *SW-corners*: (*i*) if v is the right end-vertex of $prev(v)$ and the top end-vertex of $next(v)$; (*ii*) if v is the right end-vertex of both $prev(v)$ and $next(v)$, and c is the first corner associated with v; (*iii*) if v is the top end-vertex of both $prev(v)$ and $next(v)$, and c is the second corner associated with v.

- *SE-corners*: (*i*) if v is the top end-vertex of $prev(v)$ and the left end-vertex of $next(v)$; (*ii*) if v is the top end-vertex of both $prev(v)$ and $next(v)$, and c is the first corner associated with v; (*iii*) if v is the left end-vertex of both $prev(v)$ and $next(v)$, and c is the second corner associated with v.

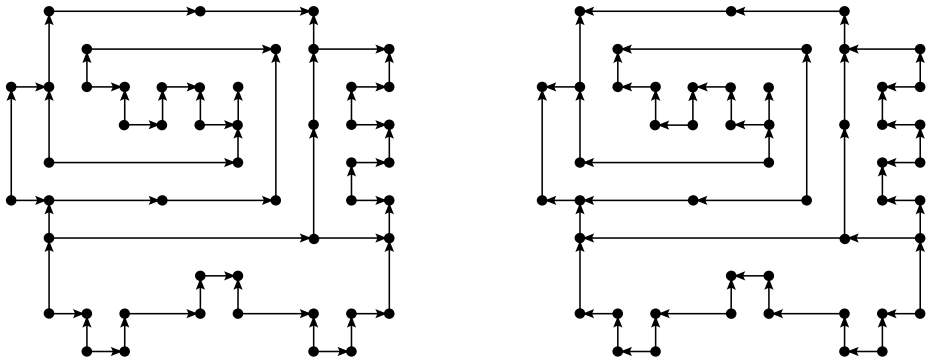
As an example, the reflex corner associated with vertex v_3 in Fig. 2.3 is a SW-corner, and the two corners associated with vertex v_4 are a SE-corner (the first one) and a SW-corner (the second one). The following property is a direct consequence of the definition of kitty corners.

Property 7 *Two reflex corners are kitty corners only if they form either a SW-NE pair or a SE-NW pair.*

2.3.3 Turn-Regularity and Switch-Regularity

Let G be an embedded 4-planar graph, H be an orthogonal representation of G, and Γ be a planar drawing of H. Let Γ_r be an orientation of Γ such that all vertical segments are directed upward

and all horizontal segments are directed rightward, and let Γ_ℓ be an orientation of Γ such that all vertical segments are directed upward and all horizontal segments are directed leftward. Observe that Γ_r is an upward planar drawing in the North-East direction and that Γ_ℓ is an upward planar drawing in the North-West direction. Γ_r and Γ_ℓ induce two orientations on H. We denote the oriented orthogonal representations by H_r and H_ℓ, respectively (see Fig. 2.4). In turn, H_r and H_ℓ induce two orientations on G. We denote the embedded 4-planar digraphs by G_r^H and G_ℓ^H, respectively. Observe that G_r^H and G_ℓ^H are embedded upward planar digraphs. Also, note that different orthogonal representations of G induce, in general, different orientations on G; since we work with a fixed orthogonal representation of a graph, we use G_r and G_ℓ in the rest of the paper, omitting the reference to H.

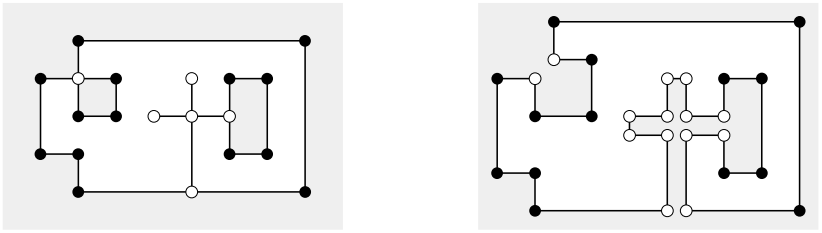


(a) (b)

Figure 2.4: Two orientations of the same orthogonal representation. (a) H_r. (b) H_ℓ.

To simplify the proofs of this section, we assume that the boundary of each face of H contains no vertices of degree one and no multiple occurrences of the same vertex. There is no loss of generality in this assumption, since a face f whose boundary contains vertices of degree one or multiple occurrences of the same vertex can be replaced, for the purpose of the proofs of this section, by an *expanded* face obtained from f as follows (see Fig. 2.5): (*i*) each degree one vertex v of f is replaced by a pair of vertices and by an edge connecting them, perpendicular to the edge incident with v in f; (*ii*) each vertex occurring k times on the boundary of f is replaced by k distinct vertices (note that $k \leq 4$ in an orthogonal representation); (*iii*) each edge occurring twice on the boundary of f is replaced by two edges having the same direction. Note that the proofs of this section consider a single face independently from the rest of H; thus, the expansion process is to be considered local to a single face of H. With the above assumption, each vertex v_i on the boundary of a face of H has exactly one associated corner, as described in Section 2.3.2. We denote with c_i the corner associated with v_i.

In order to show the connection between switch-regularity and turn-regularity, we first establish a connection between switches of G_r and G_ℓ and corners of H. We observe that since Γ_r (Γ_ℓ) is an



(a) (b)

Figure 2.5: (a) A face f of an orthogonal representation. The vertices affected by the expansion are represented as white circles. (b) The expanded face corresponding to f.

upward planar drawing of G_r (G_ℓ), it induces an upward consistent labeling on G_r (G_ℓ). Let f be a face of G_r or G_ℓ; two switches v_i and v_j of f are *consecutive* if no vertex on the portion of the boundary of f from v_i to v_j is a switch of f.

Property 8 *Let v_i and v_j be two consecutive switches of a face of G_r or G_ℓ, and c_i and c_j be the associated corners. One of the following holds:*

- *If v_i and v_j are both L-labeled switches (an LL-transition), then $rotation(c_i, c_j) = -2$.*
- *If v_i is an L-labeled switch and v_j is an S-labeled switch (an LS-transition), then $rotation(c_i, c_j) = -1$.*
- *If v_i is an S-labeled switch and v_j is an L-labeled switch (an SL-transition), then $rotation(c_i, c_j) = 1$.*
- *If v_i and v_j are both S-labeled switches (an SS-transition), then $rotation(c_i, c_j) = 2$.*

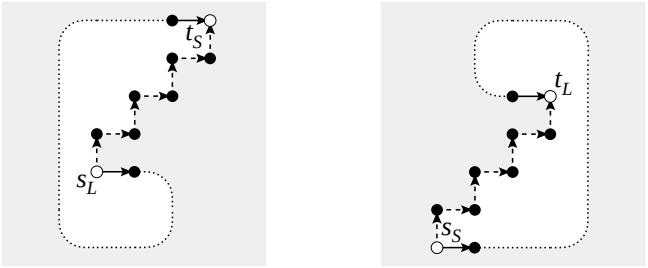
Proof: We only consider G_r; the proof for G_ℓ is similar. Since v_i and v_j are consecutive switches, they are either connected by a single edge or by a "staircase." Fig. 2.6 shows the possible arrangements of non-flat corners between c_i and c_j for a face of G_r. Note that flat corners do not affect the value of *rotation*, and can safely be ignored. □

Let σ be a sequence of L-labels and S-labels. In the rest of the section, we denote the number of LL-transitions in σ as n^σ_{LL}, the number of LS-transitions in σ as n^σ_{LS}, the number of SL-transitions in σ as n^σ_{SL}, and the number of SS-transitions in σ as n^σ_{SS}.

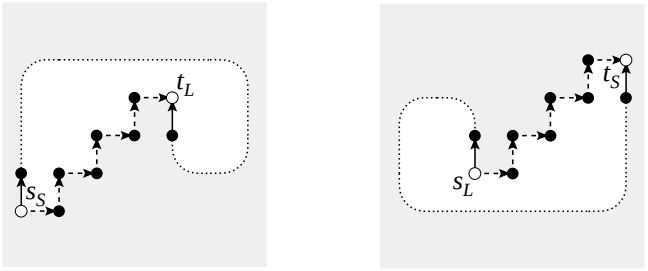
Property 9 *Let σ be a sequence of L-labels and S-labels, and let $\rho = l_1 \sigma l_n$. We have:*

- *if $l_1 = L$ and $l_n = L$, then $n^\rho_{LS} = n^\rho_{SL}$;*
- *if $l_1 = L$ and $l_n = S$, then $n^\rho_{LS} = n^\rho_{SL} + 1$;*
- *if $l_1 = S$ and $l_n = L$, then $n^\rho_{SL} = n^\rho_{LS} + 1$;*

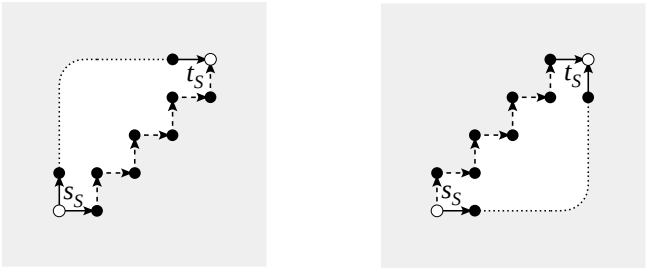
(a)



(b)



(c)



(d)

Figure 2.6: The shape of a face f of H_r between two consecutive switches. The "staircase" portion of the boundary of f (represented by dashed segments) may consist of a single edge. (a) LL-transitions. (b) LS-transitions. (c) SL-transitions. (d) SS-transitions.

- *if* $l_1 = S$ *and* $l_n = S$, *then* $n^{\rho}_{SL} = n^{\rho}_{LS}$.

Lemma 2 *Let* σ *be a sequence of* L*-labels and* S*-labels, and let* $\rho = L\sigma S$. *Then, we have:*

- $L_\rho = n^{\rho}_{LL} + n^{\rho}_{SL} + 1$;
- $S_\rho = n^{\rho}_{SS} + n^{\rho}_{SL} + 1$.

Proof: We first prove that $L_\rho = n^{\rho}_{LL} + n^{\rho}_{SL} + 1$. For each sequence of L-labels, the number of labels is equal to the number of LL-transitions plus one. So the number of L-labels in ρ is equal to the number of LL-transitions in ρ plus the number of sequences of L-labels in ρ. In turn, since each sequence of L-labels in ρ, except the first one, is preceded by an S-label, the number of sequences of L-labels is equal to the number of SL-transitions in ρ plus one. Hence, the claim.

The proof that $S_\rho = n^{\rho}_{SS} + n^{\rho}_{SL} + 1$ is similar, once we observe that each sequence of S-labels in ρ, except the last one, is followed by an L-label. □

We now describe how to simplify portions of the boundary of a face of H by removing switches (and hence corners), without affecting the value of *rotation* for the remaining corners. In the rest of the section, we denote the label of a switch v_i by $label(v_i)$ and add an index to *rotation* to denote with respect to which face it is computed. Let $\{v_1, v_2, v_3, v_4\}$ be four consecutive switches of a face f of H. If either $label(v_2) = S$ and $label(v_3) = L$, or $label(v_2) = L$ and $label(v_3) = S$, we *collapse* the pair of switches $\{v_2, v_3\}$ by replacing the portion of the boundary of f between $prev(v_2)$ (included) and $next(v_3)$ (included) with a single edge having the same direction as $prev(v_2)$ and $next(v_3)$.

Lemma 3 *Let* f *be a face of* H *and let* $\{v_1, v_2, v_3, v_4\}$ *be four consecutive switches of* f *in* G_r *or* G_ℓ *such that either* $label(v_2) = S$ *and* $label(v_3) = L$, *or* $label(v_2) = L$ *and* $label(v_3) = S$. *Let* f' *be the face obtained from* f *by collapsing the pair of switches* $\{v_2, v_3\}$. *Then,* $rotation_f(c_1, c_4) = rotation_{f'}(c_1, c_4)$.

Proof: We prove the case $label(v_2) = S$ and $label(v_3) = L$ by case analysis. Let σ be the sequence of labels of v_1, v_2, v_3, and v_4 in f and σ' be the sequence of labels of v_1 and v_4 in f'. From Property 8 we have:

σ	$rotation_f(c_1, c_4)$	σ'	$rotation_{f'}(c_1, c_4)$
$S\ SL\ S$	$2 + 1 - 1 = 2$	SS	2
$S\ SL\ L$	$2 + 1 - 2 = 1$	SL	1
$L\ SL\ S$	$-1 + 1 - 1 = -1$	LS	-1
$L\ SL\ L$	$-1 + 1 - 2 = -2$	LL	-2

The proof of the case $label(v_2) = L$ and $label(v_3) = S$ is analogous. □

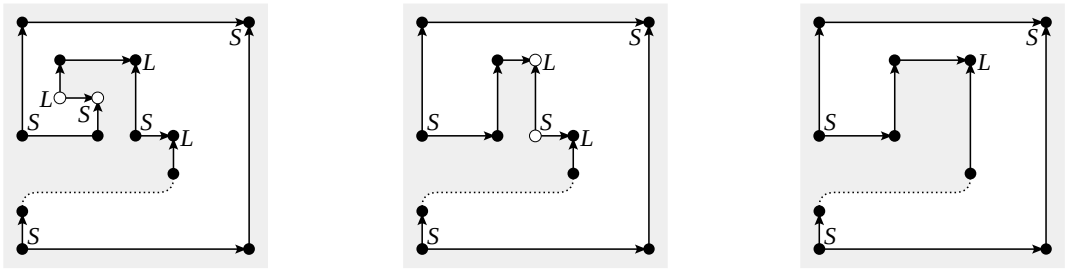
Lemma 4 *Let* f *be a face of* H, $V = \{v_1, W, v_k\}$ *be a sequence of* k *consecutive switches of* f *in* G_r *or* G_ℓ, *and* σ *be the sequence of labels corresponding to* W. *Let* f' *be the face obtained from* f *by applying the collapse operation to pairs of switches of* V *as many times as possible, and* $V' = \{v_1, W', v_k\}$ *be the resulting sequence of switches. We have that:*

- $rotation_f(c_1, c_k) = rotation_{f'}(c_1, c_k)$;
- *if* $S_\sigma > L_\sigma$, *then* W' *is a sequence of* $S_\sigma - L_\sigma$ S*-labeled switches;*
- *if* $S_\sigma < L_\sigma$, *then* W' *is a sequence of* $L_\sigma - S_\sigma$ L*-labeled switches;*
- *if* $S_\sigma = L_\sigma$, *then* W' *is empty.*

Proof: Clearly, $rotation_f(c_1, c_k) = rotation_{f'}(c_1, c_k)$ by Lemma 3. We now prove the case $S_\sigma > L_\sigma$; the other two cases are analogous.

If $L_\sigma = 0$, then σ consists entirely of S-labeled switches, and so the lemma is trivially true because no collapse operation can be applied, and hence $W' = W$. Otherwise, there is at least one subsequence $\{v_{i-1}, v_i, v_{i+1}, v_{i+2}\}$, $1 < i < k-1$ of V such that either $label(v_i) = S$ and $label(v_{i+1}) = L$, or $label(v_i) = L$ and $label(v_{i+1}) = S$. Hence, the collapse operation can be applied, and the process can be repeated as long as $L_\sigma > 0$. The first and last switches of V, v_1 and v_k, will never be eliminated, because they can only match v_{i-1} or v_{i+2} in the pattern of four consecutive switches. Their presence also guarantees that every SL or LS subsequence of σ can be matched to a pattern and thus eliminated. Since one L-labeled switch and one S-labeled switch are removed each time the collapse operation is applied, there will be $S_\sigma - L_\sigma$ S-labeled switches left when the process is complete. □

An example of the collapsing process is shown in Fig. 2.7. Let σ be the sequence of labels corresponding to a sequence W of consecutive switches of a face of G_r or G_ℓ, and let W' the sequence of switches obtained from W as described in Lemma 4. In the rest of the section, the sequence of labels σ' corresponding to W' is referred to as the *collapsed version* of σ.



(a) $S\ SS(SL)LS\ L$ (b) $S\ SS(LS)\ L$ (c) $S\ SS\ L$

Figure 2.7: Collapsing a sequence of switches. In each step, the switches being collapsed are represented as white circles, and the SL or LS pair of labels being removed is shown in parentheses.

Theorem 3 *An orthogonal representation* H *of an embedded* 4*-planar graph* G *is turn-regular if and only if the embedded upward planar digraphs* G_r *and* G_ℓ *are both switch-regular.*

Proof: In the proof we adopt the regular expression formalism (see, e.g., [94]); in particular, we use the concatenation of strings and the Kleene star operator.

Only if. We prove the claim for G_r; the proof for G_ℓ is similar. Suppose, for a contradiction, that H is turn-regular and G_r is not switch-regular; then, there is a face f of G_r that does not have a switch-regular labeling.

We first consider the case in which f is an internal face: σ_f must contain two distinct maximal subsequences of S-labels with length greater than one. Thus, the labeling of f can be expressed as $\sigma_f = SS\sigma_1 SSS^*L\sigma_2$, where $\sigma_1 = L((SL)^*L^*)^*$ is any sequence with an initial L-label and with no two consecutive S-labels, and σ_2 is any (possibly empty) sequence of S-labels and L-labels such that σ_f satisfies Property 3. We show that there is an L-labeled switch v_j in σ_1 and an L-labeled switch v_k in $L\sigma_2$ such that c_j and c_k are kitty corners. This implies that f and thus H are not turn regular; a contradiction.

To simplify matters, we consider $\sigma'_f = SS\sigma'_1 SSS^*L\sigma'_2$ instead of σ_f, where σ'_1 and σ'_2 are the collapsed versions of σ_1 and σ_2. Note that every label in σ'_f is also in σ_f, and that the corresponding switch is associated with a corner of f. In the regular expression describing σ_1, each S-label is followed by an L-label; thus, the initial L-label of σ_1 guarantees that σ_1 contains at least one more L-label than S-label. Hence, by Lemma 4, $\sigma'_1 = LL^*$. Let $l \geq 1$ be the length of σ'_1 and $s \geq 2$ be the length of the SSS^* subsequence between σ'_1 and $L\sigma'_2$. The following table shows the general structure of σ'_f and the values of *rotation*, which can be easily verified by Property 8.

			$\overbrace{\quad\quad\quad\quad}^{\sigma'_1}$				$\overbrace{\quad\quad\quad\quad\quad\quad\quad\quad}^{SSS^*}$					$\overbrace{\quad}^{\sigma'_2}$
v_i	v_1	v_2	v_3	v_4	$\cdots$	v_{2+l}	v_{3+l}	v_{4+l}	$\cdots$	v_{2+l+s}	v_{3+l+s}	$\cdots$
σ'_f	S	S	L	L	$\cdots$	L	S	S	$\cdots$	S	L	$\cdots$
$rotation(c_1, c_i)$	0	2	3	1	$\cdots$	$5-2l$	$4-2l$	$6-2l$	$\cdots$	$2(s+1)-2l$	$2s+3-2l$	$\cdots$

We consider two cases for the value of s, namely $2 \leq s \leq l+1$ and $s \geq l+2$, and prove the claim differently in the two cases.

If $2 \leq s \leq l+1$, then v_{4-s+l} is an L-labeled switch, since $3 \leq 4-s+l \leq 2+l$; v_{3+l+s} is also an L-labeled switch, and we have, by Property 5:

$$\begin{aligned} rotation(c_{4-s+l}, c_{3+l+s}) &= rotation(c_1, c_{3+l+s}) - rotation(c_1, c_{4-s+l}) \\ &= (2s+3-2l) - \{5 - 2[(4-s+l)-2]\} \\ &= 2 \end{aligned}$$

Hence, c_{4-s+l} and c_{3+l+s} are kitty corners.

If $s \geq l+2$, we consider the structure of the sequence $L\sigma_2 S$, where the final S is the label of v_1 (we recall that σ_f is a circular sequence). For ease of reference, we denote $L\sigma_2 S$ as ρ. By Properties 8 and 9, we have:

$$\begin{aligned} rotation(c_{3+l+s}, c_1) &= -2n^\rho_{LL} - n^\rho_{LS} + n^\rho_{SL} + 2n^\rho_{SS} \\ &= -2n^\rho_{LL} - (n^\rho_{SL}+1) + n^\rho_{SL} + 2n^\rho_{SS} \\ &= 2(n^\rho_{SS} - n^\rho_{LL}) - 1 \end{aligned}$$

On the other hand, by Properties 4 and 5, and by the above table, we have:

$$\begin{aligned} \mathit{rotation}(c_{3+l+s}, c_1) &= 4 - \mathit{rotation}(c_1, c_{3+l+s}) \\ &= 4 - (2s + 3 - 2l) \\ &\leq 4 - [2(l+2) + 3 - 2l] \\ &\leq -3 \end{aligned}$$

And thus, by combining the two results:

$$\begin{aligned} 2(n^\rho_{SS} - n^\rho_{LL}) - 1 &\leq -3 \\ n^\rho_{LL} - n^\rho_{SS} &\geq 1 \end{aligned}$$

Now, by Lemma 2, we have:

$$\begin{aligned} L_{\sigma_2} &= L_\rho - 1 &&= n^\rho_{LL} + n^\rho_{SL} \\ S_{\sigma_2} &= S_\rho - 1 &&= n^\rho_{SS} + n^\rho_{SL} \end{aligned}$$

And thus $L_{\sigma_2} > S_{\sigma_2}$, because

$$\begin{aligned} L_{\sigma_2} - S_{\sigma_2} &= n^\rho_{LL} - n^\rho_{SS} \\ &\geq 1 \end{aligned}$$

Let f' be the face obtained from f by repeatedly collapsing pairs of consecutive switches of $\{v_{3+l+s}, \ldots, v_1\}$. By Lemma 4 and the above discussion, σ'_2 has only L-labels; let v_k be the last (L-labeled) switch in σ'_2. Then, by Properties 4 and 5, and by Lemma 4, we have:

$$\begin{aligned} \mathit{rotation}(c_3, c_k) &= 4 - \mathit{rotation}_f(c_k, c_3) \\ &= 4 - \mathit{rotation}_{f'}(c_k, c_3) \end{aligned}$$

And since between v_k and v_3 in f' there are an LS-, an SS-, and an SL-transition, we have, by Property 8:

$$\begin{aligned} \mathit{rotation}(c_3, c_k) &= 4 - (-1 + 2 + 1) \\ &= 2 \end{aligned}$$

Hence, c_3 and c_k are kitty corners.

We now consider the case in which f is the external face: σ_f must contain two consecutive S-labels. Thus, the labeling of f can be expressed as $\sigma_f = SS\sigma_1$ where σ_1 is any (possibly empty) sequence of S-labels and L-labels such that σ_f satisfies Property 3. Again, to simplify matters, we consider $\sigma'_f = SS\sigma'_1$ instead of σ_f, where σ'_1 is the collapsed version of σ_1. Note that, by Property 3, $L_{\sigma_1} > S_{\sigma_1}$. Hence, by Lemma 4, σ'_1 has only L-labels; in particular, since also σ'_f must satisfy Property 3, $\sigma'_1 = LLLL$. The following table shows the structure of σ'_f and the values of *rotation*, which can be easily verified by Property 8.

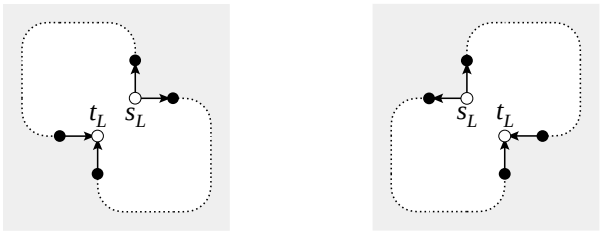
v_i	v_1	v_2	v_3	v_4	v_5	v_6
σ'_f	S	S	L	L	L	L
$\mathit{rotation}(c_1, c_i)$	0	2	3	1	-1	-3

Then, by Property 5, we have:

$$\begin{aligned} \mathit{rotation}(c_3, c_6) &= \mathit{rotation}(c_1, c_6) - \mathit{rotation}(c_1, c_3) \\ &= -6 \end{aligned}$$

Hence, by Property 6, c_3 and c_6 are kitty corners.

If. Suppose, for a contradiction, that G_r and G_ℓ are both switch-regular and H is not turn-regular; then, there is a face f of H with a pair $\{c_j, c_k\}$ of kitty corners. Note that c_j and c_k are associated with a pair of L-labeled switches in either G_r (see Fig. 2.8(a)) or G_ℓ (see Fig. 2.8(b)).



(a) (b)

Figure 2.8: The two possibilities for a pair of kitty corners: (a) a SW-NE pair of corners associated with two switches in G_r; (b) a SE-NW pair of corners associated with two switches in G_ℓ.

We assume that c_j and c_k are associated with the L-labeled switches v_j and v_k in G_r; the proof for G_ℓ is similar. The labeling of the face f of G_r can be expressed as $\sigma_f = L\sigma_1 L\sigma_2$, where σ_1 and σ_2 are any two sequences of S-labels and L-labels such that σ_f satisfies Property 3. The following table shows the general structure of σ_f.

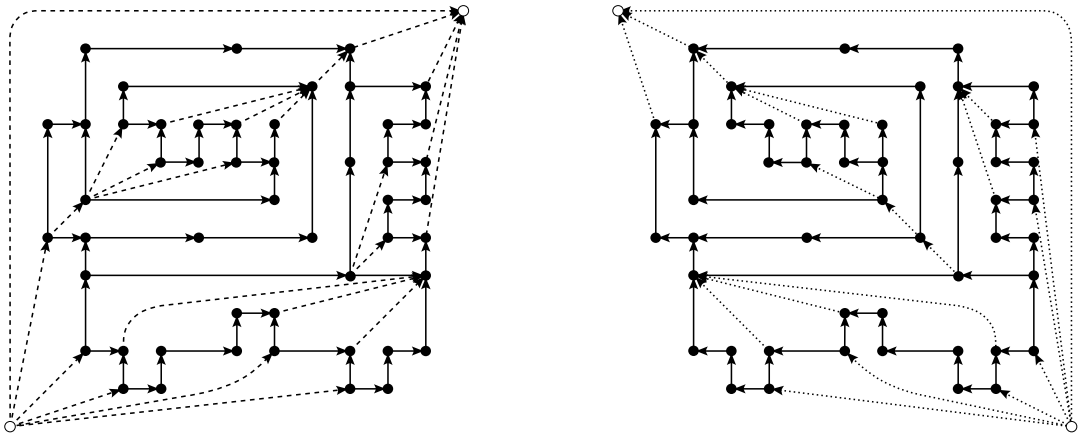
		σ_1				σ_2					
v_i	v_j	v_{j+1}	$\cdots$	v_{k-1}	v_k	v_{k+1}	$\cdots$	v_n	v_1	$\cdots$	$v_j - 1$
σ_f	L		$\cdots$		L			$\cdots$			
$rotation(c_j, c_i)$	0		$\cdots$		2			$\cdots$			

We show that σ_1 contains at least two consecutive S-labeled switches and that, if f is an internal face, σ_2 also contains at least two consecutive S-labeled switches. Thus, f does not have a switch-regular labeling and G_r is not switch-regular; a contradiction. For ease of reference, we denote $L\sigma_1 L$ as ρ. By Property 9, $n^\rho_{LS} = n^\rho_{SL}$, and thus, by Property 8, the total contribution of the LS- and SL-transitions to $rotation(c_j, c_k)$ is zero. Since $rotation(c_j, c_k) = 2$, still from Property 8, it follows that $n^\rho_{SS} > n^\rho_{LL} \geq 0$. Hence, σ_1 contains at least two consecutive S-labeled switches. If f is an internal face, the same argument can be used to prove that σ_2 contains at least two consecutive S-labeled switches, because, by Property 6, also $rotation(c_k, c_j) = 2$. □

2.4 Orientations and Paths

Let G be an embedded 4-planar graph and H be a turn-regular orthogonal representation of G. As seen in Section 2.2.2, a complete saturator of an embedded upward planar digraph consists of two vertices s and t and a set of (directed) saturating edges. Fig. 2.9(a) shows a complete saturator of graph G_r corresponding to the oriented orthogonal representation H_r shown in Fig. 2.4(a).

Fig. 2.9(b) shows a complete saturator of graph G_ℓ corresponding to the oriented orthogonal representation H_ℓ shown in Fig. 2.4(b). In the rest of the paper we never consider the saturating edges of G_r and G_ℓ incident with s or t, even when not explicitly stated. Let f be an internal face of H; a maximal vertical or horizontal chain of f is said to be *unconstrained* if both its end-vertices correspond to a right turn of f. Note that an unconstrained maximal chain of f may consist of a single, degree one vertex.



(a) (b)

Figure 2.9: (a) G_r (edges represented as solid segments) and a complete saturator (edges represented as dashed segments). (b) G_ℓ (edges represented as solid segments) and a complete saturator (edges represented as dotted segments).

We now construct two partially-directed graphs, one representing the "left" relation between maximal vertical chains of H, the other representing the "below" relation between maximal horizontal chains of H. The graph representing the "left" relation between maximal vertical chains of H is constructed as follows. We first augment H with the saturating edges of G_r and G_ℓ incident with an end-vertex of an unconstrained maximal vertical chain of H. We then orient the horizontal edges of H from left to right, reverse the orientation of the saturating edges of G_ℓ, and leave the vertical edges of H not oriented so that they can be traversed in both ways. We denote by H_x the resulting graph (see Fig. 2.10(a)). Similarly, the graph representing the "below" relation between maximal horizontal chains of H is constructed as follows. We first augment H with the saturating edges of G_r and G_ℓ incident with an end-vertex of an unconstrained maximal horizontal chain of H. We then orient the vertical edges of H from bottom to top and leave the horizontal edges of H not oriented so that they can be traversed in both ways. We denote by H_y the resulting graph (see Fig. 2.10(b)).

The following theorem shows how turn-regularity characterizes those orthogonal representations for which the "left" relation between maximal vertical chains and the "below" relation between maximal horizontal chains are uniquely determined.

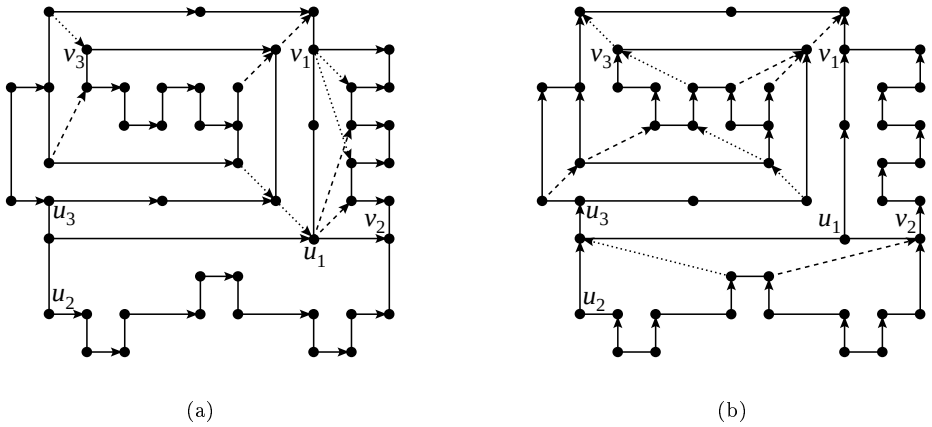


Figure 2.10: (a) H_x. (b) H_y. Both graphs are obtained using the complete saturators shown in Fig. 2.9.

Theorem 4 *Let H be an orthogonal representation. H_x and H_y are uniquely determined if and only if H is turn-regular.*

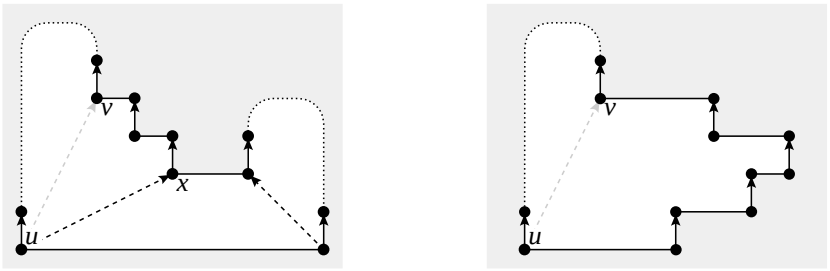
Proof: Easily follows from Theorem 3, the construction of H_x and H_y, and the definition of switch-regular embedded upward planar digraph. □

Note that H_x and H_y are no longer orthogonal representations, and may, in general, be non-planar. From the definition of saturator, it follows that each saturating edge from G_r and G_ℓ used in the construction of H_x and H_y has both end-vertices on the same face of H. Two saturating edges in H_x or H_y are said to *cross* each other if their end-vertices appear alternately on the boundary of a common face of H. In the rest of the paper we refer to the maximal chains of non-oriented edges of H_x as maximal vertical chains of H_x, and denote by $mvc(v)$ the maximal vertical chain of H_x containing vertex v. Analogously, we refer to the maximal chains of non-oriented edges of H_y as maximal horizontal chains of H_y, and denote by $mhc(v)$ the maximal horizontal chain of H_y containing vertex v.

In the rest of this section, we present a series of technical lemmas that will be used in Section 2.5. In the proofs, we can assume the absence of degree one vertices; otherwise, the expansion mechanism described at the beginning of Section 2.3.3 can be applied.

Lemma 5 *Let H be a turn-regular orthogonal representation. Let (u, v) be a saturating edge from G_r not used in the construction of H_x (H_y); then, there exists a path from u to v in H_x (H_y). Similarly, let (u, v) be a saturating edge from G_ℓ not used in the construction of H_x (H_y); then, there exists a path from v to u in H_x (from u to v in H_y).*

Proof: We prove the claim for a saturating edge (u, v) from G_r not used in the construction of H_y; the proof for the other three cases is similar.



(a) (b)

Figure 2.11: The two cases in the proof of Lemma 5.

Let f be the face of H containing u and v. We consider in detail the case in which u is an s_S-switch and v an s_L-switch of f in G_r; the case in which u is a t_L-switch and v a t_S-switch of f in G_r is similar. Let $f(u,v)$ be the portion of f from u to v. Clearly, the last turn of $f(u,v)$ before v is a left turn, otherwise v would be the leftmost vertex of an unconstrained horizontal chain and (u,v) would be used in the construction of H_y. If $f(u,v)$ contains at least one unconstrained horizontal chain whose leftmost vertex is an s_L-switch of f in G_r, let *uhc* be the last of these chains and x be the leftmost vertex of *uhc* (see Fig. 2.11(a)). A saturating edge (u,x) from G_r is used in the construction of H_y, and the path between u and v in H_y consists of (u,x) and the subpath of $f(u,v)$ from x to v. Note that all the vertical edges of the subpath are traversed according to their direction in H_y (from bottom to top). If $f(u,v)$ does not contain such an unconstrained horizontal chain (see Fig. 2.11(b)), then the path between u and v in H_y is $f(u,v)$ itself. Note, again, that all the vertical edges of this path are traversed according to their direction in H_y (from bottom to top). If not, let x be the top end-vertex of the first vertical edge of $f(u,v)$ traversed from top to bottom; then, the corners associated with v and x would be kitty corners, and H would not be turn-regular. □

Lemma 6 *Let H be a turn-regular orthogonal representation, and let (u,v) and (x,y) be two saturating edges in H_x or H_y crossing each other; then, (u,v) and (x,y) cannot be both from G_r or both from G_ℓ.*

Proof: We prove the claim for H_y; the proof for H_x is similar. In the proof, we denote the corner associated with vertex v as c_v. Suppose, for a contradiction, that (u,v) and (x,y) are both from G_r. We recall that u, v, x and y belong to the same face of H; four cases are possible:

1. u and x are both s_S-switches (see Fig. 2.12(a)). It follows that v and y are both t_L-switches. Let p be the portion of the boundary of f between u and x not containing v and y. Clearly, there must be at least one right turn in p; let w be the corresponding vertex. Then, the $\{c_w, c_y\}$ and $\{c_w, c_y\}$ are two pairs of kitty corners, and H is not turn-regular; a contradiction.

2. u is an s_S-switch and x is a t_L-switch (see Fig. 2.12(b)). It follows that v is an s_L-switch and y is a t_S-switch. Then, $\{c_x, c_v\}$ is a pair of kitty corners, and H is not turn-regular; a contradiction.

3. u is a t_L-switch and x is an s_S-switch. Similar to Case 2.

4. u and x are both t_L-switches. Similar to Case 1.

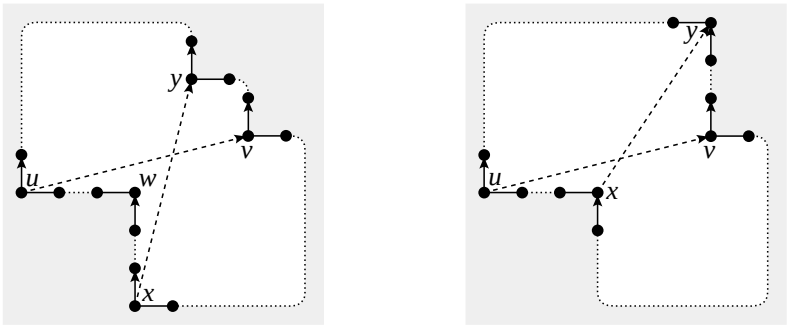
The proof for (u, v) and (x, y) both from G_ℓ is analogous. □

Lemma 7 *Let H be a turn-regular orthogonal representation, and let (u, v) and (x, y) be two saturating edges in H_x crossing each other; then, either $mvc(u) = mvc(x)$ or $mvc(v) = mvc(y)$. Similarly, let (u, v) and (x, y) be two saturating edges in H_y crossing each other; then, either $mhc(u) = mhc(x)$ or $mhc(v) = mhc(y)$.*

Proof: We prove the claim for H_y; the proof for H_x is similar. In the proof, we denote the corner associated with vertex v as c_v. By Lemma 6, (u, v) and (x, y) cannot be both from G_r or G_ℓ. We assume that (u, v) is from G_r and (x, y) is from G_ℓ. We recall that u, v, x and y belong to the same face of H. The proof proceeds by case analysis:

1. u is an s_S-switch in G_r and x is an s_S-switch in G_ℓ (see Fig. 2.13(a)). It follows that v is a t_L-switch in G_r and y is a t_L-switch in G_ℓ. Let p be the portion of the boundary of f between u and x not containing v and y. We have that p contains no right turns; suppose the opposite, for a contradiction, and let w be the corresponding vertex. Either $\{c_w, c_v\}$ or $\{c_w, c_y\}$ is a pair of kitty corners, and H is not turn-regular; a contradiction. Hence, $mhc(u) = mhc(x)$.

2. u is an s_S-switch in G_r and x is a t_L-switch in G_ℓ (see Fig. 2.13(b)). It follows that v is an s_L-switch in G_r and y is a t_S-switch in G_ℓ. Note that, from the construction of H_y, $mhc(v)$ and $mhc(x)$ are unconstrained; let w and z be the other end-vertices of $mhc(v)$ and $mhc(x)$,



(a) (b)

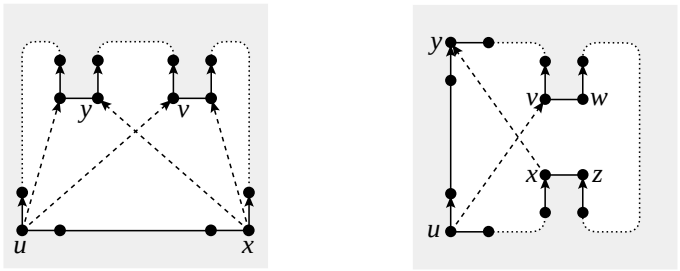
Figure 2.12: Impossible cases of crossing saturating edges in H_y.

respectively. This case is impossible, since $\{c_v, c_z\}$ and $\{c_w, c_x\}$ are two pairs of kitty corners and H is not turn-regular.

3. u is a t_L-switch in G_r and x is an s_S-switch in G_ℓ. Similar to Case 2.

4. u is a t_L-switch in G_r and x is a t_L-switch in G_ℓ. Similar to Case 1; hence, $mhc(v) = mhc(y)$.

The proof for (u, v) from G_ℓ and (x, y) from G_r is analogous. □



(a) (b)

Figure 2.13: Crossing saturating edges of H_y.

Let H be an orthogonal representation and let f be a face of H. Two reflex corners of f are *mutually visible* if there exists a planar drawing of H such that the vertices associated with the two corners can be connected by a straight-line segment that does not cross any edge of f.

Lemma 8 *Let f be a face of an orthogonal representation, c_u be a NE-, or a NW-, or a SW-, or a SE-corner of f and c_v be a NW-, or a SW-, or a SE-, or a NE-corner of f, respectively. If c_u and c_v are mutually visible, then $rotation(c_u, c_v) = 1$.*

Proof: We prove the claim for a NE-corner c_u and a NW-corner c_v. The proofs for the other three cases are similar. Let H be an orthogonal representation and Γ be a planar drawing of H such that vertices u and v, associated with corners c_u and c_v, can be connected by a straight-line segment that does not cross any edge of f. Let Γ' be the drawing obtained from Γ as follows: if $y(u) = y(v)$, we add a horizontal segment between u and v; if $y(u) < y(v)$, we add a polyline from v to u that consists of a horizontal segment, a vertex v' corresponding to a right turn, a vertical segment, a vertex u' corresponding to a left turn, and a horizontal segment; if $y(u) > y(v)$, we add a polyline from v to u that consists of a horizontal segment, a vertex v' corresponding to a left turn, a vertical segment, a vertex u' corresponding to a right turn, and a horizontal segment. Let H' be the corresponding orthogonal representation. We denote by f' and f'' the two faces of H' replacing f, and let f' be the face above edge (u, v). Note that f' is an internal face of H', regardless of f being an internal or an external face of H, and that c_u and c_v are both convex corners in f'. Clearly, $rotation_{f'}(c_v, c_u) = turn_{f'}(c_v) = 1$, since $turn_{f'}(c_{u'})$ and $turn_{f'}(c_{v'})$ (if u' and v' exist) cancel each other out. Thus, by

Properties 4 and 5, $rotation_{f'}(c_u, c_v) = 4 - rotation_{f'}(c_v, c_u) = 3$. Since c_u is a reflex corner in f, we have $rotation_f(c_u, c_v) = rotation_{f'}(c_u, c_v) - turn_{f'}(c_u) + turn_f(c_u) = 3 - 1 + (-1) = 1$ □

Lemma 9 *Let f be a face of a turn-regular orthogonal representation H. Let c_u be a NE-corner (SW-corner) of f, associated with vertex u, and let c_v be a NW-corner (SE-corner) of f, associated with vertex v, such that c_u and c_v are mutually visible; then, there exists a path from v to u (from u to v) in H_x. Similarly, let c_u be a NW-corner (SE-corner) of f and let c_v be a SW-corner (NE-corner) of f, such that c_u and c_v are mutually visible; then, there exists a path from v to u (from u to v) in H_y.*

Proof: We prove the claim for a NE-corner c_u and a NW-corner c_v. The proofs for the other three cases are similar. Let $f(u, v)$ be the portion of the boundary of f from u to v, and let $\{w_0 = u, w_1, \ldots, w_k, w_{k+1} = v\}$ be the sequence of vertices of $f(u, v)$. We assume that $f(u, v)$ does not contain vertices associated with flat corners, since their presence is irrelevant to this proof. Let c_i be the (only) corner associated with vertex w_i. From the definitions of NE- and NW-corners, it follows that $next(u)$ and $prev(v)$ are vertical edges; hence, $f(u, v)$ contains at least four vertices.

We first show that u cannot be followed by more than two consecutive left turns in $f(u, v)$. Suppose, for a contradiction, that vertices w_1, w_2, and w_3 all correspond to left turns (see Fig. 2.14(a)). Since v corresponds to a right turn, $w_3 \neq v$, and thus $k > 3$. We have $rotation(c_u, c_4) = 2$. Since, by Lemma 8, $rotation(c_u, c_v) = 1$, and since the only corners with a negative value of *turn* (exactly -1) are reflex corners, it follows, by Property 5, that there is a vertex w_i (possibly coincident with w_4 itself) in $f(w_4, v)$, whose associated corner is reflex and such that $rotation(c_u, c_i) = 2$. Thus H is not turn-regular; a contradiction.

We now consider the possible cases for w_1, w_2, w_3, and w_4, discard the impossible ones, and prove the claim by induction on the number of vertices of $f(u, v)$. Vertex w_1 cannot correspond to a right turn, since otherwise c_1 and c_v would be kitty corners and H would not be turn-regular. Hence, w_1 corresponds to a left turn. If w_2 corresponds to a left turn as well, then, by the above discussion, w_3 must be a right turn. If $w_3 = v$ (see Fig. 2.14(b)), there clearly exists a path from v to u in H_x; this is the base case of the induction. If $w_3 \neq v$ (see Fig. 2.14(c)), then, since $turn(c_2) = 1$ and $turn(c_3) = -1$, we can remove w_2 and w_3 from the sequence of vertices of $f(u, v)$ without affecting the value of $rotation(c_u, c_v)$, and the claim is proved by the induction hypothesis. Note that this process is similar to the switch collapsing process described and used in Section 2.3.3. Analogously, if w_2 corresponds to a right turn (see Fig. 2.14(d)), then, since $turn(c_1) = 1$ and $turn(c_2) = -1$, we can remove w_1 and w_2 from the sequence of vertices of $f(u, v)$ without affecting the value of $rotation(c_u, c_v)$, and the claim is proved by the induction hypothesis. □

Lemma 10 *Let H be a turn-regular orthogonal representation, and let u and v be two vertices of H. If there is no path between u and v in H_x (H_y), then there is a path between u and v in H_y (H_x).*

Proof: We consider the case in which there is no path between u and v in H_x; the other case is similar. Let s_r and t_r be the source and the sink, respectively, of the complete saturator of G_r,

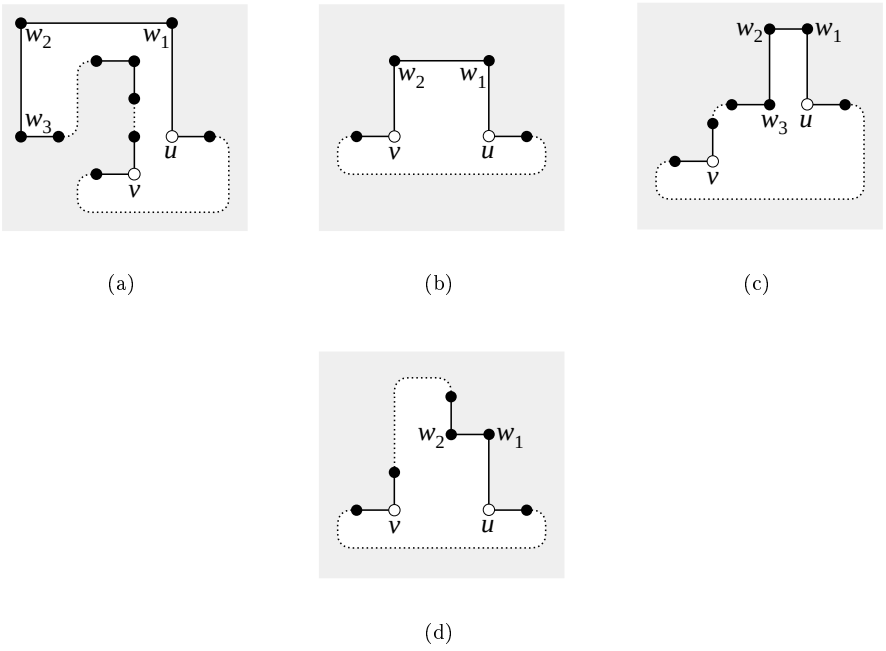


Figure 2.14: Four cases in the proof of Lemma 9.

and let s_ℓ and t_ℓ be the source and the sink, respectively, of the complete saturator of G_ℓ. In order to simplify the proof, instead of H_x and H_y, we consider the partially-directed graphs H'_x and H'_y constructed in the same way as H_x and H_y, but using all the saturating edges from G_r and G_ℓ (except (s_r, t_r) and (s_ℓ, t_ℓ)). Note that the existence of a path between u and v, $u, v \notin \{s_r, t_r, s_\ell, t_\ell\}$, in H'_x (H'_y) implies, by Lemma 5, the existence of a path between u and v, in H_x (H_y).

For each vertex w of H, we define four paths in H'_y, denoted $p_{\mathrm{NE}}(w)$, $p_{\mathrm{NW}}(w)$, $p_{\mathrm{SW}}(w)$, and $p_{\mathrm{SE}}(w)$. Informally speaking, they are paths in H'_y from w to the external face, going in the North-East, North-West, South-West, and South-East direction, respectively. In the rest of the proof, the subpath of path $p_{\mathrm{NE}}(w)$ ($p_{\mathrm{NW}}(w)$) from w to z is denoted by $p_{\mathrm{NE}}(w, z)$ ($p_{\mathrm{NW}}(w, z)$), and the subpath of path $p_{\mathrm{SW}}(w)$ ($p_{\mathrm{SE}}(w)$) from z to w is denoted by $p_{\mathrm{SW}}(z, w)$ ($p_{\mathrm{SE}}(z, w)$). The operator $+$ is used to denote the concatenation of vertices, edges, and subpaths of a given path. The formal definitions of all four paths are given for completeness, although they are very similar.

Path $p_{\mathrm{NE}}(w)$ is a path in H'_y from w to t_r, and is recursively defined as follows: (i) if $w = t_r$, then $p_{\mathrm{NE}}(w) = w$; (ii) otherwise, since the only unsaturated sink switch in G_r is t_r, there exists a vertex z such that (w, z) is either a saturating edge from G_r, or a (directed) vertical edge, or a horizontal edge with z as right end-vertex (tested in this order), and $p_{\mathrm{NE}}(w) = w + (w, z) + p_{\mathrm{NE}}(z)$. Note that $p_{\mathrm{NE}}(w)$ is non-decreasing in both the x- and y-coordinate. Similarly, path $p_{\mathrm{NW}}(w)$ is a path in H'_y from w to t_ℓ, and is recursively defined as follows: (i) if $w = t_\ell$, then $p_{\mathrm{NW}}(w) = w$; (ii) otherwise, since the only unsaturated sink switch in G_ℓ is t_ℓ, there exists a vertex z such that (w, z) is either a saturating edge from G_ℓ, or a (directed) vertical edge, or a horizontal edge with z

as left end-vertex (tested in this order), and $p_{\mathrm{NW}}(w) = w + (w, z) + p_{\mathrm{NW}}(z)$. Note that $p_{\mathrm{NW}}(w)$ is non-increasing in the x-coordinate and non-decreasing in the y-coordinate.

Path $p_{\mathrm{SW}}(w)$ is a path in H'_y from s_r to w, and is recursively defined as follows: (i) if $w = s_r$, then $p_{\mathrm{SW}}(w) = w$; (ii) otherwise, since the only unsaturated source switch in G_r is s_r, there exists a vertex z such that (z, w) is either a saturating edge from G_r, or a (directed) vertical edge, or a horizontal edge with z as left end-vertex (tested in this order), and $p_{\mathrm{SW}}(w) = p_{\mathrm{SW}}(z) + (z, w) + w$. Note that $p_{\mathrm{NE}}(w)$ is non-decreasing in both the x- and y-coordinate. Similarly, path $p_{\mathrm{SE}}(w)$ is a path in H'_y from s_ℓ to w, and is recursively defined as follows: (i) if $w = s_\ell$, then $p_{\mathrm{SE}}(w) = w$; (ii) otherwise, since the only unsaturated source switch in G_ℓ is s_ℓ, there exists a vertex z such that (z, w) is either a saturating edge from G_ℓ, or a (directed) vertical edge, or a horizontal edge with z as right end-vertex (tested in this order), and $p_{\mathrm{SE}}(w) = p_{\mathrm{SE}}(z) + (z, w) + w$. Note that $p_{\mathrm{SE}}(w)$ is non-increasing in the x-coordinate and non-decreasing in the y-coordinate.

Let w_{NE} (w_{NW}) be the next-to-last vertex of $p_{\mathrm{NE}}(w)$ ($p_{\mathrm{NW}}(w)$), and let w_{SW} (w_{SE}) be the second vertex of $p_{\mathrm{SW}}(w)$ ($p_{\mathrm{SE}}(w)$). Note that w_{NE}, w_{NW}, w_{SW}, and w_{SE} are all vertices of the external face of H. For each vertex w of H, paths $p_{\mathrm{NE}}(w)$, $p_{\mathrm{NW}}(w)$, $p_{\mathrm{SW}}(w)$, and $p_{\mathrm{SE}}(w)$ define four regions in H'_y (see Fig. 2.15): $R_t(w)$ is the subgraph of H'_y with external face formed by $p_{\mathrm{NE}}(w)$, $p_{\mathrm{NW}}(w)$, and the portion of external face of H between w_{NE} and w_{NW}; $R_l(w)$ is the subgraph of H'_y with external face formed by $p_{\mathrm{NW}}(w)$, $p_{\mathrm{SW}}(w)$, and the portion of external face of H between w_{NW} and w_{SW}; $R_b(w)$ is the subgraph of H'_y with external face formed by $p_{\mathrm{SW}}(w)$, $p_{\mathrm{SE}}(w)$, and the portion of external face of H between w_{SW} and w_{SE}; $R_r(w)$ is the subgraph of H'_y with external face formed by $p_{\mathrm{SE}}(w)$, $p_{\mathrm{NE}}(w)$, and the portion of external face of H between w_{SE} and w_{NE}.

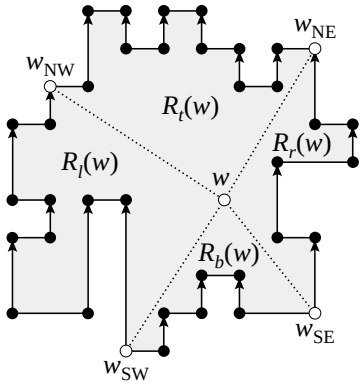


Figure 2.15: The four regions of vertex w.

We now show that if $u \in R_b(v)$, then there is a path from u to v in H'_y. We consider paths $p_{\mathrm{NE}}(u)$, $p_{\mathrm{NW}}(u)$, $p_{\mathrm{SW}}(v)$, and $p_{\mathrm{SE}}(v)$. By the definition of these paths, at least one of the following six cases applies: (i) $p_{\mathrm{NE}}(u)$ and $p_{\mathrm{SE}}(v)$ have a vertex q in common; then, $p_{\mathrm{NE}}(u, q) + p_{\mathrm{SE}}(q, v)$ is a path from u to v in H'_y. (ii) $p_{\mathrm{NE}}(u)$ and $p_{\mathrm{SE}}(v)$ cross, that is there is a saturating edge (w, x) from G_r in $p_{\mathrm{NE}}(u)$ and a saturating edge (y, z) from G_ℓ in $p_{\mathrm{SE}}(u)$ that cross each other; thus, by Lemma 7, either $mhc(w) = mhc(y)$ or $mhc(x) = mhc(z)$; w.l.o.g., assume that the former holds,

and let $mhc(w,x)$ be the portion of the common maximal horizontal chain from w to x; then, $p_{\rm NE}(u,w) + mhc(w,x) + p_{\rm SE}(x,v)$ is a path from u to v in H'_y. *(iii)* $u_{\rm NE} \in R_b(v)$; note that $u_{\rm NE}$ is a t_L-switch of the external face of G_r, since it is adjacent to t_r, and that $v_{\rm SE}$ is an s_L-switch of the external face of G_ℓ, since it is adjacent to s_ℓ. Hence, $u_{\rm NE}$ is a SW-corner and $v_{\rm SE}$ is a NW-corner of the external face of H, and they are clearly mutually visible. By Lemma 9, there exists a path from $u_{\rm NE}$ to $v_{\rm SE}$ in H'_y. Let $f(u_{\rm NE}, v_{\rm SE})$ be this path (note that it is a portion of the external face of H); then, $p_{\rm NE}(u, u_{\rm NE}) + f(u_{\rm NE}, v_{\rm SE}) + p_{\rm SE}(v_{\rm SE}, v)$ is a path from u to v in H'_y. Cases *(iv–vi)* are similar to Cases *(i–iii)*, and involve $p_{\rm NW}(u)$ and $p_{\rm SW}(v)$.

If $u \in R_t(v)$, then there is a path from v to u in H'_y. This can be proved in a similar way by considering paths $p_{\rm SW}(u)$, $p_{\rm SE}(u)$, $p_{\rm NE}(v)$, and $p_{\rm NW}(v)$.

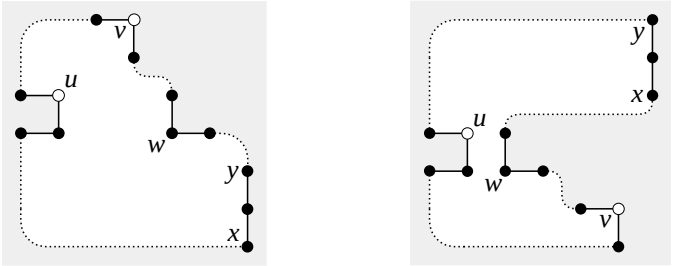
Finally, with the same technique, it is possible to prove that if $u \in R_l(v)$ ($u \in R_r(v)$), then there is a path from u to v (from v to u) in H'_x. □

2.5 Turn-Regularity and Orthogonal Relations

In this section we use graphs H_x and H_y to characterize all possible orthogonal relations in a turn-regular orthogonal representation. This leads to a characterization of turn-regular orthogonal representations in terms of orthogonal relations. We denote by $u \to v$ a directed path from vertex u to vertex v in H_x containing at least a horizontal edge or in H_y containing at least a vertical edge, and by $u \not\to v$ the absence of such a path from vertex u to vertex v.

Lemma 11 *If H_x or H_y contains a saturating edge (u,v) from G_r, then $u <_x v$ and $u <_y v$. If H_x or H_y contains a saturating edge (u,v) from G_ℓ, then $v <_x u$ and $u <_y v$.*

Proof: We prove that, if H_x contains a saturating edge (u,v) from G_r, then u is left of and below v in any planar drawing of H; the proofs for a saturating edge (u,v) from G_ℓ, and for H_y instead of H_x are similar. In particular, we show this for a saturating edge from a t_L-switch to a t_S-switch; the proof for a saturating edge from an s_S-switch to an s_L-switch is similar.



(a) (b)

Figure 2.16: The two cases in the proof of Lemma 11.

Let f be a face of G_r, u be a t_L-switch of f, and v be a t_S-switch of f. We begin by showing that u is left of v in any planar drawing of H. Since u is a t_L-switch of f in G_r, it cannot be one of the rightmost vertices of f in any planar drawing of H. On the other hand, v is one of the rightmost vertices of f in any planar drawing of H. Suppose, for a contradiction, that there exists a planar drawing Γ of H in which v is not one of the rightmost vertices of f. Then there exists a maximal vertical chain mvc in f, containing at least one edge, whose vertices are right of v in Γ; let x and y be the bottommost and topmost vertices of mvc, respectively. Two cases are possible: either mvc precedes v in the traversal of f starting at u (see Fig. 2.16(a)), or mvc follows v (see Fig. 2.16(b)). In the first case, there must be at least one right turn in the path from y to v; in the second case, there must be at least two right turns in the path from v to x. In both cases, there exists a vertex w corresponding to one of these right turns such that its associated corner and the corner associated with u are kitty corners, thus violating the turn-regularity of H; a contradiction. Thus, we have proved that u is left of v in any planar drawing of H. The proof that u is below v in any planar drawing of H is similar, hence the thesis. □

Lemma 12 *Let Γ be a planar drawing of a turn-regular orthogonal representation H, and let mvc_1 and mvc_2 be two maximal vertical chains of Γ (possibly consisting of a single vertex) such that mvc_1 is to the left of mvc_2. If the endpoint of one chain can be connected to any point of the other by a horizontal segment that does not cross any other vertical chain of Γ, then there exists a path in H_x connecting any vertex of mvc_1 to any vertex of mvc_2.*

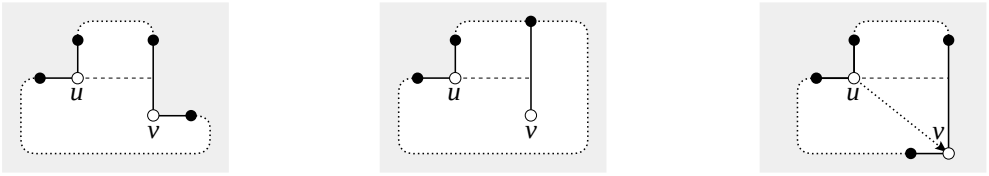
Proof: We consider only the case in which the bottom endpoint of mvc_1 can be connected to mvc_2; the other cases are similar. In this proof, we call two such chains *consecutive*. Clearly, if there is a path from some vertex of mvc_1 to some vertex of mvc_2 in H_x, there is a path from any vertex of mvc_1 to any vertex of mvc_2, because vertical edges of H_x are undirected. Let u be the bottom end-vertex of mvc_1; u and the object (either an edge or a vertex) on mvc_2 at the same y-coordinate of u belong to the same face f of H. Vertex u may correspond to a left turn, a right turn, or a U-turn of f; we consider the three cases separately.

If u corresponds to a left turn of f, then the first turn following u along the boundary of f corresponds to a vertex v of mvc_2, since otherwise the horizontal segment connecting u to mvc_2 would cross a vertical chain of Γ different from mvc_1 and mvc_2. The path from u to v in H_x is the portion of the boundary of f from u to v.

If u corresponds to a right turn of f, let c_u be the corner associated with u, and let vc_2 be the portion of mvc_2 that is part of the boundary of f. We first observe that vc_2 cannot be a single, degree one vertex, namely vertex v, since c_u and the second corner associated with v would be kitty corners, and H would not be turn-regular. Thus, let v be the bottom end-vertex of vc_2. Two cases are possible: (*i*) v corresponds to a right turn (see Fig. 2.17(a)) or a U-turn of f (see Fig. 2.17(b)). Then the corner c_v associated with v (the second one if v corresponds to a U-turn) is a NE-corner, and since c_u is a NW-corner, and c_u and c_v are mutually visible, there exists a path from u to v in H_x by Lemma 9. (*ii*) v corresponds to a left turn of f (see Fig. 2.17(c)). Then v is an s_S-switch

in G_ℓ, and since u is an s_L-switch in G_ℓ, there exists a saturating edge (v, u) in G_ℓ (recall that, by Theorems 2 and 3, if H is turn regular, then G_ℓ has a unique complete saturator). If the saturating edge (v, u), with its orientation reversed, has been used in the construction of H_x, then the path from u to v in H_x is (v, u) itself; otherwise, such path exists by Lemma 5.

If u corresponds to a U-turn of f, the proof is analogous to that of the previous case, once we let c_u be the first corner associated with u, if the only edge incident with u is vertical, or the second corner associated with u, otherwise. □



(a) (b) (c)

Figure 2.17: Three cases in the proof of Lemma 12. The horizontal segment connecting u to mvc_2 is represented as a dashed segment.

Lemma 13 *For each pair $\{u, v\}$ of vertices of H_x the following conditions hold:*

1. *$mvc(u) = mvc(v)$ if and only if $u =_x v$*
2. *$u \to v$ if and only if $u <_x v$*
3. *$v \to u$ if and only if $v <_x u$*
4. *$mvc(u) \neq mvc(v)$, $u \not\to v$, and $v \not\to u$ if and only if no x-relation can be established between u and v.*

Proof: *Only if.* We first prove that Conditions 1–4 are necessary.

Condition 1. If u and v belong to the same maximal vertical chain, they are clearly drawn with the same x-coordinate in any planar drawing of H.

Condition 2. We prove that $u <_x v$ by induction on the number of edges of the path. If $u \to v$ consists of only one edge $e = (u, v)$, then, since u and v do not belong to the same maximal vertical chain, there are three possible cases: (*i*) e is a horizontal edge of H; then, v is clearly right of u in any planar drawing of H. (*ii*) e is a saturating edge from G_r; then, by Lemma 11, u is left of v in any planar drawing of H; (*iii*) e is a saturating edge from G_ℓ with its orientation reversed; then, (v, u) is a saturating edge of G_ℓ and, again by Lemma 11, u is left of v in any planar drawing of H.

We now suppose that $u <_x v$ for each path $u \to v$ consisting of $k - 1$ edges, $k > 2$, and prove the claim for a path $u \to v$ consisting of k edges. Let (w, v) be the last edge of of the path. Two cases are possible: (*i*) w belongs to the same maximal vertical chain of v; then, by Condition 1, we have

$w =_x v$, and, by the inductive hypothesis, we have $u <_x w$, which implies $u <_x v$; (*ii*) w does not belong to the same maximal vertical chain of v; then, by the inductive hypothesis, we have $w <_x v$ and $u <_x w$, which implies $u <_x v$.

Condition 3. Analogous to the proof of Condition 2.

Condition 4. Let Γ be a planar orthogonal drawing of H. W.l.o.g., we assume that $x(u) \leq x(v)$ and $y(u) > y(v)$ in Γ. We show how to construct from Γ a different planar drawing of H such that $x(u) > x(v)$, thus showing that no x-relation can be established between u and v.

We consider the two vertical lines $x_\ell = x(u) - \frac{1}{2}$ and $x_r = x(v) + \frac{1}{2}$. We denote by Γ_c the portion of Γ between x_ℓ and x_r. We define a *moving-line* in Γ with the following properties:

- it is directed and orthogonal,
- it starts at the intersection u' between x_ℓ and a horizontal line y_u through u and ends at the intersection v' between x_r and a horizontal line y_v through v,
- it is entirely contained within the box B defined by x_ℓ, x_r, y_u, and y_v,
- it does not intersect any vertical edges of Γ_c, and
- its first and last segments are vertical.

Using compaction techniques developed for VLSI layout [71, 95, 107], the moving-line guarantees that it is possible to stretch and shift parts of Γ to obtain a planar orthogonal drawing such that $x(u) > x(v)$.

We construct a moving-line J such that all of its bends are displaced a half-unit from grid-points. This implies that the only intersections between J and Γ are between vertical segments of J and horizontal segments of Γ, and that the first and last segments of J are vertical, as required. J is constructed as follows, where the possibility or impossibility of traveling in a certain direction is given by the above properties. Travel downward from u' until it is possible to travel right. Travel right as far as possible until either x_r is reached or a vertical segment of Γ is half-unit distant. If x_r is half-unit distant, travel downward to v', and the construction of J is complete. Otherwise, travel either downward or upward until it is again possible to travel right. Choose the upward direction only if it is not possible to travel right before reaching y_v when going down. If it is not possible to travel rightward by going either up or down (staying within B), there is no moving-line. Otherwise, continue the process until v' is reached. Observe that this method will construct an x-monotone moving-line if any moving-line exists, and that it will construct a y-monotone moving-line if there is any possible y-monotone moving-line.

If there is no moving-line, then there must be a vertical chain vc of Γ that intersects both y_u and y_v. Since both $mvc(u)$ and vc intersect y_u, either Lemma 12 applies and there is a path from u to any vertex of vc, or there is another vertical chain vc' between $mvc(u)$ and vc that also crosses y_u. The argument can then be applied to $mvc(u)$ and vc', and to vc' and vc, yielding a path from u to vc. A similar argument can be applied to vc and $mvc(v)$ because both chains intersect y_v, yielding a path from vc to v, and thus a path from u to v.

We now show that if J is not y-monotone, there is a path from u to v in H_x. Consider the sequence of vertical chains $C = \{vc_1, \ldots, vc_k\}$ that define J: $vc_1 = mvc(u)$, $vc_k = mvc(v)$, and the others are those that determine upward or downward turns of J, ordered from left to right. Let vc_i and vc_{i+1} be two (consecutive) elements of C. Then there is a horizontal segment connecting one endpoint of vc_i to vc_{i+1} that either crosses no other vertical chains of Γ or touches only one of their endpoints. By (possibly repeated) application of Lemma 12, there is a path from any vertex of vc_i to any vertex of vc_{i+1}. If J is not y-monotone, there is at least one chain in C that defines an upward turn in J; let vc_j be the first such chain. This chain vc_j must cross y_v, or else it would have been possible to make a downward turn instead. By repeatedly applying the previous argument, there is a path from $vc_1 = mvc(u)$ to vc_j. We now must show that there is a path from vc_j to $vc_k = mvc(v)$. Since both vc_j and vc_k intersect y_v, the argument used in the previous paragraph applies, resulting in a path from vc_j to vc_k, and thus a path from u to v.

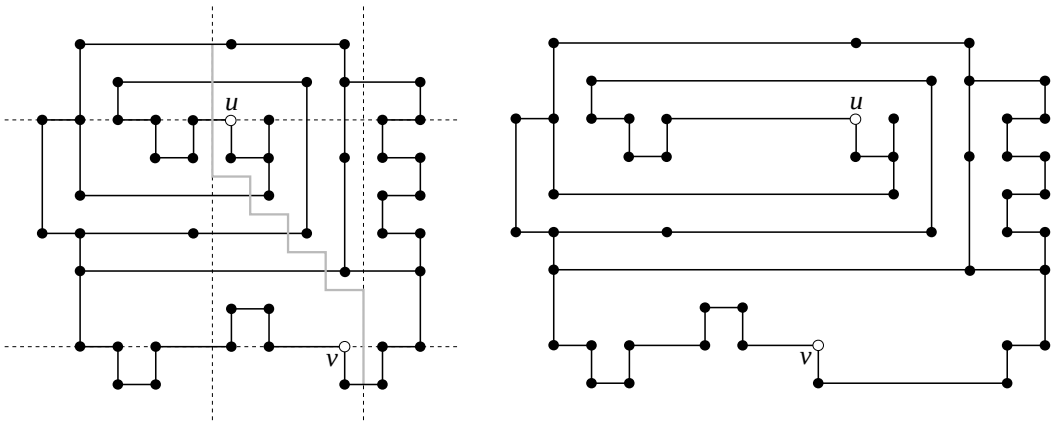


Figure 2.18: (a) A planar orthogonal drawing Γ with the extended moving-line for vertices u and v. (b) The stretched version of Γ.

As a result, there exists a moving-line J that is monotone in both x and y. This moving-line can then be used to stretch Γ horizontally so that $x(u) > x(v)$. Extend J by continuing its first vertical segment (i.e., the one that starts at u') in the positive y direction and its last vertical segment (i.e., the one that ends at v') in the negative y direction until they both intersect the external face of Γ (see Fig. 2.18(a)). Also, let $d_x = x(v) - x(u)$. Stretch Γ by increasing the x-coordinates of the vertices to the right of J by an amount $d > d_x$. As a result, a new planar orthogonal drawing is constructed in which $x(u) > x(v)$ (see Fig. 2.18(b)). This implies that there is no x-relation between u and v.

If. The sufficiency of Conditions 1–4 follows from the completeness and mutual exclusiveness of the four cases. □

As an example, we identify in the graph H_x shown in Fig. 2.10(a) three pairs of vertices corresponding to the various cases of Lemma 13: u_1 and v_1 belong to the same maximal vertical chain, there exists a directed path from u_2 to v_2, while there is neither a path between u_3 and v_3, nor they belong to the same maximal vertical chain.

The proof of the following lemma is similar to that of Lemma 13, and hence is omitted.

Lemma 14 *For each pair $\{u, v\}$ of vertices of H_y the following conditions hold:*

1. *$mhc(u) = mhc(v)$ if and only if $u =_y v$*
2. *$u \rightarrow v$ if and only if $u <_y v$*
3. *$v \rightarrow u$ if and only if $v <_y u$*
4. *$mhc(u) \neq mhc(v)$, $u \not\rightarrow v$, and $v \not\rightarrow u$ if and only if no y-relation can be established between u and v.*

As an example, we identify in the graph H_y shown in Fig. 2.10(b) three pairs of vertices corresponding to the various cases of Lemma 13: u_1 and v_2 belong to the same maximal horizontal chain, there exists a directed path from u_2 to v_3, while there is neither a directed path between u_3 and v_1, nor they belong to the same maximal vertical chain.

Lemma 15 *Let G be an embedded 4-planar graph and H be a turn-regular orthogonal representation of G. For each pair $\{u, v\}$ of vertices of G, exactly one orthogonal relation holds.*

Proof: By considering all possible combinations of the four (mutually exclusive) cases of Lemma 13 for H_x with the four (mutually exclusive) cases of Lemma 14 for H_y, we obtain the sixteen (mutually exclusive) cases shown in Table 2.1. For twelve of these cases, the orthogonal relations are simply obtained by taking the "logical and" of the corresponding relations given in Lemmas 13 and 14.

We now show that the remaining four cases are impossible. Clearly, if $mvc(u) = mvc(v)$ in H_x, then there is a path $u \rightarrow v$ or $v \rightarrow u$ in H_y, and if $mhc(u) = mhc(v)$ in H_y, then there is a path $u \rightarrow v$ or $v \rightarrow u$ in H_x. As for the "otherwise–otherwise" case, it is impossible by Lemma 10. □

We are interested in those orthogonal representations for which there is an orthogonal relation between each pair of vertices. As the following theorem shows, this class of orthogonal representations is characterized by turn-regularity.

Theorem 5 *An orthogonal representation H is turn-regular if and only if there is an orthogonal relation between every two vertices of H.*

Proof: *Only if.* It follows directly from Lemma 15.

If. Suppose, for a contradiction, that there is an orthogonal relation between every two vertices of H and that H is not turn-regular. Hence, there exists a face f of H with two vertices u and v, whose associated corners c_u and c_v are kitty corners. Assume, w.l.o.g., that the kitty corners

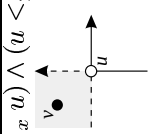
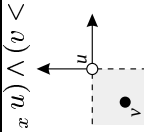
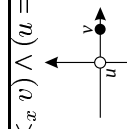
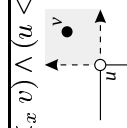
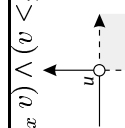
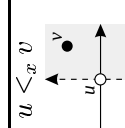
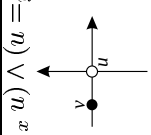
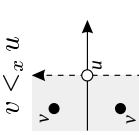
H_x \ H_y	$mhc(u) = mhc(v)$	$u \to v$	$v \to u$	otherwise
$mvc(u) = mvc(v)$	impossible	$(u =_x v) \wedge (u <_y v)$ 	$(u =_x v) \wedge (v <_y u)$ 	impossible
$u \to v$	$(u <_x v) \wedge (u =_y v)$ 	$(u <_x v) \wedge (u <_y v)$ 	$(u <_x v) \wedge (v <_y u)$ 	$u <_x v$ 
$v \to u$	$(v <_x u) \wedge (u =_y v)$ 	$(v <_x u) \wedge (u <_y v)$	$(v <_x u) \wedge (v <_y u)$	$v <_x u$ 
otherwise	impossible	$u <_y v$	$v <_y u$	impossible

Table 2.1: Orthogonal relations for a pair $\{u, v\}$ of vertices in H.

form a SW-NE pair (see Fig. 2.8(a)). We generate two new orthogonal representations H_1 and H_2 as follows. H_1 is obtained by connecting u and v with a horizontal edge such that u is the left end-vertex of the new edge and v is the right end-vertex. H_2 is obtained by connecting u and v with a vertical edge such that u is the bottom end-vertex and v is the top end-vertex. Note that adding these edges is always possible because u is not the left or bottom end-vertex of an edge in H, and v is not the right or top end-vertex of an edge in H.

H_1 and H_2 are orthogonal representations, since they satisfy Properties 1 and 2. We show this for an internal face of H_1; the proofs for the external face of H_1 and for H_2 are similar. We assume that f contains no vertex of degree one and no multiple occurrences of the same vertex; if not, we can use the same expansion technique adopted for the proofs in Section 2.3.3. Note that the only vertices affected by the insertion of the new edge are u and v, and that the only face is f. Let f' and f'' be the two faces of H_1 replacing f, and let f' be the face below edge (u, v) and f'' be the face above. The angle at u in f' is equal to $\pi/2$, while the angle at u in f'' is equal to π. Similarly, the angle at v in f' is equal to π, while the angle at v in f'' is equal to $\pi/2$. Property 1 is clearly satisfied by u and v, since the angles at u and v in f are both $3\pi/2$. As for Property 2, we prove that it is satisfied by f'; the proof for f'' is analogous. Let p be the portion of f from u (excluded) to v (excluded), n_l be the number of left turns in p, n_f be the number of flat turns in p, and n_r be the number of right turns in p. Note that the vertices of f in p are also vertices of f', and that, in addition, f' contains u and v. Thus, Property 2 is satisfied by f' if

$$\begin{aligned} n_l \cdot \pi/2 + n_f \cdot \pi + n_r \cdot 3\pi/2 + \pi/2 + \pi &= [2(n_l + n_f + n_r + 2) - 4] \cdot \pi/2 \\ (n_l + 2n_f + 3n_r + 1 + 2) \cdot \pi/2 &= (2n_l + 2n_f + 2n_r) \cdot \pi/2 \\ n_l + 3n_r + 3 &= 2n_l + 2n_r \\ n_l - n_r &= 3 \end{aligned}$$

And this is indeed the case. Let w be the first vertex of f after u, and let c_w be its associated corner. Since c_u and c_v are kitty corners in f, $rotation_f(c_u, c_v) = 2$, and thus $rotation_{f'}(c_w, c_v) = rotation_f(c_w, c_v) = 3$. That is, from the definition of *rotation*, the number of left turns in p minus the number of right turns in p is equal to three.

Since the edge connecting u and v in H_1 is horizontal, it follows that u must be left of and y-aligned with v in any planar drawing of H_1. Similarly, u must be below and x-aligned with v in any planar drawing of H_2. Also, note that a planar drawing of H can be obtained from any planar drawing of H_1 or H_2 by simply removing the extra edge. As a result, there is not an orthogonal relation between u and v; a contradiction. □

2.6 Turn-Regularity and Drawing Algorithms

In this section we first study the problem of efficiently checking whether an orthogonal representation is turn-regular. Then we show how an optimal area orthogonal drawing of a turn-regular orthogonal representation can be computed.

Theorem 6 *A turn-regular orthogonal representation with n vertices and bends can be recognized in $O(n)$ time and space.*

Proof: Let H be an orthogonal representation and let f be a face of H with n_f vertices. We show how to test whether f is turn-regular in $O(n_f)$ time and space. Since, for a planar graph, $\sum_f n_f = O(n)$, this proves the claim.

We first consider the case in which f is an internal face. We index the k reflex corners of f from c_1 to c_k, according to a counterclockwise visit of the boundary of f from an arbitrary vertex. We construct a circular list L of k elements, and we set its i-th element $L(i) = \mathit{rotation}(c_1, c_{i+1})$, where, by convention, $c_{k+1} = c_1$. Since $k = O(n_f)$, the size of L is linear with the number of vertices of f. Also, for each $1 \leq i \leq k$ we have $L(i) = O(n_f)$, because each vertex of f is associated with one or two corners, and for each corner c we have $-1 \leq \mathit{turn}(c) \leq 1$. Clearly, the construction of L requires linear time. By Property 6, in order to test the turn-regularity of f, we must verify whether there exist two indices $1 \leq i < j \leq k$ such that $L(j) - L(i) = \mathit{rotation}(c_1, c_{j+1}) - \mathit{rotation}(c_1, c_{i+1}) = \mathit{rotation}(c_{i+1}, c_{j+1}) = 2$.

Let min and max be the minimum and maximum values stored in L, respectively. We construct an array A of $\mathit{max} - \mathit{min} + 3$ boolean elements whose index is in the range $\mathit{min} - 2 \ \ldots \ \mathit{max}$. We denote the i-th element of A by $A(i)$ and we set it equal to true if there exists in L an element whose value is i, equal to false otherwise.

The algorithm to test whether f is turn-regular consists of the following test for each element j of L. Let ρ be the value of $L(j)$; if $A(\rho - 2)$ is true, then there exists an element $i < j$ of L such that $L(i) = \rho - 2$. Hence, $L(j) - L(i) = 2$, c_{i+1} and c_{j+1} are kitty corners, and f is not turn-regular. If for each element of L the result of the above test is false, then f is turn-regular.

We now consider the case in which f is the external face. By Property 6, in order to test the turn-regularity of f, we must verify whether there exist two indices $1 \leq i < j \leq k$ such that either $L(j) - L(i) = 2$ or $L(j) - L(i) = -6$. Thus, the array A consists of $\mathit{max} - \mathit{min} + 9$ boolean elements, its index is in the range $\mathit{min} - 2 \ \ldots \ \mathit{max} + 6$, and, in addition to $A(\rho - 2)$, we also test whether $A(\rho + 6)$ is true.

Since verifying the value of an element of A requires constant time, and the number of elements of L is $O(n_f)$, the overall procedure requires linear time. □

The optimal area drawings that we want to compute are planar. The next lemma guarantees the planarity of drawings that satisfy the orthogonal relations of a turn-regular orthogonal representation.

Theorem 7 *Let H be a turn-regular orthogonal representation, and let Γ be an orthogonal drawing of H such that, for each pair $\{u, v\}$ of vertices of H, the orthogonal relation between u and v is satisfied. Then Γ is planar.*

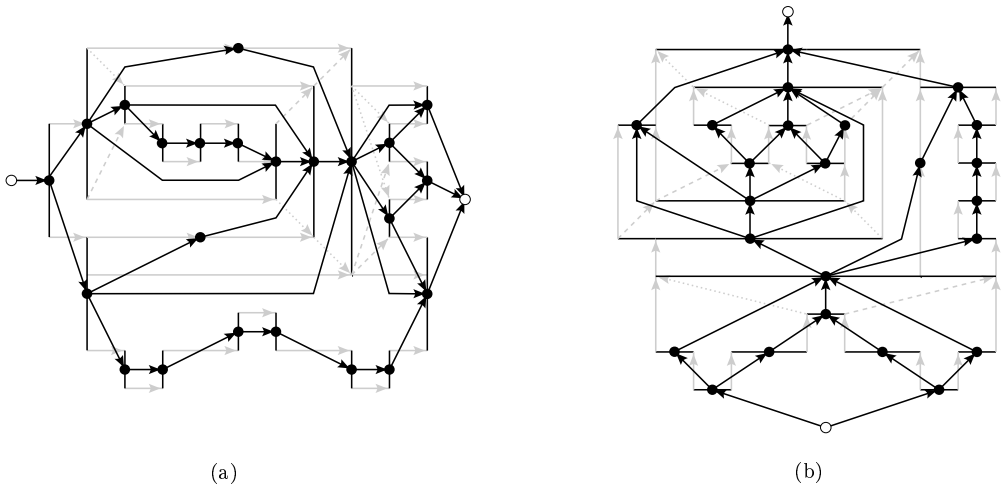
Proof: Since H is a turn-regular orthogonal representation, by Lemma 15 for each pair of vertices of H exactly one orthogonal relation is satisfied in all possible planar drawings of H. Suppose, for a contradiction, that there exists a drawing Γ of H such that:

- Γ is a non-planar drawing of H, and
- Γ satisfies all orthogonal relations defined by the turn-regularity of H.

Let (u, v) and (w, z) be two edges of Γ that cross each other. We assume that (u, v) is a vertical edge with u below v, and that (w, z) is a horizontal edge with w left of z. The proof for the case of overlapping vertices or edges is similar. Since Γ satisfies all orthogonal relations defined by the turn-regularity of H and, by Lemma 15, for each pair of vertices of H exactly one orthogonal relation is satisfied in all possible planar drawings of H, we conclude that the following orthogonal relations hold: (i) $u <_x z$, (ii) $w <_x u$, (iii) $z <_y v$, and (iv) $u <_y w$.

Since H is a planar orthogonal representation, there exists at least one planar drawing Γ' of H. Since in Γ' the pair of edges (u, v) and (w, z) do not cross, the four orthogonal relations given above cannot be simultaneously satisfied, contradicting the fact that in a turn-regular orthogonal representation exactly one orthogonal relation holds between any two vertices. □

We are now ready to present two different algorithms that compute optimal area drawings of turn-regular orthogonal representations. These algorithms are variations of the two compaction procedures described in [54]. For the first algorithm, we define two digraphs, denoted D_x and D_y. D_x is obtained from H_x by shrinking each maximal vertical chain to a single vertex, by removing possible multiple edges, and by adding a super-source and a super-sink (see Fig 2.19(a)). Thus, there is a one-to-one correspondence between maximal vertical chains of H_x and vertices of D_x, and a many-to-one correspondence between directed edge of H_x and edges of D_x. Note that in the shrinking process we "preserve the embedding," i.e., the circular ordering of the edges around each vertex v of D_x is induced by the circular ordering of the directed edges "around" the maximal vertical chain of H_x corresponding to v. D_y is obtained analogously from H_y by shrinking the maximal horizontal chains (see Fig 2.19(b)).



(a) (b)

Figure 2.19: (a) D_x. (b) D_y. Both graphs are obtained from graphs H_x and H_y shown in Fig. 2.10.

Property 10 *D_x and D_y are planar st-digraphs.*

Proof: We first prove that D_x and D_y are planar. In particular, we prove the planarity of D_y; the proof for D_x is analogous. Let D be the graph obtained from D_y by removing the edges that correspond to saturating edges in H_y. Note that, if we ignore the direction of the edges, D can be thought of as the result of the shrinking process applied to H. Since H is an embedded planar graph, D is an embedded planar graph as well. In particular, there is a one-to-one correspondence between internal faces of H and faces of D, and if the end-vertices of a saturating edge e in H_y belong to face f of H, the end-vertices of the directed edge of D_y corresponding to e belong to the face of D corresponding to f. Let (u, v) and (w, z) be two saturating edges in H_y that cross each other. We recall that u, v, w, and z belong to the same face f of H. Thus, from the discussion above, we can concentrate our attention on f alone. We prove that the crossing "disappears" during the shrinking process through which D_y is obtained. By Lemma 7, either $mhc(u) = mhc(w)$ or $mhc(v) = mhc(z)$. It follows that either u and w, or v and z are shrunk to the same vertex in D_y and the crossing "disappears" in the process.

It remains to prove that D_x and D_y are st-digraphs. The non-vertical edges of H_x are all directed from left to right, and the non-horizontal edges of H_y are all directed from bottom to top. Hence, by construction, D_x and D_y are acyclic. And still by construction, they have a single source and a single sink. □

Theorem 8 *Let H be a turn-regular orthogonal representation with n vertices and bends. A planar orthogonal drawing of H with optimal area can be constructed in $O(n)$ time and space.*

Proof: We recall that there is a one-to-one correspondence between maximal vertical chains of H_x (and hence of H) and vertices of D_x; similarly, there is a one-to-one correspondence between maximal horizontal chains of H_y (and hence of H) and vertices of D_y.

We compute the x-coordinates of the vertical segments representing the maximal vertical chains of H as follows. We assign unit weights to the edges of D_x and compute an optimal weighted topological numbering X of D_x (see page 89 of [54]). We then set the length of each horizontal directed edge (u, v) of H equal to $X(v') - X(u')$, where u' and v' are the vertices of D_x representing the maximal vertical chains of H containing u and v, respectively. In the same way, it is possible to compute the y-coordinates of the horizontal segments representing the maximal horizontal chains of H. Let Γ be the resulting drawing.

Lemma 13 shows that H_x represents all the x-relations between vertices of H. This information is represented by D_x, as well, since there is a many-to-one correspondence between directed edge of H_x and edges of D_x. A similar argument holds for D_y. Since the edges of D_x and D_y are assigned positive weights, and since X and Y are weighted topological numberings of D_x and D_y, the x-relations and y-relations between every two vertices of H are satisfied in Γ. Thus, by Theorem 7, Γ is planar.

Since the edges of D_x and D_y are assigned unit weights, and since the weighted topological numbering X and Y are both optimal, the width and height of Γ are both minimum. Hence, Γ has

optimal area.

The time complexity of the algorithm is $O(n)$ since the number of vertices of D_x and of D_y are both not greater than the number of vertices of H, and computing an optimal weighted topological numbering of an n-vertex planar st-digraph requires $O(n)$ time and space. □

Theorem 9 *Let H be a turn-regular orthogonal representation with n vertices and bends. A planar orthogonal drawing of H with optimal area $\mathcal{A}$ and whose total edge length is optimal among all drawings with area $\mathcal{A}$ can be constructed in $O(n^{7/4} \log n)$ time and $O(n)$ space.*

Proof: To compute a planar orthogonal drawing with minimum area $\mathcal{A}$ and whose total edge length is minimum among all drawings with area $\mathcal{A}$, we use a flow technique similar to that described in [54]. Let H be a turn-regular orthogonal representation with n vertices. From Theorem 4, we know that the embedded planar st-digraphs D_x and D_y are uniquely determined. As seen in the proof of Theorem 8, a planar orthogonal drawing of H with optimal area can be found by computing an optimal weighted topological numbering on D_x and D_y, independently. This means that the width w and the height h of the drawing can be minimized independently.

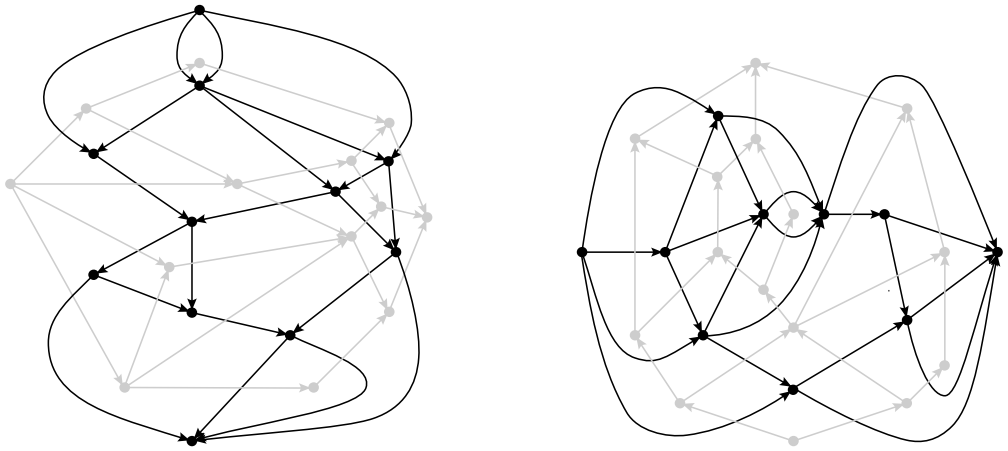
We construct two flow networks N_x and N_y associated with D_x and D_y, respectively, and show that the cost of any integer feasible flow with value w on N_x plus the cost of any integer feasible flow with value h on N_y is equal to the total edge length of a planar orthogonal drawing of H with width w and height h. Consider the embedded planar st-digraph D_y, and consider the external face of D_y split into two regions, the "left" external face and the "right" external face. We construct N_y as follows (see Fig. 2.20(b)):

- For each internal face f of D_y, we consider a node v_f in N_y.
- For the external face of D_y, we consider two nodes in N_y, one for the "left" external face and one for the "right" external face.
- For each edge e of D_y, let f_l be the face to the left of e and f_r be the face to the right of e. We consider a dual arc from v_{f_l} to v_{f_r} in N_y. The upper capacity of this arc is set to $+\infty$ and the lower capacity is set to 1. Finally, the cost of the arc is set to 1 if e corresponds to an edge of H and to 0 if e corresponds to a saturating edge.

Since all arcs of N_y have an infinite upper capacity, there always exists an integer feasible flow with value h in N_y. Each unit of flow on an arc of N_y corresponds to a unit of length of the dual edge in D_y. In particular, the flow on each arc with cost 1 in N_y corresponds to the length of a vertical edge of H. Thus, computing a minimum cost flow with value h on N_y corresponds to minimizing the total length of the vertical edges of H in an orthogonal drawing with height h.

The construction of N_x is analogous to that of N_y, and computing a minimum cost flow with value w on N_x corresponds to minimizing the total length of the horizontal edges of H in an orthogonal drawing with width w.

In order to prove that the obtained drawing Γ is planar, we observe that D_x and D_y represent the x- and y-relations between every two vertices in any planar drawing of H. These relations are



(a) (b)

Figure 2.20: (a) N_x. (b) N_y. The corresponding planar st-digraphs D_x and D_y are represented in grey.

satisfied by the x- and y-coordinates obtained by computing the minimum cost flows on N_x and N_y, and thus, by Theorem 7, Γ is planar. In particular, we observe that:

- If there is a directed edge (u, v) in D_x (D_y), u will be right of (below) v in Γ, since the flow on the dual arc of (u, v) in N_x (N_y) is at least 1.
- The lengths of the edges of D_x and D_y are consistent. Recall that D_x and D_y are planar st-digraphs, and that the boundary of each face f of a planar st-digraph consists of two directed paths enclosing f, with common origin and destination. N_x and N_y are the dual planar st-digraphs of D_x and D_y; thus, for each face f of D_x (D_y), the incoming arcs of v_f in N_x (N_y) are duals of the edges of the "top" ("left") path of f, and the outgoing arcs of v_f in N_x (N_y) are duals of the edges of the "bottom" ("right") path of f. The consistency of the lengths of the edges of D_x and D_y then follows from the conservation property of the flow. And since there is a many-to-one correspondence between directed edge of H_x (H_y) and edges of D_x (D_y), also the lengths of the edges of Γ are consistent.

Hence, the minimum total edge length of a planar orthogonal drawing of H with optimal area $\mathcal{A} = w^* \cdot h^*$, is equal to the minimum cost of a flow with value w^* on N_x plus the minimum cost of a flow with value h^* on N_y. Each of the two minimum cost flows can be computed in $O(n^{7/4} \log n)$ time and $O(n)$ space as shown in [80]. □

We recall that the minimum number of bends for an orthogonal representation of a 4-planar graph with n vertices is $O(n)$ [20, 149]. The algorithm described in [145] produces such an orthogonal representation, and there exist various algorithms for producing an orthogonal representation with a sub-optimal, $O(n)$ number of bends (see, e.g, [17, 127, 148]).

2.7 Experiments

In this section, we present the results of an experimental study on a test suite of planar orthogonal representations of randomly generated biconnected 4-planar graphs. The analysis of the test suite has shown that the percentage of regular faces is quite high (95%). Motivated by this result, we have designed compaction heuristics based on the idea of "face regularization."

2.7.1 Compaction Heuristics

We have implemented a compaction algorithm for orthogonal representations based on the results described in the previous sections. Namely, let H be a given orthogonal representation of a 4-planar graph. The algorithm is as follows:

- H is first tested for turn-regularity, using the algorithm described in Theorem 6.
- If H is turn-regular, the algorithm computes an orthogonal drawing of H with optimal area and optimal total edge length within that area by applying the techniques in Theorem 9.
- If H contains some faces that are not turn-regular, an algorithm is applied to make these faces turn-regular. The algorithm adds dummy vertices and edges to H, creating a new orthogonal representation H' that is turn-regular. The techniques in Theorem 9 are then used to find a drawing Γ' of H' with optimal area and optimal total edge length within that area. Finally, the dummy vertices and edges are removed from Γ' to yield an orthogonal drawing Γ of H. In general the orthogonal drawing Γ does not have optimal area and total edge length.
- Two simple approaches are used to make non-regular faces turn-regular:
  1. The first approach is an improvement of the standard rectangularization method described in [54, 145]. When a dummy edge is inserted, a dummy vertex is added only if it really needed. Each non-regular face is divided into two or more smaller, rectangular faces.
  2. The second approach recursively adds a straight edge (randomly chosen to be either horizontal or vertical) between each pair of kitty corners, until the face has been decomposed in smaller (but not necessarily rectangular) turn-regular faces. In general, this technique adds a much smaller number of dummy edges than the first approach.

In the following we call the two heuristic compaction algorithms derived from the two regularization approaches described above `Heur1` and `Heur2`, respectively. Our implementations of these algorithms use the *GDToolkit* library[4] and will be included in its next release.

[4] `http://www.dia.uniroma3.it/~gdt`

2.7.2 Test Suite and Experimental Results

Heuristics `Heur1` and `Heur2` were tested on a set of 530 randomly generated biconnected 4-planar graphs with 10-3000 vertices. The results are compared to a third compaction heuristic, `StdComp`, in which all faces, both turn-regular and not, are decomposed into rectangles using the rectangularization method of `Heur1`.

The graphs in the test suite have been generated with a technique used in other experimental studies on orthogonal drawings [16]. Each biconnected 4-planar graph is generated from a cycle of three vertices by performing a random series of *InsertVertex* and *InsertEdge* operations. The *InsertVertex* operation subdivides an existing edge into two new edges separated by a new vertex. The *InsertEdge* operation inserts a new edge between two existing vertices on the same face. Any biconnected planar graph can be generated by a sequence of these two operations. Also, for each graph to be generated, the density of the graph (the number of edges divided by the number of vertices) is randomly chosen before the generation algorithm is run.

Our first experiment consisted in studying the percentage of non-regular faces in the graphs of our test suite. We have found that the percentage of non-regular faces decreases exponentially with the density of the graphs, i.e., with the ratio of the number of edges to the number of vertices (see Fig. 2.21).

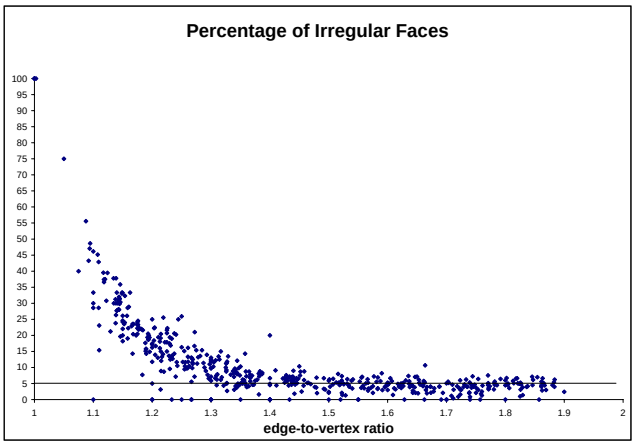


Figure 2.21: The percentage of non-regular faces with respect to the edge-to-vertex ratio.

We have then analyzed the results of the three heuristics. In particular, we have considered, for `Heur1` and `Heur2`, the improvement in the drawing area, total edge length, and maximum edge length with respect to `StdComp`. `Heur2` performs better than `Heur1` in most cases; also, the improvement in area and total edge length of `Heur2` with respect to `StdComp` increases with the number of vertices of the graph (see Fig. 2.22). The average improvement in area and total edge length is 19-25% for graphs with 3000 vertices, and there are some graphs in the test suite for which the improvement is more than 45%.

We have also run the three heuristics on a very large graph with $10,000$ vertices. The drawing

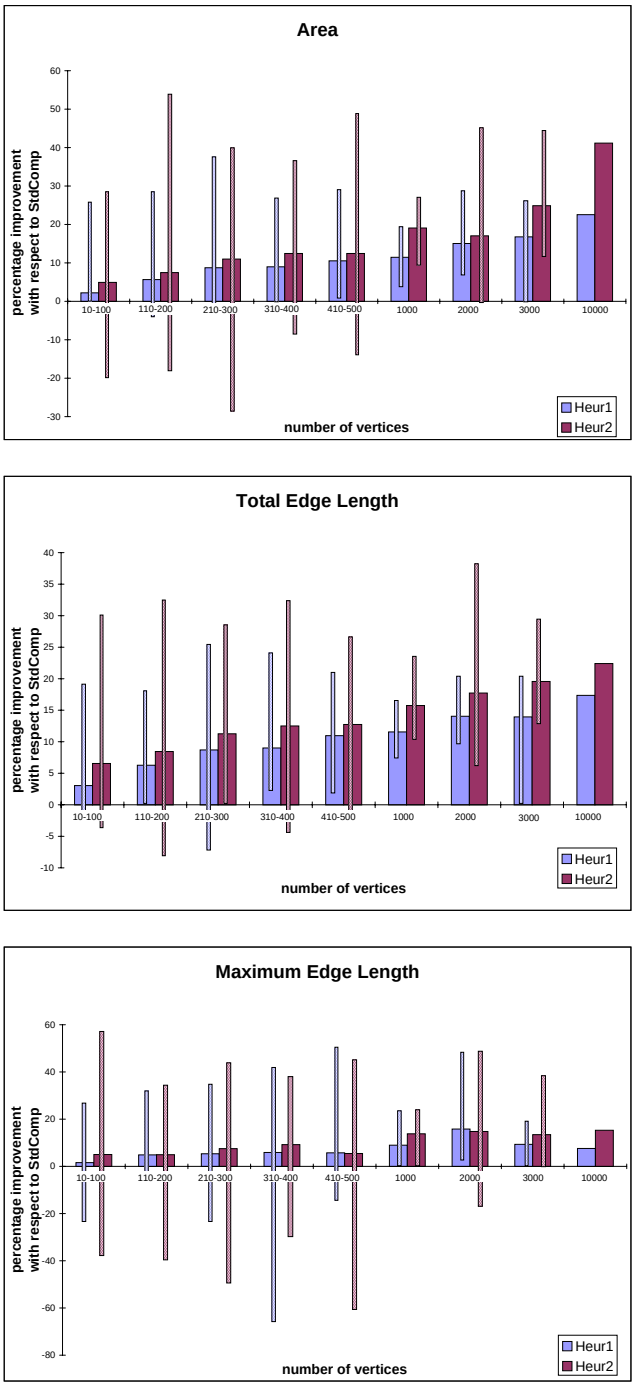


Figure 2.22: The average percentage improvement in area, total edge length, and maximum edge length of `Heur1` and `Heur2` with respect to `StdComp`. The narrow columns show the minimum and maximum improvement.

computed by `Heur2` improves the area by 41% and the total edge length by 22.4% with respect to the drawing computed by `StdComp`.

2.8 Conclusions and Open Problems

We introduced the notion of turn-regularity which allows the characterization of a class of orthogonal representations that are optimally compactable in terms of area in polynomial-time. In particular, given a turn-regular orthogonal representation, we provided a linear-time algorithm to compute a planar drawing with minimum area, and a polynomial-time algorithm to compute a planar drawing with optimal area and minimum total edge length within that area of the given orthogonal representation.

This work bears some similarity to an ongoing project at the Max-Planck-Institute für Informatik [103, 102] in which the problem of minimizing the total edge length in planar drawings of orthogonal representations is considered by solving an associated ILP problem. Some overlaps are unavoidable since that research effort addresses a question in the same class of problems as the one that we study in this paper. However, the work differs significantly because the respective proof techniques are different, our approach is driven by a combinatorial analysis that does not lead to a formulation of the problem as an ILP problem, and the problem we study combines both area minimization and total edge length minimization.

We provided several implementations of heuristics for making orthogonal representations turn-regular and we used them in the compaction algorithm, instead of the standard rectangularization step. Experiments on a randomly generated test suite of biconnected 4-planar graphs show that the new compaction strategy works much better than the standard one, especially for very large graphs.

Some of the open problems related to this work are the following:

- To find other families of orthogonal representations for which an optimal area drawing can be computed in polynomial time.
- To investigate other effective heuristics for making an orthogonal representation turn-regular by adding a small number of edges.
- The problem of computing an orthogonal representation with the minimum number of bends has been extensively investigated in a variable embedding setting [61, 62, 78]. It would be interesting to study the compaction problem when it is possible to change the embedding of the input graph.

Acknowledgments

We would like to thank Maurizio Patrignani for useful discussions.

Symbol	*Description*	*Section*
G	an embedded 4-planar graph	§2.2.1
H	an orthogonal representation of G	§2.2.1
Γ	a drawing of H	§2.3.1
Γ_r	an orientation of Γ with all vertical segments directed upward and all horizontal segments directed rightward	§2.3.3
Γ_ℓ	an orientation of Γ with all vertical segments directed upward and all horizontal segments directed leftward	§2.3.3
H_r	the orientation of H induced by Γ_r	§2.3.3
H_ℓ	the orientation of H induced by Γ_ℓ	§2.3.3
G_r	the orientation of G induced by H_r	§2.3.3
G_ℓ	the orientation of G induced by H_ℓ	§2.3.3
H_x	a partially-directed graph representing the "left" relation between maximal vertical chains of H	§2.4
H_y	a partially-directed graph representing the "below" relation between maximal horizontal chains of H	§2.4
D_x	a planar st-digraph obtained by shrinking the maximal vertical chains of H_x	§2.6
D_y	a planar st-digraph obtained by shrinking the maximal horizontal chains of H_y	§2.6
N_x	the dual planar st-digraph of D_x	§2.6
N_y	the dual planar st-digraph of D_y	§2.6

Table 2.2: The various symbols used to denote graphs, orthogonal representations, and drawings.

Chapter 3

Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms

3.1 Introduction

Graph drawing algorithms have traditionally been developed using a batch model, where the graph is redrawn from scratch every time a drawing is desired. These algorithms, however, are not well suited for interactive applications, where the user repeatedly makes modifications to the graph and requests a new drawing. When the graph is redrawn it is important to preserve the look (the user's "mental map" [115]) of the original drawing as much as possible, so the user does not need to spend a lot of time relearning the graph.

The problems of incremental graph drawing, where vertices are added one at a time, and the more general case of interactive graph drawing, where any combination of vertex/edge deletion and insertion is allowed at each step, have been starting to receive more attention. See, for example, the work of Biedl and Kaufmann [18], Brandes and Wagner [31], Bridgeman et. al. [43], Cohen et. al. [48], Fößmeier [73], Miriyala, Hornick, and Tamassia [114], Moen [116], North [120], Papakostas, Six, and Tollis [126], Papakostas and Tollis [125], and Ryall, Marks, and Shieber [136]. However, while the algorithms themselves have been motivated by the need to preserve the user's mental map, much of the evaluation of the algorithms has so far focused on traditional optimization criteria such as the area and the number of bends and crossings (see, for example, the analysis in Biedl and Kaufmann [18], Fößmeier [73], Papakostas, Six, and Tollis [126], and Papakostas and Tollis [125]). Mental map preservation is often achieved by attempting to minimize the change between drawings — typically by allowing only very limited modifications (if any) to the position of vertices and edge

Previously published as S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *Journal of Graph Algorithms and Applications*. 4(3):47–74, 2000.

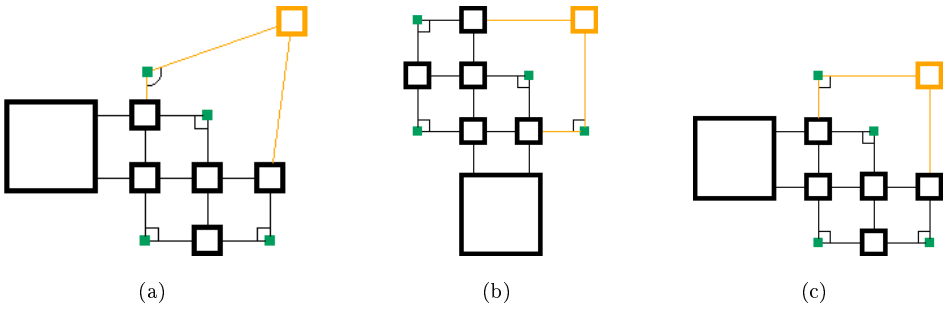


Figure 3.1: The rotation problem of InteractiveGiotto. (a) is the user-modified graph (the user's changes are shown in orange); (b) and (c) show the uncorrected and corrected outputs, respectively. While (b) and (c) are both clearly drawings of the graph shown in (a), the resemblance is more readily seen in the properly rotated and reflected drawing (c).

bends in the existing drawing — making it important to be able to measure precisely how much the look of the drawing changes. Animation can be used to provide a smooth transition between the drawings and can help compensate for greater changes in the drawing, though it is still important to limit, if not minimize, the difference between the drawings because if there is a very large change it can become difficult to generate a clear, useful animation. It is thus still important to have a measure of how the look of the drawing changes.

Studying "difference" metrics to measure how much a drawing algorithm changes the user's mental map has a number of benefits, including

- providing a basis for studying the behavior of constraint-based interactive drawing algorithms like InteractiveGiotto [43], where meaningful bounds on the movement of any given part of the drawing are difficult to obtain,
- providing a technique to compare the results of different interactive drawing algorithms, and
- providing a goal for the design of new drawing algorithms by identifying which qualities of the drawing are the most important to preserve.

Finding a good difference metric also has an immediate practical benefit, namely solving the "rotation problem" of InteractiveGiotto. Giotto [146], the core of InteractiveGiotto, does not take into account the coordinates of vertices and bends in the original drawing when constructing a new drawing — and, as a result, the output of InteractiveGiotto may be rotated by a multiple of 90 degrees and/or be a mirror-image reflection of the original drawing (Figure 3.1). The problem can be solved by computing the value of the metric for each of the possible rotations and choosing the rotation with the smallest value.

Eades et. al. [63], Lyons, Meijer, and Rappaport [108], and Misue et. al. [115] have proposed several models for formalizing the notion of the mental map, though more work needs to be done to formally define potential difference metrics and then experimentally validate (or invalidate) them.

Validation can be via user studies similar to those done by Purchase, Cohen, and James [131, 132] to evaluate the impact of various graph drawing aesthetics on human understanding.

Motivated by applications to InteractiveGiotto, this paper will focus primarily on difference metrics for orthogonal drawings, though many of the metrics can be used without modification for arbitrary drawings. Section 3.3 describes several potential metrics, Section 3.4 presents a framework for evaluating the suitability of the metrics along with a preliminary evaluation, and Section 3.5 outlines plans for future work.

3.2 Preliminaries

Paired Sets of Objects Every metric presented in this paper compares two drawings D and D' of the same graph G. Each object of G has a representation in both D and D'. For example, each vertex of G has a representation in both drawings — if vertices are drawn as rectangles, this representation consists of the position, size, color, line style, etc. of the rectangle. Similarly, each edge of G has a representation in both drawings, and if edges are drawn as polylines, this representation consists of the positions of the bends and endpoints, plus the color, line style, and so forth.

A *paired set* of objects is a set of pairs describing the representation of each object in the two drawings. The paired set of vertices of G is the set of pairs (r_v,r'_v), where r_v and r'_v are the representations of v in D and D', respectively, for all vertices v of G. The paired set of edges is defined similarly. Referring to a paired set is simply a way of matching up the elements of each drawing according to the underlying object of G that they represent.

It should be noted that the only features of the representations that are considered here are the geometric features such as position and size; other features like vertex color and line style are also very important in preserving the look of the drawing and may be able to at least partially compensate for geometric changes but are not considered further at this stage.

Point Set Selection Most of the metrics are based on point sets, working with paired sets of points derived from the edges and vertices of the graph rather than the edges and vertices themselves. Once derived, each point is independent from the others — there is no notion of a group of points being related because they were derived from the same vertex, for example.

Points can be selected in a number of ways. North [120] suggests that vertex positions are a significant visual feature of the drawing, and two vertex-centered methods — *centers* and *corners* — are used here to reflect that idea. "Centers" consists of the center points of each vertex; this captures how vertices move. "Corners" uses the four corners of each vertex, taking into account both vertex motion (the movement of the center) and changes in the vertex dimension. It seems important to take into account changes in vertex dimension because a vertex with a large or distinctive shape can act as a landmark to orient the user to the drawing; loss of that landmark makes orientation more difficult. Other choices of points can include edge bends and endpoints.

Points can also be derived from groups of graph objects. For example, the vertices of the graph

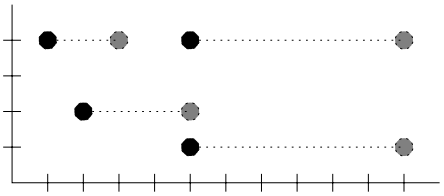


Figure 3.2: Two point sets (black and gray) superimposed. (Corresponding points in the two sets are connected with dotted lines.) As shown, the Euclidean distance metric (Section 3.3.1) would report a distance of 4.25. However, translating the gray points one unit to the left and then scaling by 1/2 in the x direction allows the point sets to be matched exactly, for a distance of 0. It should be noted that exact matches are not possible in general.

can be partitioned, and points derived from the centroids or convex hull of the partitions. Point sets based on partitioning can be used to capture information about larger units of the graph, such as groups of vertices representing related objects.

Drawing Alignment The key features of a graph object's representation are coordinates, which means metrics may be sensitive to the particular values of those coordinates — the scaling and translation of one point set relative to the other can make a large difference in the value of the metric (Figure 3.2).

To eliminate this effect, the drawings are *aligned* before coordinate-sensitive metrics are computed. This is done by extracting a (paired) set of points from the drawings and applying a point set matching algorithm to obtain the best fit. In general the matching algorithm should take into account scaling, translation, and rotation, though it may be possible to eliminate one or more of the transformations for certain metrics or if something is known about the relationship between the two drawings. For example, interactive drawing algorithms often preserve the rotation of the drawing (see Biedl and Kaufmann [18] and Papakostas and Tollis [125] for examples), eliminating the need to consider rotation in the alignment stage. Even if the algorithm does not preserve the rotation, for orthogonal drawings there are only eight possible rotations for the second drawing relative to the first — four multiples of $\pi/2$, applied to the original drawing and its reflection about the x axis — which can be handled by computing the metric separately for each rotation and taking the minimum value instead of incorporating rotation into the alignment process.

A great deal of work has been done on point set matchings; see Alt, Aichholzer, and Rote [1], Chew et. al. [47], and Goodrich, Mitchell, and Orletsky [83] for several methods of obtaining both optimal and approximate matchings. Different methods can be applied when the correspondence between points is known as it is here; Imai, Sumino, and Imai [97] provide an algorithm that minimizes the maximum distance between corresponding points under translation, rotation, and scaling. In the implementation used in Section 3.4, the alignment is performed by using gradient search to minimize the distance squared between points.

3.3 Metrics

The difference metrics being considered fall into six categories:

- **distance**: metrics based on the distance between points or the distance points move between drawings
- **proximity**: metrics based on the nearness of points and the clustering of points according to the distance between them
- **partitioning**: metrics based on partitioning points according to measures other than proximity
- **orthogonal ordering**: metrics based on the relative angle between pairs of points
- **shape**: metrics based on the sequence of horizontal and vertical segments of the graph's edges
- **topology**: metrics based on the embedding of the graph

Proximity, ordering, and topology are suggested by Eades et. al. [63], Lyons, Meijer, and Rappaport [108], and Misue et. al. [115] as qualities which should be preserved; distance (also suggested by Lyons, Meijer, and Rappaport [108]), shape, and partitioning reflect intuition about what causes drawings to look different. Within each category specific metrics were chosen to capture intuition about what qualities of the drawing are important to preserve.

An alternative taxonomy is given by Biedl et. al. [19]. This taxonomy is similar to the one given above, with the main distinctions being the inclusion of "feature similarity" based on the appearance of regions of the drawing, and the grouping of all measures based on the comparison of point sets into a single "metric similarity" category.

In the following, let P denote the (paired) set of points, and p_i and p'_i be the coordinates of point i in drawings D and D', respectively. Also let $d(p, q)$ be the Euclidean distance between points p and q.

3.3.1 Distance

The distance metrics reflect the simple observation that drawings that look very different cannot be aligned very well, and vice versa. Since the alignment is based on distance minimization, these metrics essentially measure the quality of the alignment.

In order to make the value of the distance metrics comparable between pairs of drawings, they are scaled by the graph's *unit length* u. For orthogonal drawings the unit length can be computed by taking the greatest common divisor of the Manhattan distances between vertex centers and bend points on edges. Non-orthogonal portions of the drawing, such as modifications of an orthogonal drawing made by the user, can be ignored during the computation. While the determination of the unit length will be unreliable if only a small portion of the drawing is orthogonal, scaling by the unit

length is not necessary in some applications (e.g., solving the rotation problem of InteractiveGiotto) and can often be supplied manually if it is required (e.g., the drawing algorithm is known to place vertices on a unit grid).

Hausdorff Distance The *undirected Hausdorff distance* is a standard metric for determining the distance between two point sets, and measures the largest distance between a point in one set and its nearest neighbor in the other.

$$\text{haus}(D, D') = \frac{1}{u} \max\{\max_i \min_j d(p_i, p'_j), \max_i \min_j d(p'_i, p_j)\}$$

where $1 \leq i, j \leq |P|$ and $j \neq i$.

Euclidean Distance *Euclidean distance*, used by Lyons, Meijer, and Rappaport [108], is a simple metric measuring the average distance moved by each point from the first drawing to the second; it is motivated by the notion that if points move a long way from their locations in the first drawing, the second drawing will look very different.

$$\text{dist}(D, D') = \frac{1}{u\,|P|} \sum_{1 \leq i \leq |P|} d(p_i, p'_i)$$

Relative Distance *Relative distance* measures the average change in the distance between each pair of points between the first drawing and the second. This measures how much the points in each drawing move relative to each other; it is similar in some respects to the orthogonal ordering metrics in Section 3.3.4.

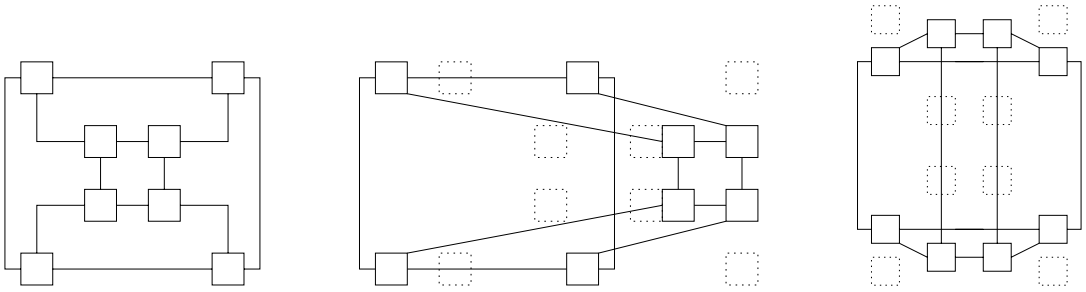
$$\text{rdist}(D, D') = \frac{1}{|P|\,(|P|-1)} \sum_{1 \leq i,j \leq |P|} |\, d(p_i, p_j) - d(p'_i, p'_j)\,|$$

3.3.2 Proximity

The proximity metrics reflect the idea that points near each other in the first drawing should remain near each other in the second drawing. This is stronger than the distance metrics because it captures the idea that if an entire subgraph moves relative to another (and there are only small changes within each subgraph), the distance should be less than if each point in one of the subgraphs moves in a different direction (Figure 3.3).

Three different metrics are used to try to capture this idea: nearest neighbor within (nn-within), nearest neighbor between (nn-between), and ϵ-clustering.

Nearest Neighbor Within *Nearest neighbor within* is based on the reasoning that if p_j is the closest point to p_i in D, then p'_j should be closest point to p'_i in D'. Considering only distances within a single drawing means that nn-within is alignment-independent and thus not subject alignment errors, but means that it is not suitable for solving the rotation problem of InteractiveGiotto.



(a) (b) (c)

Figure 3.3: Proximity: (b) looks more like (a) than (c) does because the relative shape of both the inner and outer squares are preserved even though the distance (using the Euclidean distance metric) between (c) and (a) is smaller. An aligned version of the vertices of (a), used in the computation of the distance metric, is shown with dotted lines in (b) and (c).

This metric has two versions, *weighted* and *unweighted*. In the weighted version the number of points closer to p'_i than p'_j is considered, whereas in the unweighted version only whether or not p'_j is the closest point matters. The reasoning behind the weighted version is that if there are more points between p'_i and p'_j, the visual linkage between p'_i and p'_j has been disrupted to a greater degree and the drawing looks more different.

In both cases the distance is scaled by the number of points being considered and W, the maximum weight contributed by a single point, so that the metric's value is always in the range $[0, 1]$.

$$\text{nn-w}(D, D') = \frac{1}{W\,|P|} \sum_{1 \le i \le |P|} \text{closer}(p'_i, p'_j)$$

where p_j is the closest point to p_i in D and

$$\text{closer}(p'_i, p'_j) = \left|\{k \mid d(p'_i, p'_k) < d(p'_i, p'_j)\}\right|, \qquad W = |P| - 2 \qquad \text{(weighted)}$$

$$\text{closer}(p'_i, p'_j) = \begin{cases} 0 & \text{if } d(p'_i, p'_j) \le d(p'_i, p'_k), k \ne i \\ 1 & \text{otherwise} \end{cases}, \quad W = 1 \qquad \text{(unweighted)}$$

Nearest Neighbor Between *Nearest neighbor between* is similar to nn-within but instead measures whether or not p'_i is the closest of the points in D' to p_i when the two drawings are aligned. The idea that a point should remain nearer to its original position than any other is also the force behind layout adjustment algorithms based on the Voronoi diagram [108].

$$\text{nn-b}(D, D') = \frac{1}{W\,|P|} \sum_{1 \le i \le |P|} \text{closer}(p_i, p'_i)$$

where

$$\text{closer}(p_i, p_i') = \left|\{j \mid d(p_i, p_j') < d(p_i, p_i')\}\right|, \qquad W = |P| - 1 \qquad \text{(weighted)}$$

$$\text{closer}(p_i, p_i') = \begin{cases} 0 & \text{if } d(p_i, p_i') \leq d(p_i, p_j'), j \neq i \\ 1 & \text{otherwise} \end{cases}, \quad W = 1 \qquad \text{(unweighted)}$$

Unlike nn-within, nn-between is not alignment- and rotation-independent and thus is suitable for solving the rotation problem.

ϵ-Clustering The definition for an ϵ-cluster is from Eades et. al [63]: An ϵ-cluster for a point p_i is the set of points p_j such that $d(p_i, p_j) \leq \epsilon$, where a reasonable value to use for ϵ is

$$\epsilon = \max_i \min_{j \neq i} d(p_i, p_j)$$

The *ϵ-cluster* metric measures how the ϵ-cluster for p_i compares to that for p_i'. Let $C_D = \{(i,j) \mid d(p_i, p_j) \leq \epsilon_D\}$ and $C_{D'} = \{(i,j) \mid d(p_i', p_j') \leq \epsilon_{D'}\}$. Then

$$\text{clust}(D, D') = 1 - \frac{|C_D \cap C_{D'}|}{|C_D \cup C_{D'}|}$$

The idea is that points should be in the same ϵ-cluster in both drawings.

3.3.3 Partitioning

The partitioning metrics are based on dividing the point set into subsets according to some criteria, and measuring qualities of these partitions. The motivation for this is to capture "visual units" that the user may use for orientation when learning the new drawing.

Fixed Relative Position Partitioning A variety of partitioning methods are possible. A simple one is to divide the point set so that the points in each group have the same relative position in both drawings. This identifies blocks of the graph that are the same in both drawings — the larger the partitions, the more unchanged parts and the more similar the drawings.

Two metrics are computed, the average partition size and the number of partitions. Both are scaled to have values between 0 and 1:

$$\text{alsz}(D, D') = 1 - \frac{\left(\frac{1}{k} \sum_{1 \leq i \leq k} |S_i|\right) - 1}{|P| - 1} \qquad \text{[average size]}$$

$$\text{alct}(D, D') = \frac{k - 1}{|P| - 1} \qquad \text{[number of partitions]}$$

where the set of partitions is $\{S_1, \ldots, S_k\}$. The idea is that as the drawings become more different, the partition size will decrease and the number of partitions will increase.

The partitioning method and the metrics computed are obviously quite simple, and can be made much more sophisticated. For example, the partitions can be adjusted to only include points that are also close physically; while it tends not to be the case that points from widely separated regions of the drawing are in the same partition, it can occur. A large distance between points interferes with their grouping as a single visual unit.

Additional sophistications can address things such as how visually separate two adjacent partitions are, since if they are very near or entertwined in one drawing, it is more difficult for the observer to distinguish them and use them as landmarks for the other drawing. Partitions can also be weighted to take into account how well the partition reflects a distinct unit of the graph, so that partitions containing only points derived from a connected subgraph are better than those containing points from several unconnected portions of the graph, even if those subgraphs are physically close together.

3.3.4 Orthogonal Ordering

The *orthogonal ordering* metric reflects the desire to preserve the relative ordering of every pair of points — if p_i is northeast of p_j in D, p'_i should remain to the northeast of p'_j in D' (Eades et. al. [63] and Misue et. al. [115]). The simplest measurement of difference in the orthogonal ordering is to take the angle between the vectors $p_j - p_i$ and $p'_j - p'_i$ (*constant-weighted* orthogonal ordering). This has the desirable feature that if p_j is far from p_i, $d(p_j, p'_j)$ must be larger to result in the same angular move, which reflects the intuition that the relative position of points near each other is more important that the relative position of points that are far apart.

However, simply using the angular change fails to take into account situations such as that shown in Figure 3.4. This problem can be addressed by introducing a weight that depends on the particular angles involved in the move in addition to size of the move (*linear-weighted* orthogonal ordering).

$$\text{order}(D, D') = \frac{1}{W\,|P|} \sum_{1 \le i,j \le |P|} n \min(\text{order}(\theta_{ij}, \theta'_{ij}), \text{order}(\theta'_{ij}, \theta_{ij}))$$

where θ_{ij} is the angle from the positive x axis to the vector $p_j - p_i$, θ'_{ij} is the angle from the positive x axis to the vector $p'_j - p'_i$, and

$$\text{order}(\theta_{ij}, \theta'_{ij}) = \int_{\theta_{ij}}^{\theta'_{ij}} \text{weight}(\theta)\ d\theta, \qquad W = \min\left\{ \int_0^{\pi} \text{weight}(\theta)\ d\theta, \int_{\pi}^{2\pi} \text{weight}(\theta)\ d\theta \right\}$$

The weight functions are

$$\text{weight}(\theta) = \begin{cases} \frac{\frac{\pi}{2} - (\theta \bmod \frac{\pi}{2})}{\frac{\pi}{4}} & \text{if } (\theta \bmod \frac{\pi}{2}) > \frac{\pi}{4} \\ \frac{\theta \bmod \frac{\pi}{2}}{\frac{\pi}{4}} & \text{if } (\theta \bmod \frac{\pi}{2}) \le \frac{\pi}{4} \end{cases} \qquad \text{(linear-weighted)}$$

$$\text{weight}(\theta) = 1 \qquad \text{(constant-weighted)}$$

The λ-matrix model for measuring the difference of two point sets, used by Lyons, Meijer, and Rappaport [108], is based on the concept of order type of a point set, from Goodman and Pollack [82].

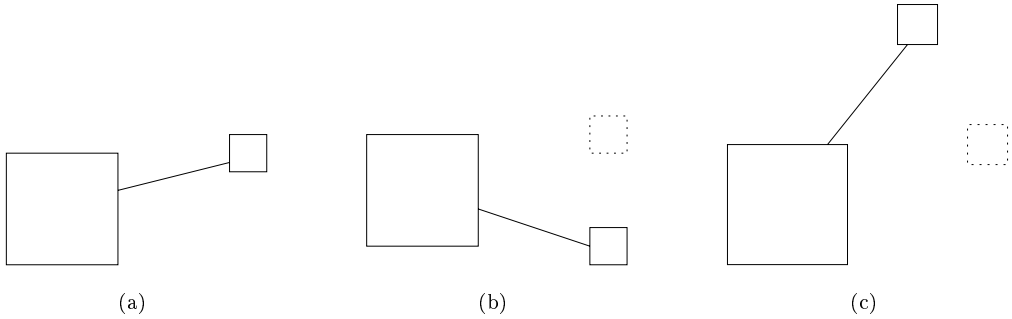


Figure 3.4: Orthogonal ordering: Even though the angle the vertex moves relative to the center of the large vertex is the same from (a) to (b) and from (a) to (c), the perceptual difference between (a) and (c) is much greater. The original location of the vertex is shown with a dotted box in (b) and (c) for comparison purposes.

This model tries to capture the notion of the relative position of vertices in a straight-line drawing and is thus related to the orthogonal ordering metric.

3.3.5 Shape

The *shape* metric is motivated by the reasoning that edge routing may have an effect on the overall look of the graph (Figure 3.5). The shape of an edge is the sequence of directions (north, south, east, and west) traveled when traversing the edge; writing the shape as a string of N, S, E, and W characters yields the *shape string* of the edge. For non-orthogonal edges the direction is taken to be the most prominent direction; for example, if the edge goes from (1,1) to (4,2) the most prominent direction is east. For each pair of edges (e_i, e'_i), the edit distance between the corresponding shape strings is computed. Two methods are used for determining the edit distance. One uses dynamic programming to compute the minimum number of insertions, deletions, or replacements of characters needed to transform one string into the other. The other is similar, but normalizes the measure according to the length of the strings; the algorithm is given by Marzal and Vidal [109]. The value of the shape metric is the average edit distance over the graph's edges.

$$\text{shape}(D, D') = \frac{1}{|E|} \sum_{1 \leq i \leq |E|} \text{edits}(e_i, e'_i)$$

Shape is scale- and translation-independent.

The shape metric is similar in spirit to the cost function used by Brandes and Wagner [28] in their dynamic extension of Giotto [146]. Their cost function counts the number of changes in angles at vertices and edge bends; the shape metric takes this in account to some degree by noting changes in the direction of an edge segment.

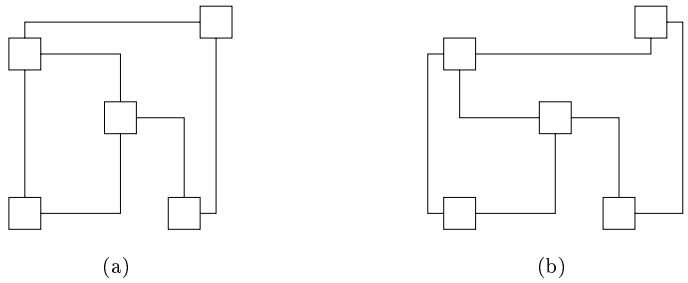


Figure 3.5: Shape: (a) and (b) look different even though the graphs are the same and the vertices have the same coordinates.

3.3.6 Topology

The topology metric reflects the idea that preserving the order of edges around a vertex is important in preserving the mental map (Eades et. al. [63] and Misue et. al. [115]) — comparing the drawing produced by Giotto in Figures 3.6 and 3.7 to the user's input illustrates this. However, since most interactive orthogonal drawing algorithms always preserve topology, it is not useful as a means of comparing these algorithms. (See, for example, the algorithms of Bridgeman et. al. [43], Biedl and Kaufmann [18], Fößmeier [73], Papakostas, Six, and Tollis [126], and Papakostas and Tollis [125].) Topology is also alignment-independent and so can not be used to solve the rotation problem of InteractiveGiotto. As a result, it is not discussed in any more detail here.

3.4 Analyzing the Metrics

Once defined, the suitability of the metrics must be evaluated. A good metric for measuring the difference between drawings should satisfy the following three requirements:

- it should *qualitatively* reflect the visual difference between two drawings, i.e. the value increases as the drawings diverge;
- it should *quantitatively* reflect the visual difference so that the magnitude of the difference in the metric is proportional to the perceived difference; and
- in the rotation problem of InteractiveGiotto, the metric should have the smallest value for the correct rotation, though this requirement can be relaxed when the difference between drawings is high since in that case there is no clear "correct" rotation.

The third point is the easiest to satisfy — in fact, most of the metrics defined in the previous section can be used to solve the rotation problem — but is still important worth considering since the problem was one of the factors that first inspired this work.

The running time of the metrics is not considered at this point. While efficiency is clearly a concern, the goal at this stage is to identify the type of measure that best captures visual similarity. Once this has been done, efficient implementations and/or approximations can be considered.

Some preliminary work has been done on evaluating the proposed metrics with respect to the first and third criteria. Evaluating the qualitative behavior of potential metrics requires a human-generated master ordering of pairs of drawings based on the visual difference between the existing drawing and the new drawing in each pair. This is very difficult to do when each pair of drawings is of a different graph, and most interactive drawing algorithms only produce a single drawing of a particular user-modified graph. InteractiveGiotto, however, makes it possible to obtain a series of drawings of the same input by relaxing the constraints preserving the layout. By default InteractiveGiotto preserves edge crossings, the direction (left or right) and number of bends on an edge, and the angles between consecutive edges leaving a vertex. Recent modifications allow the user to turn off the last two constraints on an edge-by-edge or vertex-by-vertex basis, making it possible to produce a series of drawings of the same graph by relaxing different sets of constraints. A smooth way of relaxing the constraints is to use a breadth-first ordering, expanding outward from the user's modifications. In the first step all of the constraints are applied, in the second step the bend and angle constraints are relaxed for all of the modified objects, in the third step the angle constraints are relaxed for all vertices adjacent to edges whose bends constraints have been relaxed, in the fourth step the bend constraints are relaxed for all edges adjacent to vertices whose angle constraints have been relaxed, and so on, alternating between angle and bend constraints until all of the constraints have been relaxed. This relaxation method is based on the idea that the user is most willing to allow restructuring of the graph near where her changes were made and so the drawings produced resemble drawings that an actual user might encounter. The result is a series of drawings of the same graph — a *relaxation sequence* — bearing varying degrees of similarity to the original.

3.4.1 An Example

Figures 3.6 and 3.7 show portions of two relaxation sequences produced by InteractiveGiotto; the base graphs and user modifications are those used in the first two steps of Figure 2 in Bridgeman et. al. [43]. Giotto's redraw-from-scratch drawing of the graph is also included for comparison. Figures 3.8 and 3.9 show the results of several metric-and-pointset combinations for each sequence of drawings. Since Giotto and InteractiveGiotto do not preserve the orientation of the original drawing, each metric is computed for the eight possible rotations (four multiples of $\pi/2$, with or without a reflection around the x-axis) and the lowest value chosen. The color of each column indicates the *confidence*, a measure of how much lower the metric's value is for the best rotation as compared to the second best; red is the most confident and purple is the least confident, with white indicating that the metric is rotation-independent (and so confidence is meaningless) and black indicating that two rotations had the same lowest value. The shading of the column indicates whether or not the metric chose the right rotation — hashing means that the wrong rotation had the lowest value. The correct rotation is defined to be the rotation chosen by the metric/pointset combination with the highest confidence;

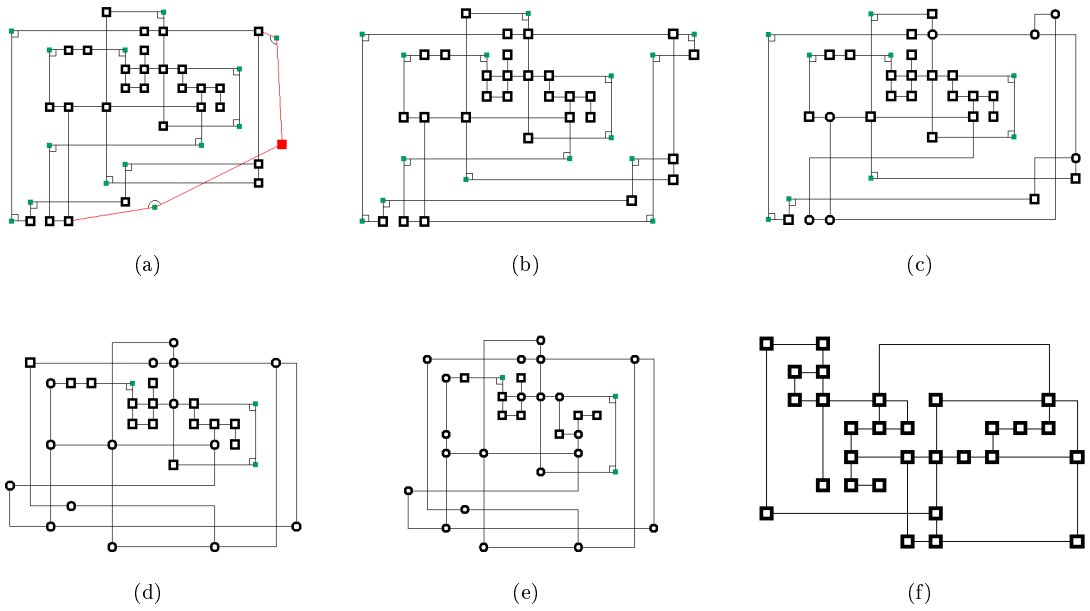


Figure 3.6: Relaxations for stage 1. (a) shows the user's modifications; (b)–(e) show the output from InteractiveGiotto for relaxation steps 0, 4, 6, and 7; (f) is the output from Giotto. (Intermediate relaxation steps produced drawings identical to those shown and have been left out.) Rounded vertices and bends without markers indicate vertices and bends for which the constraints have been relaxed.

in practive this is nearly always the rotation a human would pick as the correct answer.

The point sets shown in Figures 3.8 and 3.9 show use the following abbreviations:

- **centers**: vertex centers
- **corners**: vertex corners
- **hulls (cen)**: the vertex centers point set is partitioned using fixed relative partitioning; the points used are the points on the convex hull of each partition
- **hulls (cor)**: the vertex corners point set is partitioned using fixed relative partitioning; the points used are the points on the convex hull of each partition
- **centroids (cen)**: the vertex centers point set is partitioned using fixed relative partitioning; the points used are the centroids of each partition
- **centroids (cor)**: the vertex corners point set is partitioned using fixed relative partitioning; the points used are the centroids of each partition

The t or f in brackets following the point set name indicates whether or not points derived from parts of the graph modified by the user are included in the point set. The first position indicates

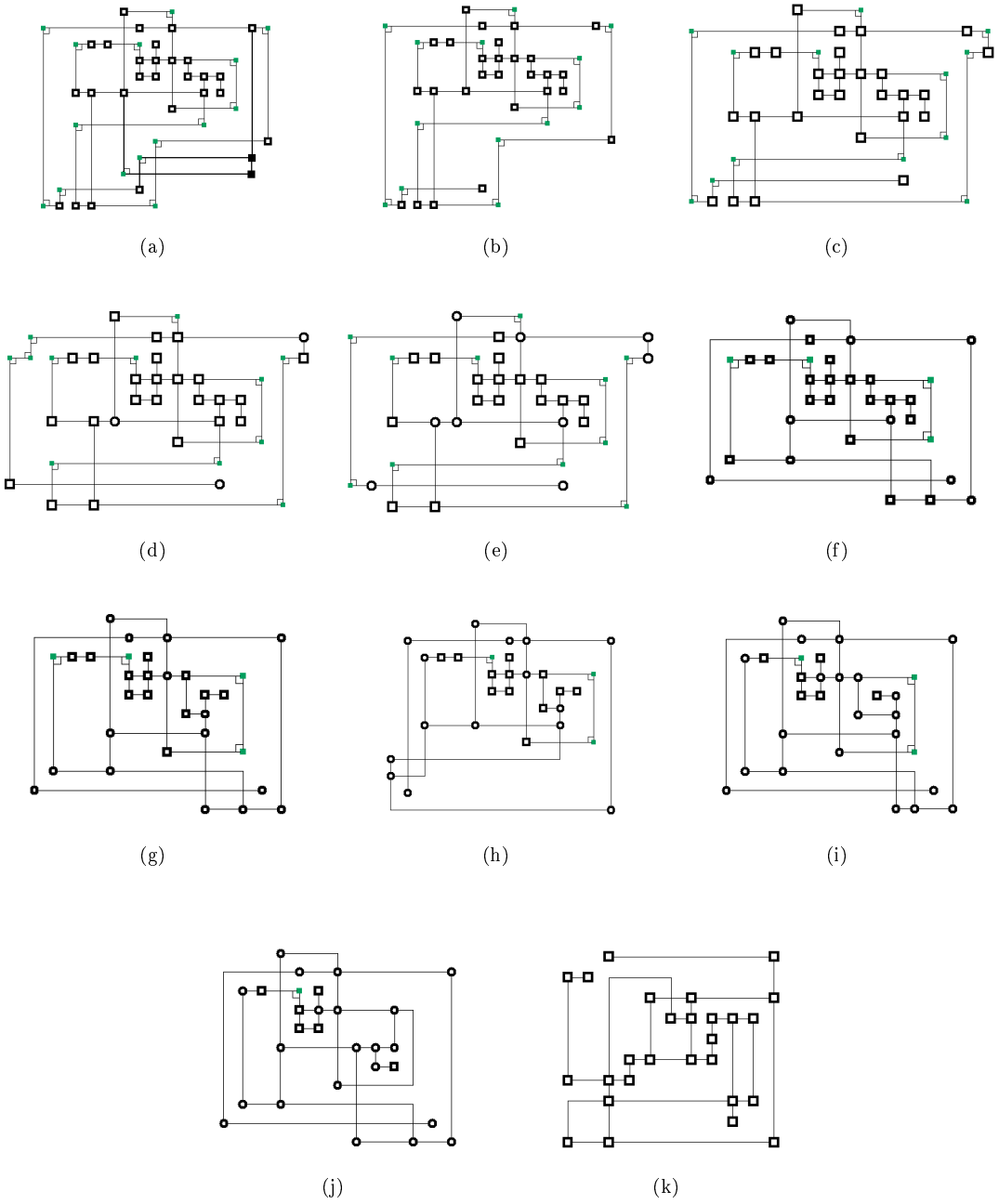


Figure 3.7: Relaxations for stage 2. (a) shows the starting graph, (b) shows the user's modifications (two vertices and their adjacent edges deleted), (c) is the fully-constrained output from InteractiveGiotto (step 0), (d)–(j) show the output from InteractiveGiotto for relaxation steps 2–8 (step 1 produced the same drawing as step 0), and (k) is the output from Giotto. Rounded vertices and bends without markers indicate vertices and bends for which the constraints have been relaxed.

the value for the point set used for the computation of the metric, the second the point set used for drawing alignment.

The relaxed drawings are labelled as follows:

- n**i**: InteractiveGiotto's drawing of the nth relaxation step
- **xg**: Giotto's output

3.4.2 Experimental Setup

For the experimental study, a set of 14 graphs with 30 vertices each were extracted from the 11,582-graph test suite used in the experimental studies of Di Battista et. al. [56, 57]. 30-vertex graphs were chosen as being small enough to work with easily, but large enough so that one modification does not affect the entire graph. Future experiments will consider graphs of different sizes.

A random modification was applied to each graph to simulate a user's modification. A modification is one of:

- *edge insert*: insertion of a single edge between two randomly chosen vertices
- *edge delete*: deletion of a single edge
- *edge split*: insertion of a single vertex at the midpoint of an existing edge, splitting the edge into two new edges
- *vertex insert*: insertion of a vertex along with a number of adjacent edges; both the number of edges and their endpoints is chosen randomly
- *vertex delete*: deletion of a vertex and its incident edges

Operations were not allowed to disconnect the graph. Since InteractiveGiotto preserves the embedding of the graph and the edge crossings and bends, new edges were routed so as to mimic how an actual user might draw the edge, instead of simply connecting the endpoints with a straight line (and potentially introducing many edge crossings). Only a single modification was made in each graph because as the user's changes affect a larger portion of the graph, the difference between the new drawings and the original rapidly becomes high and it is difficult to determine an ordering of the relaxation sequence.

Modifications of each type were applied to each of the 14 original graphs, though due to some difficulties with Giotto, the total number of modified graphs generated was only 63.

For each modified graph, a series of progressively relaxed drawings was obtained by incrementally relaxing the constraints given to InteractiveGiotto. The number of relaxed drawings for each modified graph ranged from 8 to 17, but was generally around 11. Each sequence of drawings was then ordered by a human according to increasing visual distance from the original drawing and this ordering was compared to the orderings produced by sorting the drawings in each relaxation sequence according to the computed values of the metric. The orderings were compared using the

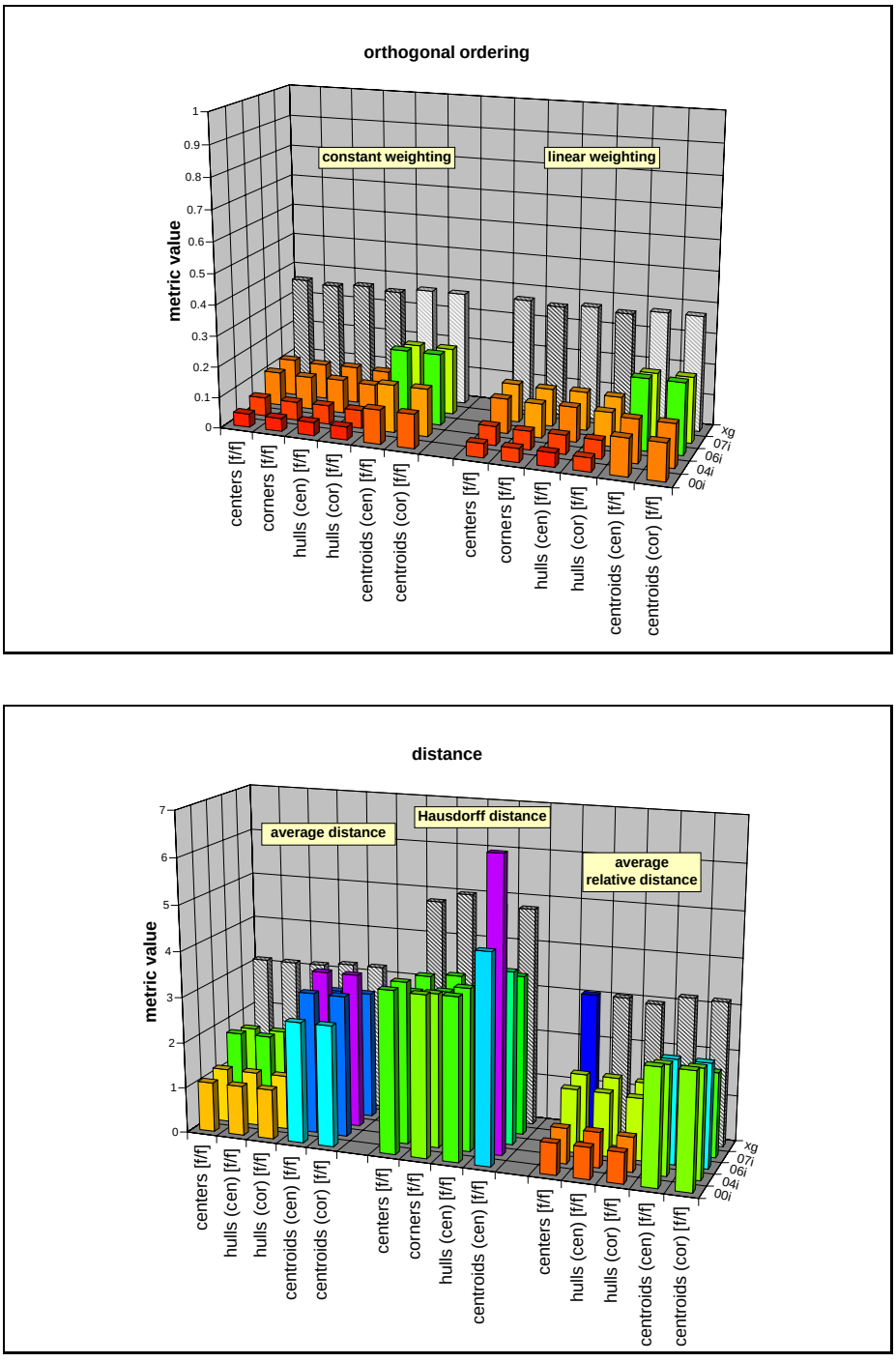


Figure 3.8: Selected metric values for each drawing in stage 1.

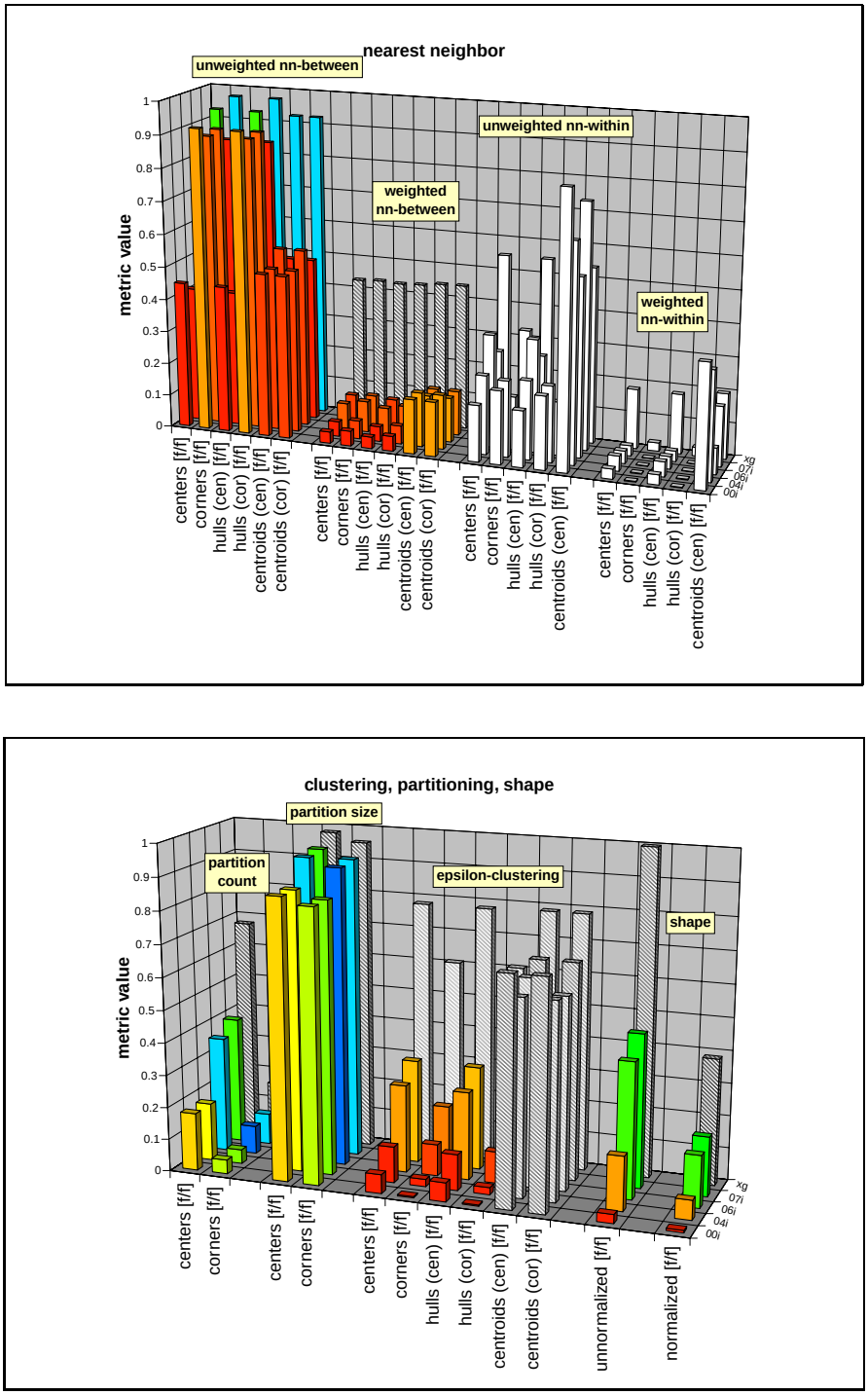


Figure 3.8: Selected metric values for each drawing in stage 1. (continued)

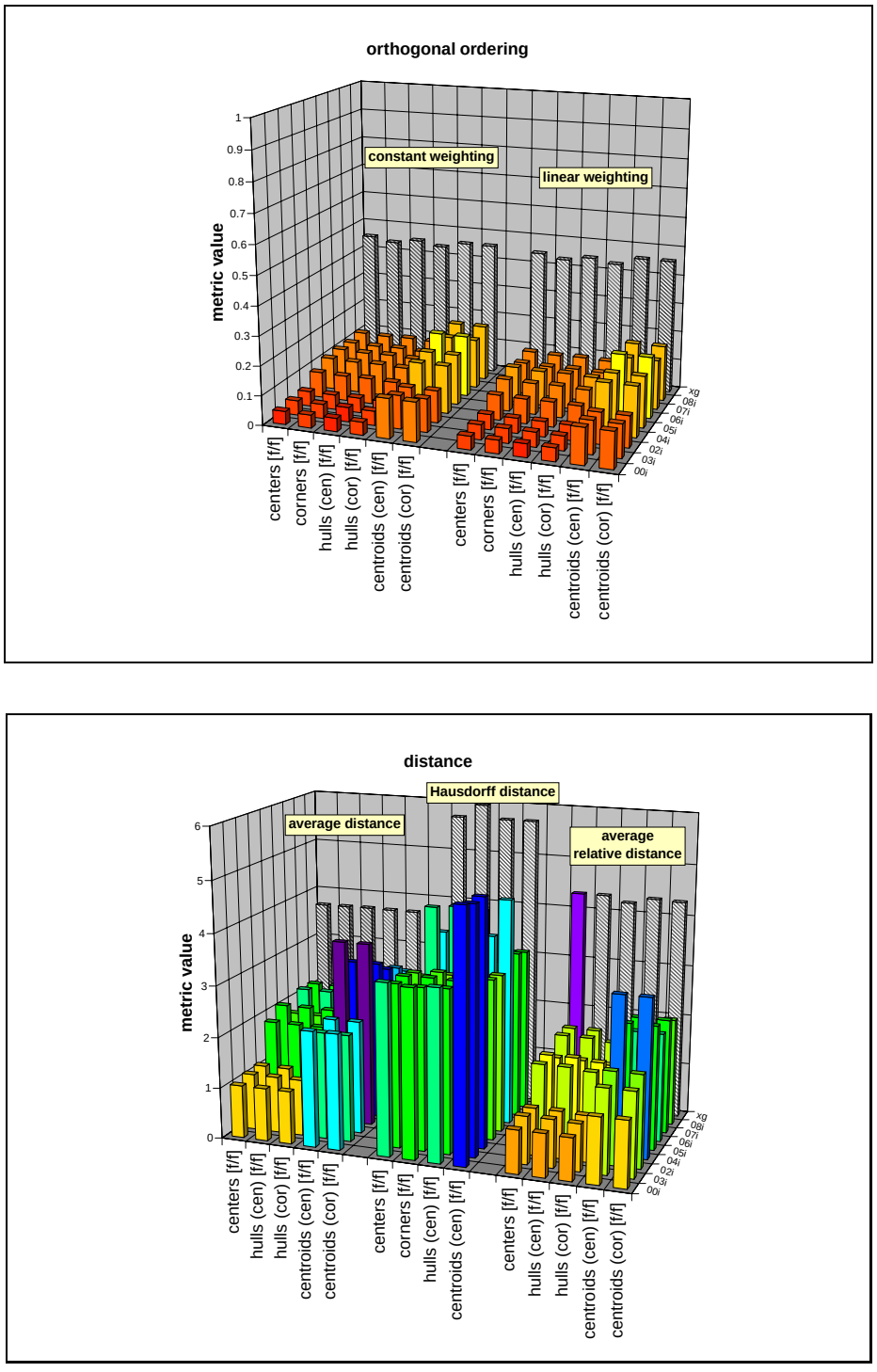
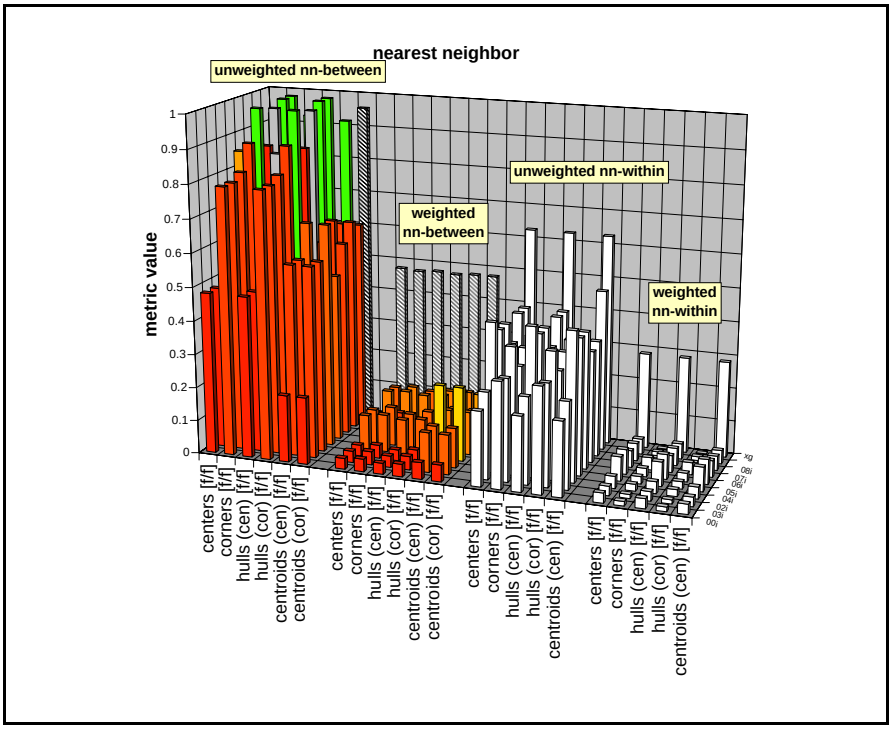


Figure 3.9: Selected metric values for each drawing in stage 2.



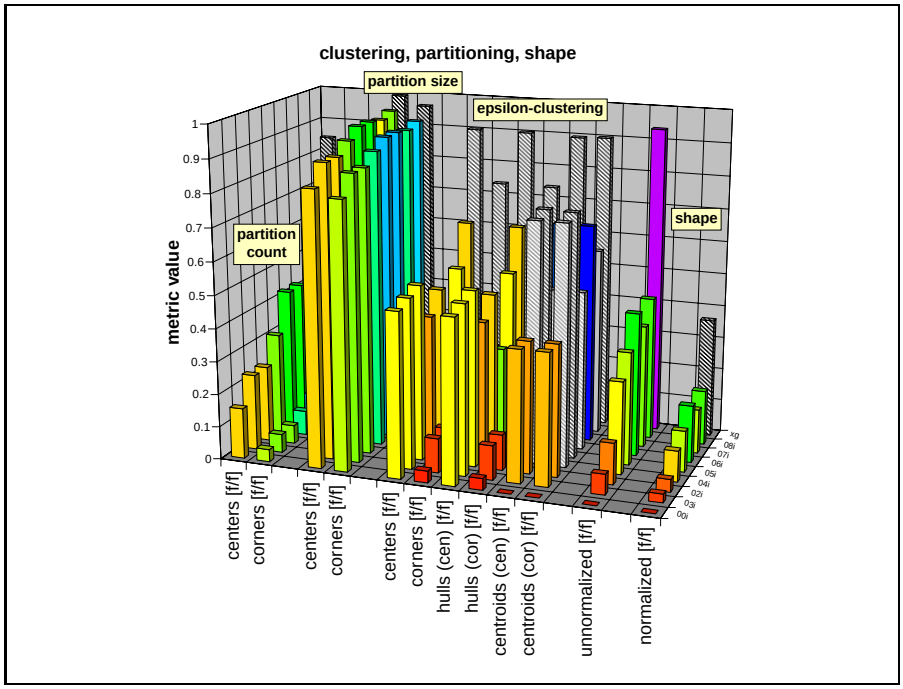


Figure 3.9: Selected metric values for each drawing in stage 2. (continued)

number of inversions: Two drawings D_i and D_j were judged to be out of order if D_i came after D_j in the metric ordering but before D_j in the human ordering, D_i and D_j were ranked equally in the metric ordering but not the human ordering, or D_i and D_j were ranked equally in the human ordering but not the metric ordering. The number of inversions was normalized by the maximum number of inversions for the sequence to adjust for different sequence lengths.

3.4.3 Experimental Results

Some of the metric names used in Figures 3.10 and 3.11 have modifiers:

- **nn-b** and **nn-w**: *unw* and *w* denote the unweighted and weighted versions, respectively
- **order**: *const* and *linear* denote the constant-weighted and linear-weighted versions, respectively
- **shape**: *norm* indicates that the normalized edit distance was used

Ordering Ability

Figure 3.10 shows the frequency with which different metric-and-pointset combinations did a particularly good or bad job of ordering the drawings for each graph. (Not all combinations of metrics and point sets shown in the figure were evaluted — the gray regions around the borders in Figure 3.10 mark combinations which were not computed.) A metric/pointset combination was flagged as doing a good job on a particular relaxation sequence if the number of inversions was noticeably lower than that for other metric/pointset combinations on the same sequence; similarly, a metric/pointset combination was flagged as doing a bad job if the number of inversions was noticeably higher than others. In a few cases no metric/pointset combination stood out as being noticeably better or worse than the others and so nothing was flagged for that sequence.

The orthogonal ordering metrics were strikingly better than most other metrics for all point sets except for those based on partition centroids. The point sets based on vertex corners fared particularly well, yielding the best ordering for over one-quarter of the sequences tested.

Shape, Euclidean distance, and the weighted nn-between metrics also stand out as being better than most of the other metrics, though they are not quite as good as orthogonal ordering.

Looking at the worst metrics, nn-within stands out as most often doing the worst job of ordering the drawings, particularly for point sets that are centroids of partitions of vertex corners. ϵ-clustering and partition size/count also do a consistently worse job of ordering.

In general, the partition centroid point sets were worse than the others, indicating that too much information is lost in these point sets to be of much use for similarity comparisons.

It is also important to consider the variation in the relative success of a given metric/pointset combination when applied to different graphs. Many did the worst job of ordering the drawings for at least one graph, and all did the best job at least once — even the most consistently best metric (orthogonal ordering) was the best only about 30% of the time. The variation is particularly large

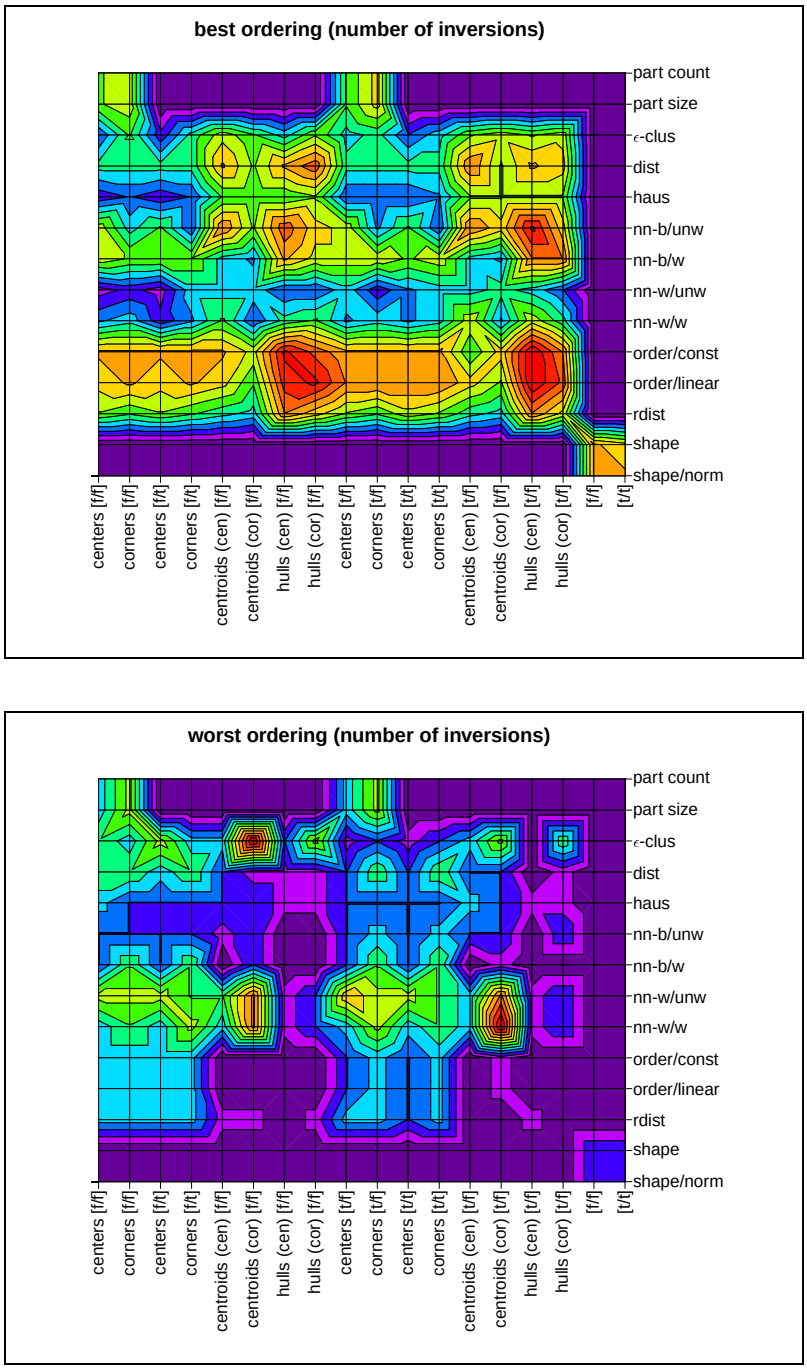


Figure 3.10: Purple is the lowest frequency; red is the highest. Peaks in the top picture indicate metric/pointset combinations which tend to produce orderings closest to the human-specified ordering; peaks in the bottom picture indicate combinations which tend to produce orderings most unlike the human-specified ordering.

for the partition size and count metrics, as shown by a middle-of-the-road ranking in both the best and worst plots in Figure 3.10. These metrics are very sensitive to the slightest movement of vertices with respect to each other, and very quickly reach their maximum values if there is much change in the drawing.

The variation in performance of the metrics for different graphs can also be seen when breaking down the results by the type of modification. Shape, average relative distance, orthogonal ordering, nn-between, and the partition size and count metrics perform better on vertex deletions. Nearly all of the metrics, except nn-within and shape, perform noticeably worse on edge insertions. The partition size and count and the Hausdorff distance metrics also perform worse on vertex insertions. This behavior is explained by noting that in InteractiveGiotto, inserting an object into the graph tends to disrupt the drawing more than a deletion.

Rotation Ability

The other criterion evaluated at this stage is how suitable the metrics are for solving the rotation problem. The measurement for each metric/pointset combination is the percentage of drawings in the relaxation sequence for which the correct rotation was chosen. The percentage correct for each combination, averaged over all of the experimental graphs, is shown in Figure 3.11. (Note that nn-within is rotation-independent and is thus not included in the chart.)

The partition size and count metrics fared the worst, getting the correct rotation only 50-54% of the time. The low results for partition size and count are largely due to many cases where the

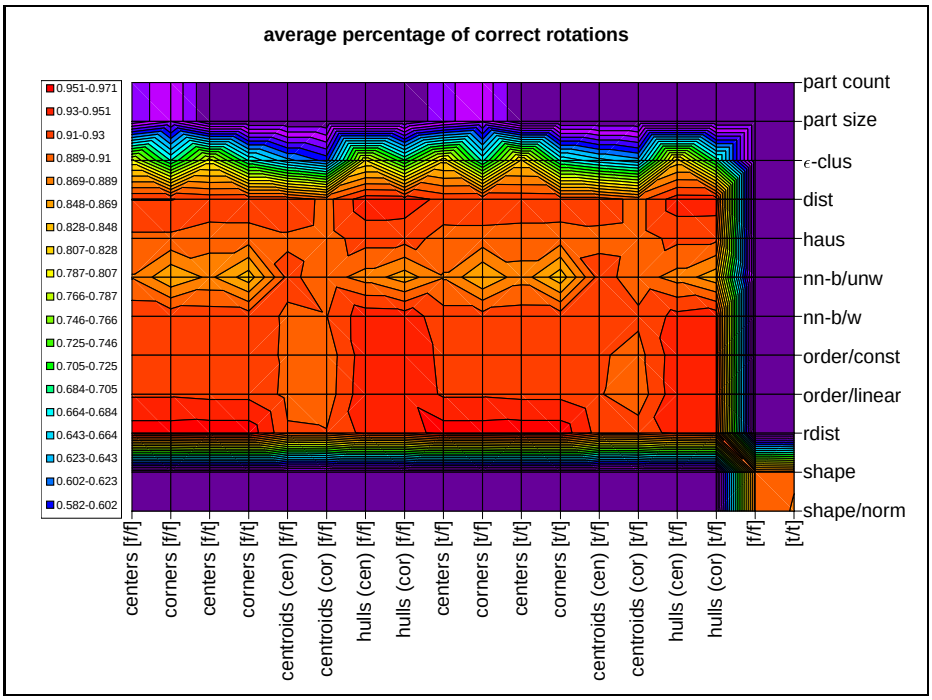


Figure 3.11: Purple is the lowest percentage correct (around 50%); red is the highest (94%).

metric could not distinguish between two or more rotations. This happened most often with the "more relaxed" drawings and reflects the fact that in these drawings the partitions tend to be very small.

The ϵ-clustering metric did somewhat better, getting 60-83% of the rotations correct; the large variation is due to whether the point set was based on vertex corners (worse) or vertex centers (better). The results here are due primarily to ϵ-clustering choosing the wrong rotation, rather than being unable to choose between multiple rotations. Point sets based on vertex corners are worse than those based on vertex centers because InteractiveGiotto places vertices so that the minimum spacing between vertices is the same as the minimum vertex dimension. As a result, the ϵ-cluster for each point typically does not contain more than four points — up to two other corners of the same vertex and up to two corners of neighboring vertices. This makes ϵ-clustering using vertex corners particularly sensitive to modifications which change vertex dimensions or separate vertices which are horizontally or vertically near each other; however, if the vertices are spaced relatively far apart compared to the vertex size in both drawings, ϵ-clustering will report a small distance.

The unweighted nn-between metric also exhibited unsatisfactory behavior for some choices of point sets, getting 83-93% of the rotations correct. This is again the result of the metric being unable to distinguish between multiple rotations for the more relaxed drawings.

The average relative distance metric stands out as the best, averaging at least 93% of the rotations correct. This average is goes up to over 97% for the center and corner point sets.

Breaking down the results by the type of modification shows that there are again some variations between groups. The correct rotation was chosen most often when edge deletions were performed, followed by vertex insertions, edge splits, vertex deletions, and edge insertions (where the best metrics achieved only about 85% correctness). In most cases the relative performance of different metric/pointset combinations is the same as in the average of all the graphs, with a few notable exceptions:

- For graphs where an edge has been inserted, average relative distance with vertex corners and vertex centers is noticeably better and unweighted nn-between is noticeably worse than the other metrics.
- For graphs where an edge has been split, unweighted nn-between is noticeably worse for point sets consisting of vertex corners.

Conclusions

If the goal is simply to solve the rotation problem, average relative distance using either corner or center point sets performs quite well. Given the success of this metric, it seems unnecessary to consider anything more complicated to solve the rotation problem.

If it is important to compare different drawings of the same graph, the orthogonal ordering metrics are the best of the metrics tested in terms of qualitative behavior. The linear-weighted version has a slight edge over the constant-weighted one. Orthogonal ordering is also nearly as good

as average relative distance in solving the rotation problem. However, there is still a good deal of room for improvement — orthogonal ordering only achieved the best ordering 30% of the time, and, considering all of the metrics, the correct ordering was found for only 9 of the 63 graphs.

3.5 Future Work

The next step is to study in more detail those cases for which certain metric/pointset combinations perform significantly worse (or better) than the average, since there are variations in performance between individual graphs. This may also indicate combinations of the existing metrics that may work better than any single one alone.

It will also be useful to test the metrics on drawings generated by other drawing algorithms. SMILE [19], for example, provides a way of obtaining many drawings of the same graph. Since the performance of some of the metrics can be traced to characteristics of InteractiveGiotto, this may prove illuminating.

Another important task is to analyze the quantitative behavior of the metrics. This requires that each drawing be assigned (by a human) a numerical value measuring how well the user's mental map is preserved. Obtaining these values is a non-trivial task, since it is difficult for a person to judge quantitatively the difference in visual distance between two pairs of drawings, even of the same graph — Is one pair twice as different as another? Only one-and-a-half times? Five percent less? Furthermore, asking if one drawing looks more like the original than another drawing may not be exactly the same question as asking which drawing does a better job of preserving the user's mental map. The chances are that a drawing which looks more like the original will do a better job of preserving the mental map, but assuming this presupposes something about how a user's mental map is structured. A solution to this may be to design an experiment in which the user gains familiarity with the original drawing, and is then timed on her response to a question involving a new drawing of the same graph. The idea is that a faster response time means that the new drawing did a better job of preserving the mental map.

Other metrics can also be developed and evaluated. Metrics based on clustering and partitioning seem particularly related to how a user navigates around the drawing, and further work is needed to extend and improve these metrics. Surveying users about what they think makes two drawings more or less similar may also lead to additional metrics. Also, combinations of the current metrics may yield better results. Both current and new metrics should also be evaluated using a larger pool of graphs, including graphs of different sizes.

Finally, once suitable metrics have been identified and validated through user studies, they can be used to compare the behavior of interactive graph drawing algorithms as well as potentially providing inspiration for new drawing algorithms.

Chapter 4

A User Study in Similarity Measures for Graph Drawing

4.1 Introduction

The question of how similar two drawings of graphs are arises in a number of situations. One application is interactive graph drawing, where the graph being visualized changes over time and it is important to preserve the user's "mental map" [115] of the original drawing as much as possible so the user does not need to spend a lot of time relearning the drawing after each update. Having to relearn the drawing can be a significant burden if the graph is updated frequently. Animation can be used to provide a smooth transition between the drawings and can help compensate for greater changes in the drawing, but it is still important to maintain some degree of similarity between the drawings to help the user orient herself to the new drawing. Related to interactive graph drawing is layout adjustment, where an existing drawing is modified so as to improve an aesthetic quality without destroying the user's mental map.

Another application is in indexing or browsing large sets of graphs. An example of a graph browser is contained the SMILE graph multidrawing system of Biedl et. al. [19]. The SMILE system tries to find a good drawing by producing many drawings of the graph and letting the user choose among them, rather than trying to code the properties of a good drawing into the algorithm. The graph browser arranges the drawings so that similar ones are near each other, to help the user navigate the system's responses. Related to this is the idea of using similarities between drawings as a basis for indexing and retrieval. Such a system has applications in character and handwriting recognition, where a written character is transformed into a graph and compared to a database of characters to find the closest match.

Some material previously published as S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. In *Graph Drawing (Proceedings of GD '00)*, volume 1984 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2000. This version submitted to the *Journal of Graph Algorithms and Applications*.

Let M be a similarity measure defined so that M's value is always nonnegative and is 0 when the drawings are identical. In order to be useful, M should satisfy three properties:

Rotation: Given drawings D and D', $M(D, D'_\theta)$ should have the minimum value for the angle a user would report as giving the best match, where D'_θ is D' rotated by an angle of θ with respect to its original orientation.

Ordering: Given drawings D, D', and D'', $M(D, D') < M(D, D'')$ if and only if a user would say that D' is more like D than D'' is like D.

Magnitude: Given drawings D, D', and D'', $M(D, D') = \frac{1}{c}M(D, D'')$ if and only if a user would say that D' is c times more like D than D'' is like D.

This paper describes a user study performed in order to evaluate several potential similarity measures with respect to rotation and ordering, and to test a possible method for obtaining data to be used for evaluating measures with respect to magnitude. Data cannot be collected directly for the magnitude part as it can be for rotation and ordering because it is very difficult to assign numerical similarity values to pairs of drawings; Wickelgren [154] observes that it is more difficult to assign numerical values than to judge ordering. As a result, other data must be gathered — for example, response times on a particular task — with the hope that the data is sufficiently related to the actual similarity values to be useful. This can be partially tested by using the data (e.g., response times) to order the drawings, and determining whether the results are consistent with user responses on the ordering part.

This study improves on our previous work [38] in several ways:

- **More Experimental Data:** A larger pool of users (103 in total) was used for determining the "correct" behavior for the measure.
- **Refined Ordering Part:** Users made only pairwise judgements between drawings rather than being asked to order a larger set.
- **Addressing of Magnitude Criterion:** The previous experiment did not address magnitude at all.
- **More Realistic Drawing Alignment:** The previous drawing alignment method allowed one drawing to be scaled arbitrarily small with respect to the other; the new method keeps the same scale factor for both drawings.
- **Refinement of Measures:** For those measures computed with pairs of points, pairs involving points from the same vertex are skipped.
- **New Measures:** Several new measures have been included.

We describe the experimental setup in Section 4.2, the measures evaluated in Section 4.3, the results in Section 4.4, and conclusions and directions for future work in Section 4.5.

4.2 Experimental Setup

This study focuses on similarity measures for orthogonal drawings of nearly the same graph. "Nearly the same graph" means that only a small number of vertex and edge insertions and deletions are needed to transform one graph into the other. In this study, the graphs differ by one vertex and two or four edges. The focus on orthogonal drawings is partly practical, given the availability of an orthogonal drawing algorithm capable of producing many drawings of the same graph, and partly motivated by the amount of work done on interactive orthogonal drawing algorithms. (See, for example, Biedl and Kaufmann [18], Fößmeier [73], Papakostas, Six, and Tollis [126], and Papakostas and Tollis [125].) Being able to produce multiple drawings of the same graph is key, because it can be very difficult to judge if one pair of drawings is more similar than another if the graphs in each pair are different.

4.2.1 Graphs

The graphs used in the study were generated from a base set of 20 graphs with 30 vertices each, taken from an 11,582-graph test suite. [57] Each of 20 base graphs was first drawn using Giotto [146]. Each of the Giotto-produced base drawings was modified by adding a degree 2 and a degree 4 vertex, for a total of 40 modified drawings. Each modified drawing is identical to its base drawing except for the new vertex and its adjacent edges, which were placed in a manner intended to mimic how a user might draw them in an editor. Because InteractiveGiotto (used in the next step) preserves edge crossings and bends, routing the edges realistically avoids introducing a large number of extra crossings. Finally, a large number of new drawings were produced for each modified drawing using InteractiveGiotto [43], and four drawings were chosen from this set. The four drawings chosen range from very similar to the base drawing to significantly different.

4.2.2 Definition

The experiment consisted of three parts, to address the three evaluation criteria. In all cases, the user was asked to respond as quickly as possible without sacrificing accuracy. Each trial timed out after 30 seconds if the user did not respond.

Rotation Part The rotation part directly addresses the rotation criterion. The user is presented with a screen as shown in Figure 4.1. The one drawing D on the left is the base drawing; the eight drawings $D_1, \ldots, D_8$ on the right are eight different orientations of the same new (InteractiveGiotto-produced) drawing derived from D. The eight orientations consist of rotations by the four multiples of $\pi/2$, with and without an initial flip around the x-axis. For orthogonal drawings, only multiples of $\pi/2$ are meaningful since it is clear that rotation by any other angle is not the correct choice. The vertices are not labelled in any of the drawings to emphasize the layout of the graph over the specifics of vertex names.

The user's task is to choose which of $D_1, \ldots, D_8$ looks most like the base drawing. A "can't decide" button is provided for cases in which the drawings are too different and the user cannot make a choice. The user's choice and the time it took to answer are recorded.

Ordering Part The ordering part directly addresses the ordering criterion. In this part, the user is presented with a screen as shown in Figure 4.2. The one drawing D on the left is the base drawing; the two drawings D_1 and D_2 on the right are two different (InteractiveGiotto-produced) new drawings of the same modified drawing derived from D.

The user's task is to choose which of D_1 and D_2 looks most like the base drawing. A "can't decide" button is provided for cases in which the drawings are too different and the user cannot make a choice. The user's choice and the time it took to answer are recorded.

Difference Part The difference part addresses the magnitude criterion by gathering response times on a task, with the assumption that a greater degree of similarity between the drawings will help the user complete the task more quickly. The screen presented to the user is shown in Figure 4.3. The drawing D on the left is the base drawing; the drawing D_1 on the right is one of the InteractiveGiotto-produced new drawings derived from D.

The user's task is to identify the vertex present in the right drawing that is not in the left drawing. The vertices are labelled with random two-letter names — corresponding vertices in drawings in a single trial have the same name, but the names are different for separate trials using the same base drawing to prevent the user from simply learning the answer. Displaying the vertex names makes the task less difficult, and mimics the scenario where the user is working with a dynamically updated graph where the vertex labels are important.

The user's choice and the time it took to answer are recorded.

4.2.3 Methodology

The three parts were assigned to students as part of a homework assignment in a second-semester CS course at Brown University. A total of 103 students completed the problem.

Before being assigned the problem, the students had eight lectures on graphs and graph algorithms, including one on graph drawing. They had also been assigned a programming project involving graphs, so they had some familiarity with the subject.

The homework problem made use of an online system which presented the displays shown in Figures 4.1, 4.2, and 4.3. A writeup was presented with the problem explaining how to use the system, and the directions were summarized each time the system was run.

Each of the three parts was split into four runs, so the students would not have to stay focused for too long without a break. The graphs used were divided into 10 batches: the first batch (the practice batch) contained two modified drawings along with their associated new drawings, and each of the other nine batches contained three modified drawings and the associated new drawings. All of the students were assigned the practice batch for the first run of each part, and were randomly

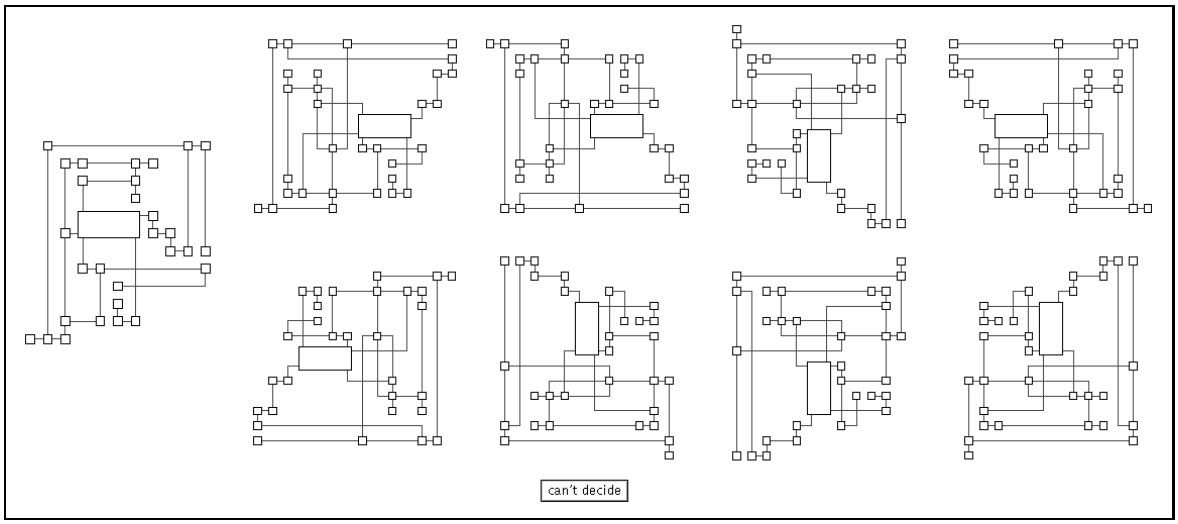


Figure 4.1: The rotation part.

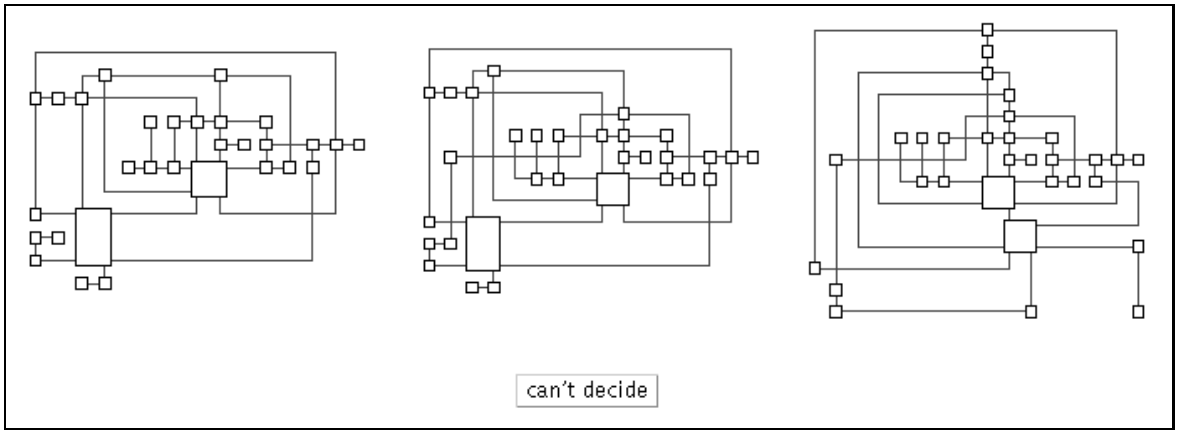


Figure 4.2: The ordering part.

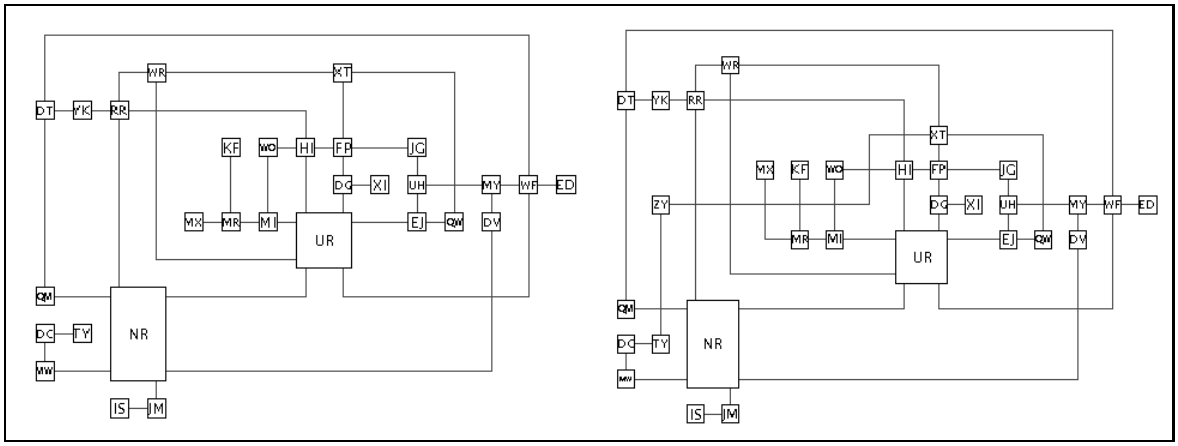


Figure 4.3: The difference part.

assigned three of the other batches for later runs so that each batch was completed by 1/3 of the students. A given student worked with the same batches for all three of the parts. Within each run of the system, the individual trials were presented in a random order and the order of the right-hand drawings in the rotation and ordering parts was chosen randomly.

After the students completed all of parts, they answered a short questionnaire about their experiences. The questions asked were as follows:

1. **(Ordering and Rotation)** What do you think makes two drawings of nearly the same graph look similar? Are there factors that influenced your decisions? Did you find yourself looking for certain elements of the drawing in order to make your choice?

2. **(Difference)** What factors helped you locate the extra vertex more quickly? Did you compare the overall look of the two drawings in order to aid your search, or did you just scan the second drawing?

3. **(All Parts)** As you consider your answers, think about what this means for a graph drawing algorithm that seeks to preserve the look of the drawing. What types of things would it have to take into account?

4.3 Measures Evaluated

All of the measures evaluated in this study are described below. Most are the same as or similar to those described in [38]; the primary difference is that all of the measures have been scaled so that 0 means the drawings are identical and 1 is the maximum difference. The upper bound is frequently based on the worst-case scenario for point positioning and may not be achievable by an actual drawing algorithm. (For example, the upper bound may only be achieved when all of the vertices are placed on top of each other, an impossible situation with most drawing algorithms.)

4.3.1 Preliminaries

Corresponding Objects Most of the measures make use of the fact that the graph in the drawings being compared is the same. (If the graphs are not the same, those parts that are different can be ignored and only the common subgraphs used.) This means that each vertex and edge of G has a representation in each of the drawings, and it is meaningful to talk about the *corresponding vertex or edge* in one drawing given a vertex or edge in the other drawing.

Point Set Selection With the exception of the shape measures, all of the measures are defined in terms of point sets derived from the edges and vertices of the graph rather than the edges and vertices themselves. Points can be selected in a variety of ways; inspired by North [120], one point set contains the four corners of each vertex. A second point set, suggested by feedback from the study (section 4.4.4), is a subset of the corner point set containing only points near the edge of the

drawing. Like vertices and edges, each point in one drawing has a corresponding point in the other drawing.

A change from the previous experiment [38] is that many of the measures computed using pairs of points now do not include pairs of points derived from the same vertex. This can have a great effect on measures which involve nearest neighbors, for example, because a point's nearest neighbor will often be another corner of the same vertex, which does not convey much information about how that vertex relates to other vertices in the drawing. In this study, the point's nearest neighbor is the nearest point from a different vertex. This is not explicitly written in the definitions below for clarity of notation, but it should be assumed unless stated otherwise.

Drawing Alignment For the measures involving the comparison of coordinates between drawings, the value of the measure is very dependent on how well the drawings are aligned. Consider two identical drawings, but let the x-coordinate of each point in the second drawing be one bigger than the x-coordinate of the corresponding point in the first drawing. The average distance moved by each point will be reported as 1, even though the drawings actually are the same. Aligning the drawings before comparing the coordinates removes this effect.

In the previous experiment [38], alignment was done by simultaneously adjusting the scale and translation of one drawing with respect to the other so as to minimize the distance squared between corresponding points. This had the effect of potentially reducing one drawing to a very small area if the drawings did not match well. This has been replaced by a new alignment method which separates the determination of the scale and translation factors into two steps. First, the scale factor is set to ensure that the two drawings are drawn to the same scale. Since the drawings are orthogonal drawings, there is a natural underlying grid which can be used to adjust the scale. Once scaled, the translation factor is chosen so as to minimize the distance squared between corresponding points. The new alignment method is intended to better match how a person might try to match up the drawings — it does not seem likely that someone would mentally shrink or enlarge one drawing drastically with respect to the other, but rather would work with the current scale and try to adjust the translation.

Suitability for Ordering vs. Rotation and Ordering Some of the measures do not depend on the relative rotation of one drawing with respect to the other. This means that they fail the rotation test, however, they are included because there may be situations in which the measure is not being used to determine the proper rotation for the drawings. Furthermore, a successful ordering-only measure could be combined with one which is successful at rotation but less so at ordering to obtain a measure which is good at both. Measures suitable for ordering only are marked [order only] below.

Notation In the following, P and P' will always refer to the point sets for drawings D and D', respectively, and $p' \in P'$ will be the corresponding point for $p \in P$ (and vice versa). Let $d(p, q)$ be the Euclidean distance between points p and q.

4.3.2 Degree of Match

The following measures measure a degree of matching between the point sets by looking at the maximum mismatch between points in one set and points in another. The motivation for these measures is straightforward — if point sets are being used to represent the drawings, then classical measures of point set similarity can be used to compare the drawings.

Undirected Hausdorff Distance The *undirected Hausdorff distance* is a standard metric for determining the quality of the match between two point sets. It does not take into account the fact that the point sets may be labelled.

$$\text{haus}(P, P') = \max \{ \max_{p \in P} \min_{q' \in P'} d(p, q') \, , \, \max_{p' \in P'} \min_{q \in P} d(p', q) \}$$

Paired Hausdorff Distance The *paired Hausdorff distance* is an adaptation of the undirected Hausdorff distance for labelled point sets, and is defined as the maximum distance between two corresponding points:

$$\text{phaus}(P, P') = \max_{p \in P} d(p, p')$$

4.3.3 Position

These measures are motivated by the idea that the location of the points on the page is important, and points should not move too move far between drawings.

Average Distance *Average distance* is the average distance points move between drawings.

$$\text{dist}(P, P') = \frac{1}{|P|} \sum_{p \in P} d(p, p')$$

Nearest Neighbor Between *Nearest neighbor between* is based on the assumption that a point's original location should be closer to its new position than any other point's new position.

$$\text{nnb}(P, P') = \frac{1}{\text{UB}} \sum_{p \in P} \text{weight}(\text{nearer}(p))$$

where

$$\text{nearer}(p) = \{ q \mid d(p, q') < d(p, p') \, , \, q \in P \, , \, q \neq p \}$$

Unweighted In the unweighted version, the score for p counts only whether or not there are points in P' between p and p'.

$$\begin{aligned} \text{weight}(S) &= \begin{cases} 0 & \text{if } |S| = 0 \\ 1 & \text{otherwise} \end{cases} \\ \text{UB}(n) &= |P| \end{aligned}$$

Weighted In the weighted version, the number of points in P' between p and p' is taken into account.

$$\begin{aligned} \text{weight}(S) &= |S| \\ \text{UB}(n) &= |P|\,(|P|-1) \end{aligned}$$

4.3.4 Relative Position

These measures are based on the idea that the relative position of points should remain the same. There are two components to relative position — the distance between the points and the angles. All of the measures except for average relative distance are concerned with the angles.

Orthogonal Ordering *Orthogonal ordering* measures the change in angle between pairs of points.

Let θ_{pq} be the counterclockwise angle between the positive x-axis and the vector $q-p$.

$$\text{order}(P,P') = \frac{1}{W}\min\left\{\int_{\theta_{pq}}^{\theta_{p'q'}} \text{weight}(\theta)\,d\theta, \int_{\theta_{p'q'}}^{\theta_{pq}} \text{weight}(\theta)\,d\theta\right\}$$

Constant-Weighted In the constant-weighted version, all changes of angles are weighted equally.

$$\begin{aligned} \text{weight}(\theta) &= 1 \\ W &= \pi \end{aligned}$$

Linear-Weighted In the linear-weighted version, changes in the north, south, east, west relationships are weighted more heavily than changes in angle which do not affect this relationship.

$$\begin{aligned} \text{weight}(\theta) &= \begin{cases} \frac{(\theta \bmod \pi/2)}{\pi/4} & \text{if } (\theta \bmod \pi/2) < \pi/4 \\ \frac{\pi/2-(\theta \bmod \pi/2)}{\pi/4} & \text{otherwise} \end{cases} \\ W &= \pi/2 \end{aligned}$$

Ranking The *ranking* measure considers the relative horizontal and vertical position of the point. This is a component of the similarity measure used in the SMILE graph multidrawing system. [19] Let right(p) and above(p) be the number of points to the right of and above p, respectively.

$$\text{rank}(P,P') = \frac{1}{\text{UB}}\sum_{p\in P}\min\{\,|\,\text{right}(p)-\text{right}(p')\,|+|\,\text{above}(p)-\text{above}(p')\,|\,,\ \text{UB}\,\}$$

where

$$\text{UB} = 1.5\,(|P|-1)$$

Of note here is that the upper bound is taken as $1.5\,(|P|-1)$ instead of $2\,(|P|-1)$, the actual maximum value occurring when a point moves from one corner of the drawing to the opposite corner. The motivation for this is simply that it scales the measure more satisfactorily.

Average Relative Distance [order only] The *average relative distance* is the average change in distance between pairs of points.

$$\text{rdist}(P, P') = \frac{1}{|P|\,(|P|-1)} \sum_{p,q \in P} |\, d(p,q) - d(p',q') \,|$$

λ-Matrix [order only] The λ-matrix model is used by Lyons, Meijer, and Rappaport [108] to evaluate cluster-busting algorithms. It is based on the concept of order type used by Goodman and Pollack [82], where two sets of points P and P' have the same order type if, for every triple of points (p,q,r), they are oriented counterclockwise if and only if (p',q',r') are also oriented counterclockwise.

Let $\lambda(p,q)$ be the number of points in P to the left of the directed line from p to q.

$$\text{lambda}(P, P') = \frac{1}{\text{UB}} \sum_{p,q \in P} |\, \lambda(p,q) - \lambda(p',q') \,|$$

where the upper bound for a set of size n is:

$$\text{UB}(n) = n \left\lfloor \frac{(n-1)^2}{2} \right\rfloor$$

4.3.5 Neighborhood

These measures are guided by the philosophy that each point's neighborhood should be the same in both drawings. The measures do not explicitly take into account either the point's absolute position or its position relative to other points.

Nearest Neighbor Within [order only] For *nearest neighbor within*, a point's neighborhood is simply its nearest neighbor.

Let $\text{nn}(p)$ be the nearest neighbor of p in the p's point set and $\text{nn}(p)'$ be the corresponding point in P' to $\text{nn}(p)$. Ideally, $\text{nn}(p)'$ should be p''s nearest neighbor.

$$\text{nnw}(P, P') = \frac{1}{\text{UB}} \sum_{p \in P} \text{weight}(\text{nearer}(p))$$

where

$$\text{nearer}(p) = \{\, q \mid d(p',q') < d(p', \text{nn}(p)')\,,\; q \in P\,,\; q \neq p\,,\; q \neq \text{nn}(p) \,\}$$

Unweighted The unweighted version considers only whether or not $\text{nn}(p)'$ is p''s nearest neighbor.

$$\begin{aligned} \text{weight}(S) &= \begin{cases} 0 & \text{if } |S| = 0 \\ 1 & \text{otherwise} \end{cases} \\ \text{UB}(n) &= |P| \end{aligned}$$

Weighted The weighted version takes into account the number of points in P' closer to p' than $\mathrm{nn}(p)'$.

$$\begin{aligned} \mathrm{weight}(S) &= |S| \\ \mathrm{UB}(n) &= |P|\,(|P|-1) \end{aligned}$$

ϵ-Clustering [order only] *ϵ-clustering* defines the neighborhood for each point to be its ϵ-cluster, the set of points within a distance ϵ, defined as the maximum distance between a point and its nearest neighbor.

$$\mathrm{eclus} = 1 - \frac{|S_I|}{|S_U|}$$

where

$$\begin{aligned} \epsilon &= \max_{p \in P} \min_{q \in P, q \neq p} d(p,q) \\ S_I &= \{\, (p,q) \mid p \in P\,,\ q \in \mathrm{clus}(p,P,\epsilon) \text{ and } q' \in \mathrm{clus}(p',P',\epsilon') \,\} \\ S_U &= \{\, (p,q) \mid p \in P\,,\ q \in \mathrm{clus}(p,P,\epsilon) \text{ or } q' \in \mathrm{clus}(p',P',\epsilon') \,\} \\ \mathrm{clus}(p,P,\epsilon) &= \{\, q \mid d(p,q) \leq \epsilon\,,\ q \in P\,,\ q \neq p \,\} \end{aligned}$$

Separation-Based Clustering [order only] In the *separation-based clustering* measure, points are grouped so that each point in a cluster is within some distance δ of another point in the cluster and at least distance δ from any point not in the cluster. The intuition is that the eye naturally groups things based on the surrounding whitespace.

Formally, for every point p in cluster C such that $|C| > 1$, there is a point $q \neq p \in C$ such that $d(p,q) < \delta$ and there is no point $r \notin C$ such that $d(p,r) < \delta$. If C is a single point, only the second condition holds.

Let $\mathrm{clus}(p)$ be the cluster to which point p belongs.

$$\mathrm{sclus} = 1 - \frac{|S_I|}{|S_U|}$$

where

$$\begin{aligned} S_I &= \{\, (p,q) \mid p,q \in P\,,\ \mathrm{clus}(p) = \mathrm{clus}(q) \text{ and } \mathrm{clus}(p') = \mathrm{clus}(q') \,\} \\ S_U &= \{\, (p,q) \mid p,q \in P\,,\ \mathrm{clus}(p) = \mathrm{clus}(q) \text{ or } \mathrm{clus}(p') = \mathrm{clus}(q') \,\} \end{aligned}$$

4.3.6 Edges

Shape The *shape* measure treats the edges of the graph as sequences of north, south, east, and west segments and compares these sequences using the edit distance.

$$\mathrm{shape} = \frac{1}{\mathrm{UB}} \sum_{e \in E} \mathrm{edits(e,e')}$$

Regular The edit distance is not normalized for the length of the sequence, and the upper bound is as follows:

$$\mathrm{UB} = \sum_{e \in E} |\,\mathrm{length}(e) - \mathrm{length}(e')\,| + \min\{\,\mathrm{length}(e)\,,\,\mathrm{length}(e')\,\}$$

Normalized The edit distance is normalized for the length of the sequence using the algorithm of Marzal and Vidal [109], and the upper bound is as follows:

$$\text{UB} = |E|$$

4.4 Results

4.4.1 Rotation

Correctness Let an individual trial T be described by the tuple $(D, D_1, \ldots, D_8)$, where D is the base drawing and the D_i are the eight rotations the user must choose between. Let T_M be the measure's choice for trial T:

$$T_M = \begin{cases} D_k & \text{if } M(D, D_k) < M(D, D_i)\ \forall i \neq k \\ \text{tie} & \text{if } \exists\ j,k \text{ such that } M(D, D_j) = M(D, D_k) \leq M(D, D_i)\ \forall i \neq j,k\ (j \neq k) \end{cases}$$

Also, let T_k denote user k's response for trial T:

$$T_k = \begin{cases} D_k & \text{if the user chose drawing } D_k \\ \text{tie} & \text{if the user clicked the "can't decide" button or the trial timed out} \end{cases}$$

Note that T_k is only defined if user k was presented with trial T — each trial was completed by about one-third of the users.

Define the correctness of the measure M with respect to user k for T as $C(M, T, k)$:

$$C(M, T, k) = \begin{cases} 1 & \text{if } T_M = T_k \\ 0 & \text{otherwise} \end{cases}$$

$C(M, T, k)$ is undefined if T_k is undefined.

Let $\mathcal{T}$ be the set of pairs (T, k) for which T_k is defined. Then the overall correctness C_M for measure M is

$$C_M = \frac{\sum_{(T,k) \in \mathcal{T}} C(M, T, k)}{|\mathcal{T}|}$$

Figure 4.4 shows the correctness C_M for each measure on the rotation task. The light yellow area behind the columns shows the average correctness for each measure over all of the trials. The four columns for each measure show the breakdown according to type of new drawing in the trial: *similar* for drawings virtually identical to the base, *features* for drawings somewhat different from the base but with noticeable features to help identification, *contradictory* for drawings with noticeable features but where those features contradicted each other (for example, when one feature was rotated with respect to another), and *different* for drawings that are very different from the base. (The "different" set only contained two drawings, so these results should be generalized with caution.)

Finally, the "mode" column indicates the best possible performance that could be expected of a measure — because approximately 34 students gave responses for each set of drawings, there was

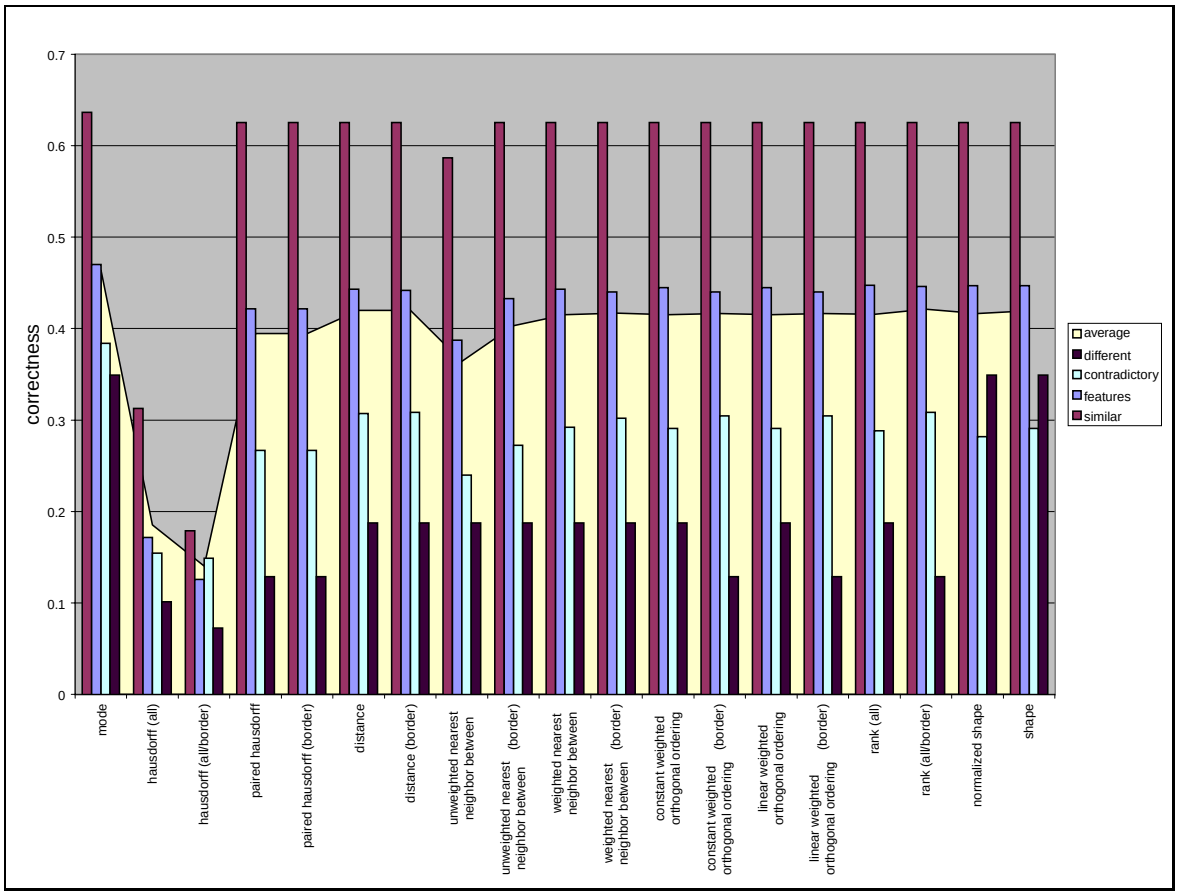


Figure 4.4: Results for the rotation part. "border" indicates that the border point set was used; "all" indicates that pairs of points derived from the same vertex were included.

the potential for getting different "correct" user responses for the same set of drawings. Since the measures always choose the same response for the same set of drawings, the best score a measure can get is if it always makes the choice that was most common among the users. Let $f(T, r)$ be the frequency with which the users completing trial T picked response $r \in \{D_1, \ldots, D_8, \text{tie}\}$. Also define the most common response T_{mode} as the response r for which $f(T, r)$ is maximized, and the correctness of the most common response for trial T and user k as

$$C(\text{mode}, T, k) = \begin{cases} 1 & \text{if } T_{\text{mode}} = T_k \\ 0 & \text{otherwise} \end{cases}$$

Then the "best possible score" shown in the mode column is:

$$C_{\text{best}} = \frac{\sum_{(T,k)\in\mathcal{T}} C(\text{mode}, T, k)}{|\mathcal{T}|}$$

Note that there is at least some user agreement as to a correct answer even for the most different drawings — if users truly could not distinguish between the drawings, one would expect each choice

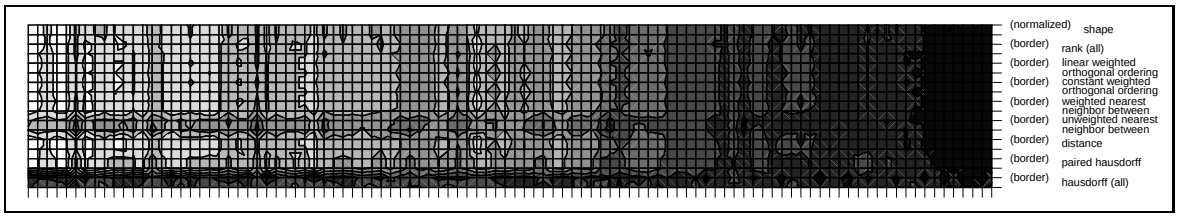


Figure 4.5: Rotation correctness for individual users. Users are shown along the x-axis; levels of gray indicate the correctness, ranging from 0 (black) to .8 (white).

to get approximately 1/8 of the responses and thus have a correctness of 0.125 in the mode column.

The average correctness for even the best measures is disappointingly low — below 50%! — meaning that even the best measure will tend to rotate drawings incorrectly much of the time. However, as one might expect, most of the measures performed better when the new drawing was more similar to the base drawing.

The correctness results are more encouraging when compared to the best possible score in the mode column. With the exception of Hausdorff distance (on both point sets) and unweighted nearest neighbor between (on the "all" point set), all of the measures performed quite well on average with respect to the mode; the performance is best for the most similar drawings and decreases as the drawings become increasingly different. The good relative performance for the most similar drawings indicates that, in these cases, the measures tend to perform as well as can be expected and the low correctness results stem from disagreements between users about the correct answer. However, the drop in relative performance with the most different drawings suggests that the measures are not picking up on the more subtle aspects of similarity — there is still some user consensus, so the users are finding some ideas of similarity that the measures do not.

Per-User Correctness The low correctness for even the mode (64% for the most similar drawings) indicates that while there is a clear preferred choice, there are a significant number of users who pick other choices. This suggests that users have different ideas about what factors make drawings look more similar, or different ideas about the relative importance of different aspects of similarity. Let $\mathcal{T}_k$ be the set of trials for which T_k is defined, i.e., the set of trials user k completed. Then the correctness of measure M for user k is

$$C_{M,k} = \frac{\sum_{T \in \mathcal{T}_k} C(M, T, k)}{|\mathcal{T}_k|}$$

Figure 4.5 shows each measure's correctness on a per-user basis. It shows two things — the measures that do badly overall (Hausdorff distance, unweighted nearest neighbor between) perform badly for every user, and that some users are predicted better than others by the measures. There is relatively little difference between the good measures for a single user, though it is interesting to note that while the border-only point sets lead to slightly worse results overall (especially for the more different drawings), measures using these point sets perform better for certain users.

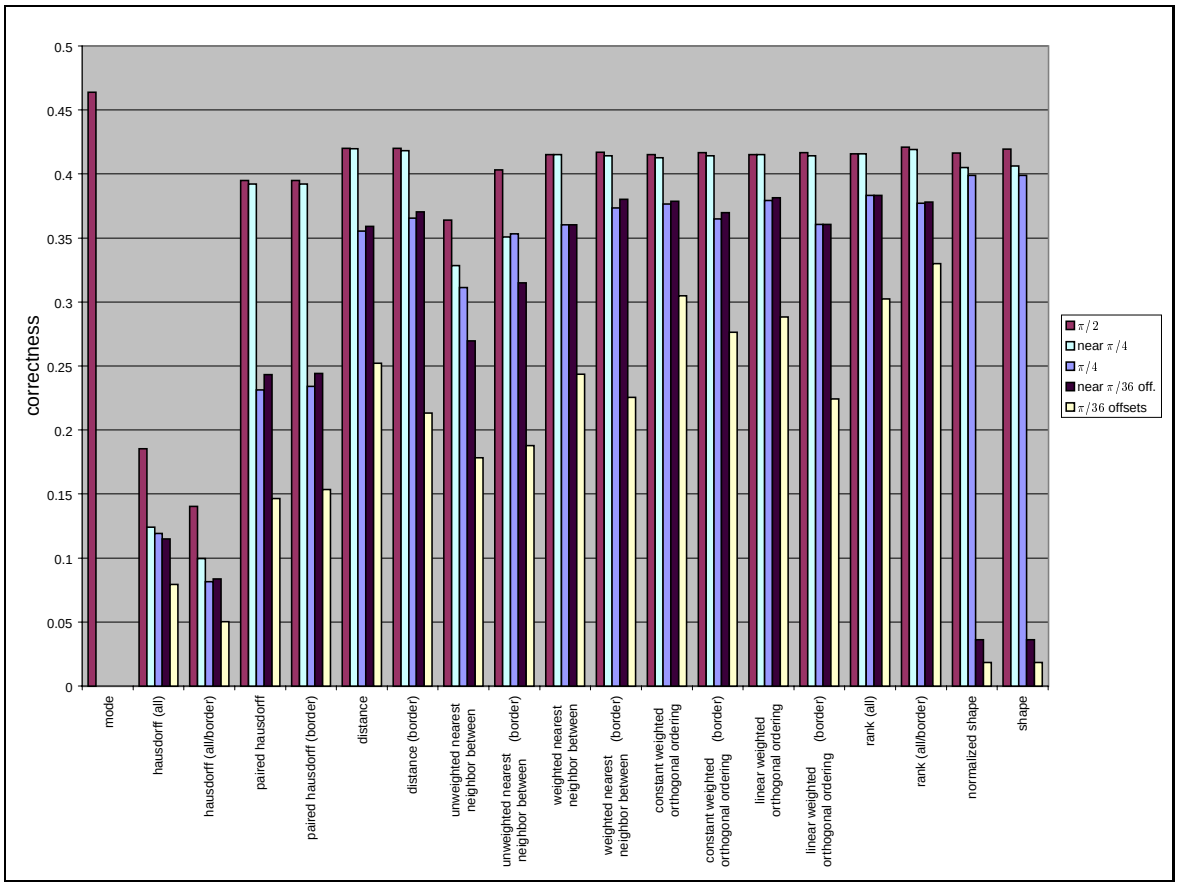


Figure 4.6: Rotation results for angles other than $\pi/4$.

Other Angles The rotation task focused on differences of $\pi/2$ in the rotation angle — a very large difference, though the only meaningful difference for orthogonal drawings since a user can easily tell that a rotation by some other angle is not the best match. Figure 4.6 shows the results for three sets of orientations: the four multiples of $\pi/2$ between 0 and 2π (labelled "$\pi/2$"), the eight multiples of $\pi/4$ (labelled "$\pi/4$"), and the eight multiples of $\pi/4$ plus the four multiples of $\pi/2$ offset by $\pi/36$ (i.e., $\pi/36, 19\pi/36, 37\pi/36, 55\pi/36$, labelled "$\pi/36$ offsets"), all with and without an initial flip around the x-axis. The "near" columns show the correctness if the measure is considered to choose the right rotation if it picked a rotation near the user's response — $\pm\pi/4$ for "$\pi/4$" and $\pm\pi/36$ for "$\pi/36$ offsets."

Ideally there would be no change in correctness with the addition of the extra orientations since the user's "correct" answer does not change, but a drop can be seen. The sharp dropoff for shape on the "$\pi/36$ offsets" is a result of a large number of ties between a rotation of θ and one of $\theta + \pi/36$. If the correctness criterion is relaxed so that measures are only expected to choose a rotation near the correct one, the measures perform better. This indicates that when the wrong rotation is chosen, it tends to be near the right one. This suggests that while the measures would likely perform less

satisfactorily when asked to pick the correct rotation in non-orthogonal applications, they would still perform reasonably well if the goal is only to obtain approximately the right orientation.

4.4.2 Ordering

Correctness For the ordering task, an individual trial T is described by the tuple (D, D_1, D_2), where D is the base drawing and D_1 and D_2 are the drawings the user must choose between. The correctness C_M of a measure M is defined in the same manner as for the rotation task.

Figure 4.7 shows how the correctness for each measure in the ordering task. The light yellow area behind the columns again shows the average correctness for each measure over all of the trials, and the columns show the breakdown according to the types of the new drawings in the trial. A few combinations with only a few samples each are not shown for space reasons.

Most of the measures again perform quite well when compared to the mode — generally above 90% on average — and the most notable exceptions include those measures that performed most poorly on rotation. Also, as expected, the measures generally performed better when one of the drawings was clearly more like the base drawing than the other.

Per-User Correctness While the correctness for the mode is reasonably high, at least for the best cases, not all users agree on the correct choice. Figure 4.8 shows each measure's correctness on a per-user basis. This is again calculated as it was for the rotation task. As for rotation, those measures that perform badly overall tend to be the worst performers for each individual user as well. However, there are a number of cases where measures that perform better than average for some users perform worse than average for others.

4.4.3 Difference

The goal in the difference part was to be able to use the user's response times as an indicator of similarity, the idea being that a user can locate the new vertex faster if the drawings are more similar. As a test of the validity of this, the times on the difference part were used to order the pairs of drawings used in the ordering task. The results were very unsatisfactory, achieving only 45% correctness (compare to Figure 4.7, where even the worst measure reached nearly 58% correctness). As a result, the times on the difference task are not a good indicator of similarity and are not suitable for evaluating measures with respect to the magnitude criterion. Using the times for the rotation task was similarly ineffective.

4.4.4 User Responses

The students' responses to the final questionnaire yielded several interesting notes. As might be expected, the responses as to what makes two drawings look similar in the rotation and ordering parts included a sizable percentage (35%) who said preserving the position, size, number of large vertices was important and another large percentage (44%) who said they looked for distinctive

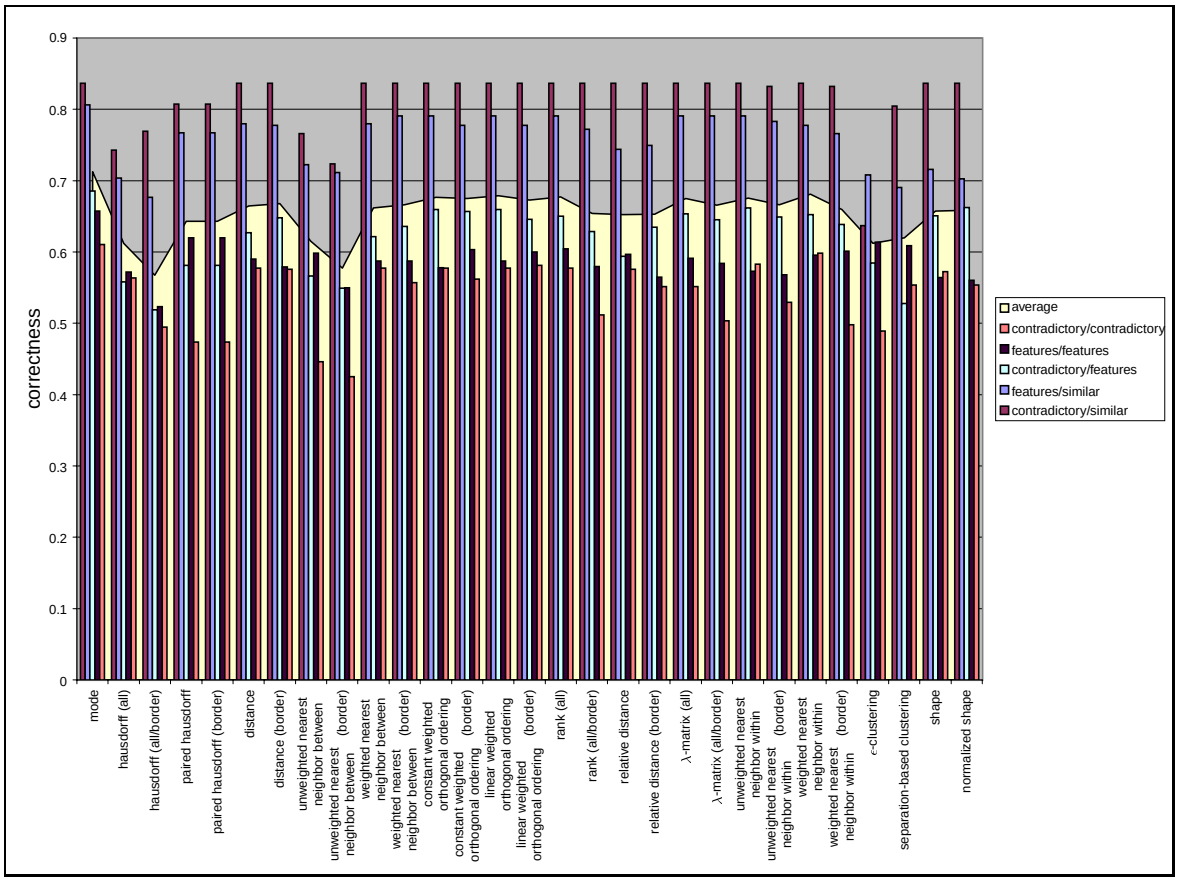


Figure 4.7: Results for the ordering part. "border" indicates that the border point set was used; "all" indicates that pairs of points derived from the same vertex were included.

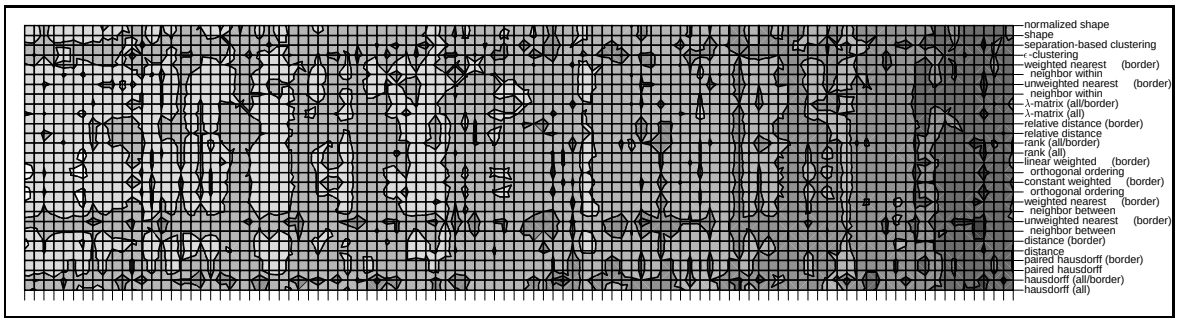


Figure 4.8: Ordering correctness for individual users. Users are shown along the x-axis; levels of gray indicate the correctness, ranging from 0 (black) to .8 (white).

clusters and patterns of vertices, such as chains, zigzags, and degree 1 vertices. More surprising was that 44% of the students said that borders and corners of the drawing are more important than the interior when looking for similarity. This is supported by research in cognitive science indicating that people often treat filled and outline shapes as equivalent, focusing primarily on the external contour (Wickelgren [154]). A number of these students mentioned the importance of "twiddly bits around the edges" — distinctive clusters and arrangements of vertices, made more obvious by being on the border. Related comments were also that the orientation and aspect ratio of the bounding box should remain the same, and that the outline of the drawing should not change. Another sizable group (34%) commented that the "general shape" of the drawing is important.

For the question about the difference part, several users expressed frustration at the difficulty of the task and commented that the system frequently timed out on them. The usefulness of the "big picture" view — looking at the overall shape of the drawing — was contested, with nearly equal numbers reporting that the overall look was useful in the task, and that it was confusing and misleading. About 16% of the users mentioned limited use of the overall look, using it on a region-by-region basis to quickly eliminate blocks that remained the same and falling back on simply scanning the drawing or matching corresponding vertices and tracing edges when the regions were too different. Another 24% reported using vertex-by-vertex matching from the beginning. A similar-sized group (20%) figured out shortcuts, such as that the edges added along with the new vertex frequently caused one of the neighboring vertices to have a degree larger than 4 and thus be drawn with a larger box, so they scanned for the neighbors of the large boxes to find the new vertex. Overall, a large number (28%) reported searching for vertices with extra edges rather than searching for new vertex directly.

For the final question, about what a graph drawing algorithm should take into account if the look of the drawing should be preserved, the most common answers echoed those from the rotation/ordering question: maintaining vertex size and shape, the relative positions of vertices, the outline of the drawing, and clusters.

4.5 Conclusions and Future Work

Several conclusions can be drawn from these results:

- The results from the rotation and ordering parts show that, with the exception of several measures that perform noticeably worse than the others, there is not a large difference in the performance of the tested measures. Distance and weighted nearest neighbor between perform slightly better than the rest on both point sets for both tasks; both orthogonal ordering measures and rank also perform well on the full point set. The λ-matrix and weighted nearest neighbor within measures have a slight edge on the ordering task.

- It is interesting to note that the worst-performing measures (undirected Hausdorff distance, unweighted nearest neighbor between, ϵ-clustering, and spacing clustering) give the least weight

to absolute and relative point positions, suggesting that absolute and relative point positions are indeed important to similarity. Furthermore, it suggests that point positions are less significant in the ordering task because the relative performance of the worst measures with respect to the mode was not as bad.

- The per-user analysis suggests that while it is meaningful to talk about "good" measures and "bad" measures in overall terms, to get the maximum performance it may be necessary to tailor the specific similarity measure used to a particular user.

- In the ordering task, the lack of difference between the full point set and the borders-only point set for the better measures seems to mesh well with the students' comments about the border being very important in the look of the drawing. For the rotation task, only the lack of difference for the orthogonal ordering measures can be used to support the students' comments, since most of the measures (except Hausdorff distance and orthogonal ordering) are already more sensitive to the borders of the drawing — rotation causes border points to move farther, giving them more weight. Interestingly, the per-user correctness analysis for ordering and rotation shows both cases where borders-only point set performs worse than the full point set for a given user, and cases where the opposite is true. More study is needed to determine if the overall results hold because the borders really are more important, or if the degree of change in the borders is simply representative of the change in the whole drawing. This could be tested by comparing drawings where the border is largely unchanged but the interior is drastically different and those where the border is greatly changed but the interior is not. Additional study is also needed to determine if the importance of the borders is a general principle common to many observers or if it is more user-specific.

- The difficulty of the difference part suggests that the amount of difference between the drawings that is considered reasonable varies greatly with the task — when the user simply needs to recognize the graph as familiar, the perimeter of the drawing and the position and shape of few key features are the most important. On the other hand, when trying to find a specific small change, the drawings need to look very much alike or else the user needs some other cues (change in color, more distinctive vertex names, etc.) in order to highlight the change.

- The lack of success with using the times from the difference task to evaluate the measures with respect to the magnitude criterion means that more study is needed to find a way to get data for this.

The students' responses on the questionnaire suggest several possible directions for future investigation.

- Large vertices are identified as being especially important, which could lead to a scheme in which changes in the position and size of large vertices are weighted more heavily than other vertices.

- Another major focus was clusters of vertices — both the presence of clusters in general, and the presence of specific shapes such as chains and zigzags. The relatively poor showing of the cluster-based measures indicates that they are not making use of clusters in the right way. The fact that the students reported looking for specific shapes suggests an approach related to the drawing algorithms of Dengler, Friedell, and Marks [51] and Ryall, Marks, and Shieber [136]. These algorithms try to produce drawings which employ effective perceptual organization by identifying Visual Organization Features (VOFs) used by human graphic designers. VOFs include horizontal and vertical alignment of vertices, particular shapes such as "T" shapes, and symmetrically placed groups of vertices. VOFs can also be used not to guide the creation of drawings from scratch, but to identify features in an existing drawing that may be important because they adhere to a particular design principle. This is related to the work of Dengler and Cowan [52] on semantic attributes that humans attach to drawings based on the layout — for example, symmetrically placed nodes are interpreted as having common properties. A similarity measure could then measure how well those structures are preserved, and an interactive graph drawing algorithm could focus on preserving the structures.

Chapter 5

A Graph Drawing and Translation Service on the World Wide Web

5.1 Introduction

The visualization of information structured as a graph or network is an emerging trend in numerous fields, including software engineering [50, 76, 77, 100, 101, 129, 134], WWW-navigation [89, 117, 118], program debugging [98, 121], database design [49, 68, 87], and visual programming languages [139]. The problem of drawing graphs has therefore received a lot of attention from researchers, and new algorithms are being developed continually; for background and an extensive survey of research in graph drawing see [54]. By making implementations of graph drawing algorithms available on the WWW, we can enable both practitioners and researchers to take advantage of the latest technological innovations in graph drawing. This, however, also requires tackling the problem of the over-abundance of formats for describing graphs and drawings. While there are efforts in this direction [91], there is still no single universally-accepted format. Therefore, researchers typically define their own formats when implementing an algorithm. Because a user cannot be expected to know the format used by each and every implementation, it is convenient to have translators that can convert the descriptions of graphs and drawings from one format to another. This will allow users to employ a large number of algorithms while knowing only a few formats.

We envision a graph drawing and translation service on the WWW that offers two kinds of services:

- a *drawing service* for constructing a drawing of a graph given by the user, using an algorithm chosen by her, and
- a *translation service* for translating the description of a graph/drawing from one format to

Previously published as S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 45–52. Springer-Verlag, 1997. Figures 5.6 and 5.7 have been updated with current statistics.

another.

Such a service would benefit both practitioners and researchers. From a practitioner's viewpoint, she gets a central facility from where she can "shop around" for an algorithm appropriate for her application. A researcher can use the service for prototyping, by combining available algorithmic components into new algorithms. In addition, both practitioners and researchers can use this service just for translation purposes. Potential uses of this service include:

- drawing graphs from user applications,
- comparing the results of graph drawing algorithms,
- development of new algorithms,
- translating between formats used to describe graphs and their drawings,
- creating a database of graphs occurring in user applications,
- performing demonstrations in educational settings, and
- providing an environment where users can "try before they buy" algorithms which are available for free download or from commercial vendors.

As a first step towards realizing our goal, we have developed a prototype graph drawing and translation service on the WWW.[1] This service consists of one or more Graph Drawing Servers with which the user can interact through an HTML form or a Java-based graph editor embedded in a Web page. In a typical scenario, the user provides the service with the input graph and selects the format of the input, the type of service desired (drawing/translation), the format of the output, and the drawing algorithm (if the graph is to be drawn). A server receives the request, performs the desired service, and sends back the result — a drawing or a translation — to the user. This service can be used to visualize a wide variety of graphs including entity-relationship diagrams, function call graphs, class hierarchies of an object-oriented program or database, and PERT charts; as the number of supported algorithms grows so will the classes of graphs for which it is useful.

Each of the servers comprising the service maintains information about the drawing algorithms and translations supported by other servers and is capable of forwarding requests it cannot satisfy to another server, transparently to the user. This feature is particularly useful for providing access to algorithms which run on different platforms — the user need only contact one of the servers and the request will be handled as if the original server could satisfy it directly. The communication protocol used between two servers is identical to that used between client and server; from a server's point of view there is no distinction between a request sent from a client and one forwarded from another server.

The Graph Drawing Server also serves as a first prototype and proof-of-concept of the even broader vision of *GeomNet* [9], a system for performing distributed geometric computing over the Internet. It consists of a family of *geometric computing servers* that execute a variety of geometric

[1] `http://www.cs.brown.edu/cgc/graphserver/`

algorithms on behalf of remote clients, which can be either users interacting through a Web browser interface, or application programs connecting directly through sockets. A number of services can be provided through the GeomNet framework, including algorithm execution, algorithm animation [8], consistency checking of topological and geometric structures, and experimental study and comparison of algorithms.

This paper is organized as follows. In Section 5.2, we describe previous work related to our service. In Section 5.3, we present the software architecture of our prototype service. In Section 5.4 and Section 5.5, respectively, we describe the graph and drawing description formats and the drawing algorithms supported by the service. In Section 5.6, we give some example interactions between users and the service. In Section 5.7, we provide usage-statistics on the service. Finally, in Section 5.8, we describe future work.

5.2 Related Work

Researchers have recognized the need of making implementations of graph drawing algorithms available on the Internet. GraphEd [90] and its successor Graphlet[2] are graph editors which include many graph layout algorithms. AGD[3] provides a C++ library of efficient drawing algorithms. GDToolkit[4] provides both a library of graph data structures and algorithm implementations and a mechanism for plug–and–play use of standard algorithms through either a graph editor or a batch application. A bibliography of additional graph drawing resources and tools is maintained by Roberto Tamassia.[5] Another system with some similarities to the Graph Drawing Server is Diagram Server [60], a network server for client applications that use drawings of graphs. It offers its clients an extensive set of facilities to represent and manage diagrams through a multi-windowing environment. However, Diagram Server, like the other implementations mentioned, are all source code packages or executables which must be downloaded and installed locally.

To the best of our knowledge, there is no (graph) translation service on the WWW other than our service and there are a few graph drawing services which are available, but they have a narrower goal than ours. Stephen North has designed a service[6] that accepts a graph sent to it by email and returns, also by email, a drawing constructed using *dot* [105]. This service also maintains a database of the graphs from various applications [119]. GraphDraw [66] is a Web-based system that allows users to draw graphs using force-directed [75] and hierarchical drawing methods [135]. This system is implemented as a Java applet which runs on a user's machine, and therefore requires high computational resources at a user's machine, especially for the force-directed methods. GraphPack [106] is a system based on `cgi-bin` scripts that supports several graph drawing algorithms. A graph

[2] `http://www.uni-passau.de/Graphlet`

[3] `http://www.mpi-sb.mpg.de/AGD/`

[4] `http://www.dia.uniroma3.it/ gdt/`

[5] `http://www.cs.brown.edu/people/rt/gd.html`

[6] `http://www.research.att.com/dist/drawdag/mail_server`

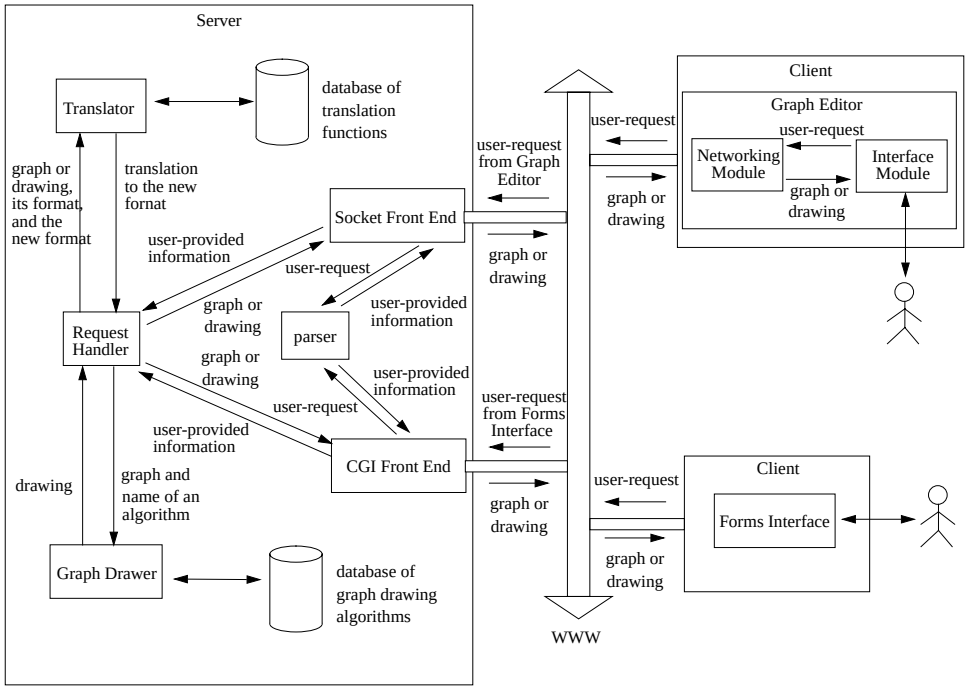


Figure 5.1: The software architecture of our graph drawing and translation service.

drawing demo applet written at Sun Microsystems illustrates the use of Java and has rather limited capabilities [151].

5.3 Software Architecture of our Prototype Service

We believe that a graph drawing and translation service should satisfy the following requirements:

- The input/output format used for a graph should be as independent as possible from the algorithm used for drawing it.
- It should be easy to add new formats and algorithms to the service.
- The user interface should be simple and intuitive.
- The architecture should be client-server based with the assumption that the client machine and the network may be slow, whereas the server is reasonably fast. This assumption is important for a network such as the WWW.

We now describe the software architecture of our prototype service (see Figure 5.1). The service has seven modules: the *Forms Interface*, the *Graph Editor*, the *Socket Front End*, the *CGI Front End*, the *Request Handler*, the *Parser*, the *Translator*, and the *Graph Drawer*. The Forms interface and the Graph Editor run on the user's machine; the other modules all run on the server.

5.3.1 Client Side Modules

The client machine is responsible for maintaining the interface presented to the user. There are currently two types of interfaces supported by the service: the forms interface and the graph editor.

Forms Interface. The forms interface is simply an HTML page with a form that the user can fill to place her request. Figure 5.4 shows a request made using a form. No code is needed to handle server communications or manage the form, as this is done by the user's Web browser. When the user submits the form, the browser sends a GET or POST request to the server's CGI Front End (described below).

Graph Editor. The graph editor is implemented as a Java applet and consists of two modules, the *interface module* and the *networking module*. The interface module is responsible for creating and managing the graph editor window, supporting basic graph editing operations such as insertion, deletion, and movement of vertices, edges, and edge bends, and enabling the user to interactively create a graph to be sent to the server. Users can also load a graph from a Web-accessible URL. Users can also save a graph or drawing by first converting it to the desired format by using the translation service and then saving it in text format using the "Save As" feature of the Web-browser. This approach is necessary because of security restrictions that prevent a Java applet from writing directly to a local disk.

The networking module is responsible for communicating with the server — it encodes a user-request and sends the message to the server, provides the interface with updates on the current status of the request, receives the response, and ensures a response will be received even if the server fails.

The networking module communicates with the Socket Front End of the server (described below).

5.3.2 Server Side Modules

We now describe the modules that run on the server. All of the server modules are written in Java, with the exception of a small C component to handle the integration of non-Java algorithms and translation filters. As a result the server is easily portable to a variety of systems.

Request Processor. The request processor is the central component of the service. It receives a request from the Socket and CGI Front Ends and satisfies the request by invoking the Translator and the Graph Drawer. If it is a translation request, then it simply asks the translator to carry out the desired translation. Satisfying a drawing request is more complicated, and consists of executing the following steps:

- consult the Graph Drawer to determine the input and output formats of the the drawing algorithm requested by the user,
- use the Translator to convert the input graph to the input format,

- construct a drawing of the graph by using the Graph Drawer,
- use the Translator again to convert the drawing to the output format requested by the user, and finally,
- send the output to the front end — Socket or CGI — from which the request was received.

The Request Processor "keeps an eye on" on the Graph Drawer and Translator by terminating any processes that do not finish within a set period of time. This is to prevent bugs in a drawing algorithm or translation filter from causing an infinite loop that will tie up server resources indefinitely.

CGI Front End. Communication with the CGI Front End follows the HTTP protocol: a GET or POST request referencing the front end's URL is sent to the Web server on the server's machine. In GET requests the submitted information is appended to the script's URL, whereas POST requests contain the information in the body of the HTTP message. Because even small graphs can lead to very long URLs using the GET method, POST is the preferred method of communication with the CGI Front End. Once the request is received, the Web server automatically invokes the CGI Front End to process the request. When done, the front end returns the result to the Web server, which forwards it back to the client. Anything obeying this protocol may communicate with the CGI Front End, but it is usually invoked by a user submitting a form using the Forms Interface.

The CGI Front End executes the following steps to satisfy a request:

- parse the form data by using the Parser to extract the information provided by the user in the form,
- send the extracted information to the Request Processor, and
- send the output received from the Request Processor to the client.

Socket Front End. The Socket Front End would typically be used by any non-Forms Interface client, though currently is only used by the Graph Editor. It handles communications over a socket using a custom protocol with messages consisting of keyword/value pairs. We elected to use a custom protocol because when the Graph Drawing Server was first developed there was very little Java-based support for distributed computing. Furthermore, our protocol is very easy to learn and use, does not require the user to obtain additional software from a third party, is not limited to a particular programming language, machine architecture, or operating system, and can be used by applets — features which are not always available in systems built on standards like OMG's CORBA or Microsoft's DCOM. Our future plans include further investigating the newly emerging distributed computing options — the RMI (remote method invocation) package added to Java 1.1 meets many of the requirements listed above (failing only in that client programs must be written in Java). CORBA is also becoming a more attractive choice, since Java 1.2 is slated to include CORBA support.

The Socket Front End waits in a loop, listening for messages coming in on a specified port. On receiving a message from a client, it creates a socket and starts a new thread, called the *connection*

thread, to listen on that socket and handle any further messages from that client. For each message from the client, the the connection thread initiates a separate thread, called the *computation* thread, which performs the necessary actions for satisfying the request made in the message. In particular, if it is a drawing/translation request, the computation thread executes the following steps:

- use the Parser to extract the information contained in the body of the message,
- send the extracted information to the Request Processor,
- receive the output from the Request Processor,
- send a reply — the output or an error message — to the client.

The Socket Front End also keeps track of the status of the requests currently being handled, allowing it to respond to status-seeking messages from the clients.

Parser. The parser is invoked by the Socket and CGI front ends to parse the request string sent by the user. It extracts the following information: the graph to be drawn or translated, the format of the input, the type of service desired (drawing/translation), the format of the output, and the drawing algorithm to use (if the graph is to be drawn). The parsed information is then sent back to the appropriate front end.

Translator. Each graph drawing algorithm supported by the service defines its own input/output format which may be different from the format of the graph given by the user. Therefore, there may be a need for translating from one format to another on drawing requests. In addition, the user may explicitly request a translation. The Translator takes as input a graph or a drawing and the names of the current format and the format to which translation is to be done, and performs the desired translation.

The Translator maintains a database of translation functions available locally and through other servers and uses a directed graph, called the *translation graph* (Figure 5.2) to perform the translations. The vertices and edges of this graph are the formats and translation functions, respectively, supported by the Translator. To satisfy a translation request, the Translator constructs a translation sequence that gives the desired translation. Each translator sequence corresponds to a directed path in the translation graph. For example, a translation from the Edges format to the Malf-Input format is done by using the sequence Edges→Parenthetic→Malf-Input. Requests are forwarded to remote servers as necessary.

The problem of translation is a difficult one, since while every graph format encodes the same combinatorial information about the vertices and edges, many formats also include additional attributes (such as edge/vertex labels, colors, and coordinates) or topological information (such as the embedding of the graph). If two formats do not encode the same information, translating from one to the other will cause some information to be discarded, or require some attributes to be assigned what one hopes are reasonable default values.

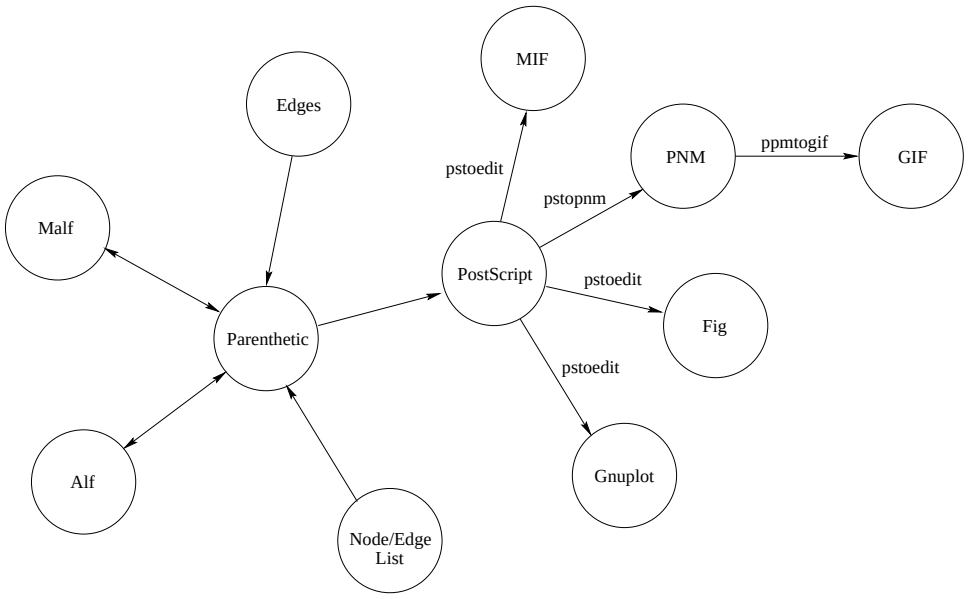


Figure 5.2: The translation graph used for performing translations. Unidirectional and bidirectional edges denote availability of functions for translation in only one and both directions, respectively. The edges that are not labeled denote translation functions implemented by us, and the remaining edges are labeled by the existing programs that perform the corresponding translations.

Currently the translation sequences employed are hard-coded to avoid using a less expressive format as an intermediate step; since Parenthetic allows arbitrary tags to be added it can express information contained in any of the other text formats and so is used as an intermediate step for many translations. This does not, however, solve the problem since a user may supply a Parenthetic-format graph to an algorithm expecting Malf-Input, in which case many of the attributes of the original graph will be lost.

One solution to this problem is to use a *difference file* during the translation from one format to another. This difference file contains information about attributes discarded as a result of translation as well as those which have been added. During a translation, the translator consults the difference file to see if a given attribute had a value in the original graph before assigning a default value, and augments the difference file with information about attributes lost during the translation step. Our service currently uses a simpler (client-side) scheme in which vertices and edges in the drawing returned by the server are matched to the corresponding objects in the graph sent to the server, and those attributes which should be preserved are simply copied. Vertices are matched by name and edges are matched by the names of their endpoint vertices. This scheme works for graphs with unique vertex names and no multiple edges; we plan to implement the more general (server-side) scheme of using difference files in the future.

Our prototype service currently supports the following input formats: Parenthetic, Edges, Node/Edge List, and Malf, and the following output formats: Parenthetic, Postscript, GIF, Gnuplot,

MIF, Fig, Malf, and Summary. Section 5.4 describes the formats in detail.

Graph Drawer. The graph drawer accepts as input the name of a drawing algorithm and a graph described in the algorithm's input format, and constructs a drawing of the graph using the algorithm.

The Graph Drawer maintains a database of drawing algorithms. Like the Translator, this database consists of algorithms supported both locally and by remote servers, and requests are forwarded by the Graph Drawer as needed. Currently, the database consists of the following algorithms: Giotto, Pair, Planarizer, Bend-Stretch, Column, and Sugiyama. Section 5.5 describes the algorithms in detail.

Our prototype service does not require any special hardware or software on the client machine other than a commonly available Web browser. The forms interface requires a browser such as Lynx or Netscape that supports HTML forms. The graph editor requires a browser that supports the Java 1.x API, such as Netscape 2.0 or higher for most platforms. Java support is being added to more browsers so that this requirement is becoming less restrictive.

The server should be reasonably fast with some kind of support for multi-threading, and with sufficient memory for storing the users' graphs and drawings.

Our service satisfies the requirements stated earlier in this section as follows:

- The user is free to specify any input/output format independent of the algorithm requested by her. This decoupling is made possible by the presence of the translator.
- The Graph Drawing Server provides a simple framework for adding new formats and algorithms. Adding a new format requires only providing a translator between the new format and some existing format; since the Translator can put together sequences of translation functions anything that could be translated to or from the chosen existing format can now be translated to or from the new format. Adding a new algorithm requires updating the database of algorithms in the Graph Drawer and, if it defines a new format, adding that format to the Translator.
- Both the forms interface and the graph editor are simple and intuitive.
- All the computationally intensive work is done by the server. Hence, the service can be used by clients with limited resources as well.

5.4 Graph and Drawing Description Formats

We now describe the graph and drawing description formats currently supported by our service. We currently support the following input formats: Parenthetic, Edges, Node/Edge List, and Malf, and the following output formats: Parenthetic, Postscript, GIF, Gnuplot, MIF, Fig, Malf, and Summary.

Parenthetic: The Parenthetic format [137] consists of nested lists of keyword-value pairs enclosed by parentheses. This format is easy for both users to read and computers to parse, and

follows the same design principles as several other popular formats such as SGML, HTML, and VRML; however, it produces larger files than the other supported text formats and it is not easy to edit Parenthetic files directly because vertex and edge IDs must be consecutive — so changes require renumbering a potentially large number of objects. The great advantage of the Parenthetic format is its flexibility: it provides a wide variety of built-in keywords and allows users to define their own keywords. There is no pre-specified order on the occurrences of keywords, and keyword-value pairs can span multiple lines. Because of this superior expressive power — it is the only currently-supported format to allow arbitrary attributes to be added — it is used as an intermediate format for several translations (see Figure 5.2). A further discussion of the Parenthetic format is given in the appendix.

Edges: Edges is a very simple format consisting of a sequence of lines, with each line describing an edge. There is no explicit declaration for vertices; instead a vertex are declared by listing it as part of an edge, and the same vertex name occurring in multiple edges refers to the same vertex. Each line has the following syntax (where, SrcVertex and DestVertex are ASCII strings):

SrcVertex $[<, >, -]$ DestVertex

Here, $[<, >, -]$ denotes either zero or one and only one occurrence of one of $<$, $>$, and $-$. SrcVertex $>$ ($<$) DestVertex denotes a directed edge from (to) SrcVertex to (from) DestVertex. SrcVertex DestVertex and SrcVertex $-$ DestVertex both indicate an undirected edge between SrcVertex and DestVertex. Edges is very compact and easy to edit, but is limited to describing only graphs (coordinates cannot be specified) and has no facilities for adding attributes to edges or vertices. This format is described in a greater detail in the appendix.

Node/Edge List: This format is a simplified version of the format supported by a system from Tom Sawyer Software. Like Edges, it is compact and easy to edit but can describe only graphs and has no support for object attributes.

A file in the Node/Edge List format is divided into the following sections: // nodes, // edges, // named edges, // directed edges, // directed named edges, and // named directed edges.

The // nodes section lists the graph's vertices, one per line. The // edges and // directed edges sections list the graph's edges one by one, with each end vertex of an edge listed on a separate line. The // named edges, // directed named edges, and // named directed edges sections list the edges with names one by one, with the edge name and the end vertices of an edge listed on separate lines. The // nodes section must be listed first, but the other sections need not be present and may be given in any order. The Node/Edge List format is described in a greater detail in the appendix.

Malf: This format is a modification of the Alf format supported by the the *Automatic Layout Facility* [14] of Diagram Server [58, 60], and has two versions: Malf-Input that can describe graphs only, and Malf-Output that can describe both a graph and a drawing. Both Malf formats are

compact — typically under 5 KB for a graph with a combined total of 250 edges and vertices — but again do not allow attributes (other than position) and rely on consecutive IDs, making editing difficult.

A file in the Malf format consists of a vertex list, followed by an edge list. The two lists are separated by a line consisting of a single '#' character.

The vertex list consists of a sequence of vertices, one per line. In the Malf-Input format, each vertex has a unique integer identity, and an optional name. The Malf-Output format also describes the position and size of each vertex in a drawing.

The edge list consists of a sequence of edges, one per line. In the Malf-Input format, each edge has a unique integer identity, a direction, and the vertex IDs of the source and destination vertices of the edge. The Malf-Output format adds a list of the bends of the edge in a drawing.

The Malf format is described in a greater detail in the appendix.

Gnuplot, MIF, and Fig: These are the formats supported by `gnuplot`, FrameMaker and `xfig`, respectively. Of these formats, MIF and Fig are the most useful because they can be edited to customize the look of the drawing; Gnuplot has a number of drawbacks, including the inability to display vertex labels. These formats produce files that are generally several times larger than any of the text formats (Parenthetic, Edges, Node/Edge List, Malf).

Postscript, GIF: These formats cannot be easily modified after creation, but are suitable for inclusion in documents. Postscript is particularly good for LaTeX documents; GIF is useful for HTML pages. These formats also produce files significantly larger than the text formats.

Summary: Summary is not a true output format; instead an HTML page is returned with a small GIF image of the graph along with links to obtain the full-size image or other formats. It also provides a link to the input graph. Since very little additional time is needed to obtain a full-size GIF and only a translation is necessary to see the output in another format, Summary provides an efficient way for a user to see the output in various forms. Summary also provides some statistics about the output, such as the size and aspect ratio of the drawing and the number of bends. Other information, such as the time taken to construct the drawing, will be added in the future.

Typical file sizes for several of the more common output formats are shown in Figure 5.3.

5.5 Drawing Algorithms

Our service supports a number of algorithms allowing users to choose the one appropriate for their applications and to compare and demonstrate them for research and educational purposes. The supported algorithms determine what types are graphs are most suited for drawing with the service — currently the focus is on orthogonal drawing algorithms, which are perform well on graphs from a

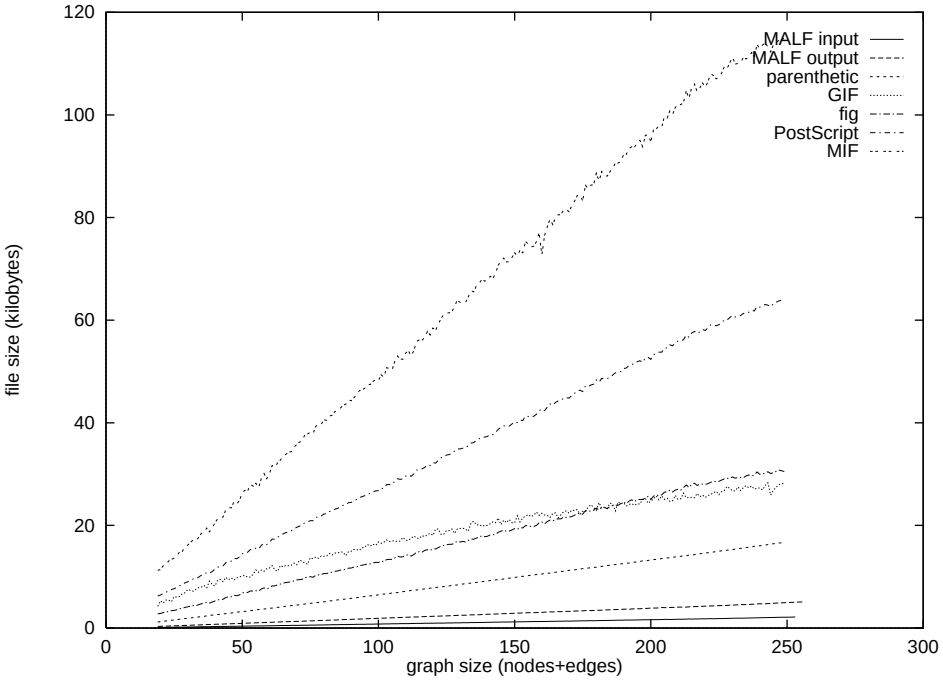


Figure 5.3: Average file size for various output formats. (The legend lists the formats according to increasing size for graphs of size 250.) The Parenthetic files used contain only the standard keywords, so those sizes represent the typical minimum file size.

variety of domains including software engineering and VLSI design. One hierarchical algorithm, well suited for graphs such as program call graphs which have a top-to-bottom order, is also available. As more algorithms are added to the server the range of graphs that can be drawn satisfactorily will increase.

Many of these algorithms are implemented within the framework of the *Automatic Layout Facility* [14] (ALF) of Diagram Server [58, 60]. The Automatic Layout Facility of Diagram Server consists of a large modular library of graph drawing algorithms and provides a tool which, given the requirements of an application, selects suitable algorithms for such requirements. It also allows the users to customize new algorithms by defining an *algorithmic path* describing the sequence of steps and intermediate representations (e.g., planar embedding, orthogonal shape, visibility representation) produced by the algorithm.

In the rest of this section, N and M denotes the number of vertices and edges, respectively, of the input graph, and C denotes the number of crossings in the drawing constructed.

The following algorithms are currently supported by our service:

Giotto: Giotto [146] is a general-purpose drawing algorithm that constructs orthogonal drawings of graphs. Given a graph, it first converts it into a planar graph by replacing edge-crossings by fictitious vertices (*planarization step*), then computes an *orthogonal representation*—a symbolic

representation of a drawing using angles—by using the bend-minimization algorithm of Tamassia [145] (*orthogonalization step*), and finally constructs a drawing from this orthogonal representation by assigning coordinates to the vertices and edges of the graph (*compaction step*). Giotto is implemented using ALF, and has the time-complexity of $O((N+C)^2 \log(N+C))$; see [57] for details.

We also provide another version of Giotto, called Giotto-With-Labels, which draws each vertex as an expanded box large enough to fit its label.

Bend-Stretch: Bend-Stretch [57] uses the same three steps — planarization, orthogonalization, and compaction — as Giotto, and differs only in the method used in the orthogonalization step, namely, it adopts the "bend-stretching" heuristic of Tamassia and Tollis [148] that only guarantees a constant number of bends on each edge but runs in linear time. The planarization step uses an incremental method and has time complexity $O(N+C)^2 \log(N+C)$, and therefore, Bend-Stretch has time complexity $O(N+C)^2 \log(N+C)$; for details see [57]. Bend-Stretch is also implemented using ALF.

Pair: Pair [57] uses a different approach from Giotto and Bend-Stretch. It is an extension of the orthogonal drawing algorithm for degree-4 graphs by Papakostas and Tollis [124] to graphs of arbitrary vertex degree. Given a graph, it first computes an st-numbering of its vertices and then employs this numbering to optimize the number of bends, row and columns used in the drawing. Pair has the time-complexity of $O((N+M)\log(N+M))$.

Column: Column [57] is similar to Pair and differs from it only in the method used to optimize the number of bends, rows and columns used in the drawing once an st-numbering has been computed. The method used is the one of Biedl and Kant [17]. Column is implemented using ALF, and has the time-complexity of $O(N+M)$; see [57] for details.

Sugiyama: Sugiyama is our implementation of the algorithm of Sugiyama, Tagawa, and Toda [143]. It constructs a hierarchical drawing of a directed graph in the following three steps: first, in a *layering* step, it assigns vertices to horizontal layers, next, in a *crossing-minimization* step, it permutes the vertices within the same layer to reduce edge-crossings, and finally, in a *bend-reduction* step, it readjusts the position of vertices within each layer to reduce edge-bends. Sugiyama is implemented using ALF; see [55] for details (where it is called Algorithm *layers*).

Planarizer: Planarizer is actually not a drawing algorithm. It is the planarization step of Giotto, and constructs a planar embedding of the input graph by replacing edge-crossings with fictitious vertices. It has time-complexity $O((N+C)^2 \log(N+C))$.

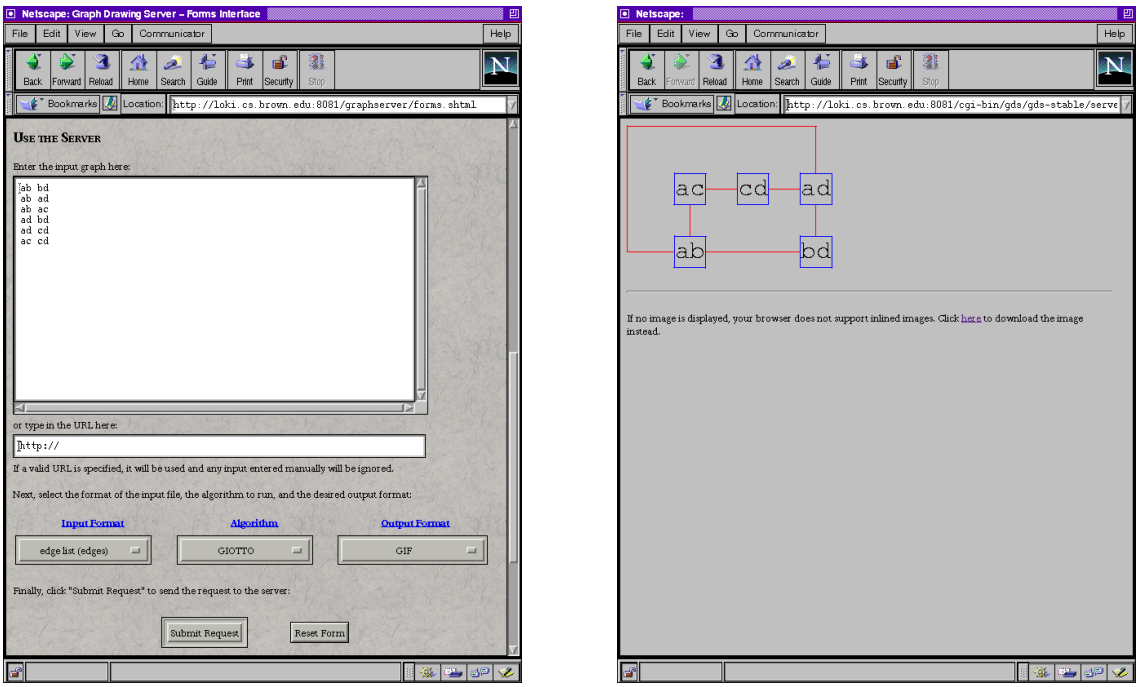
5.6 Using our Service

We now give two scenarios to show how a user interacts with our service and how the service satisfies a user request.

Scenario 1. Using the Forms Interface (see Figure 5.4).

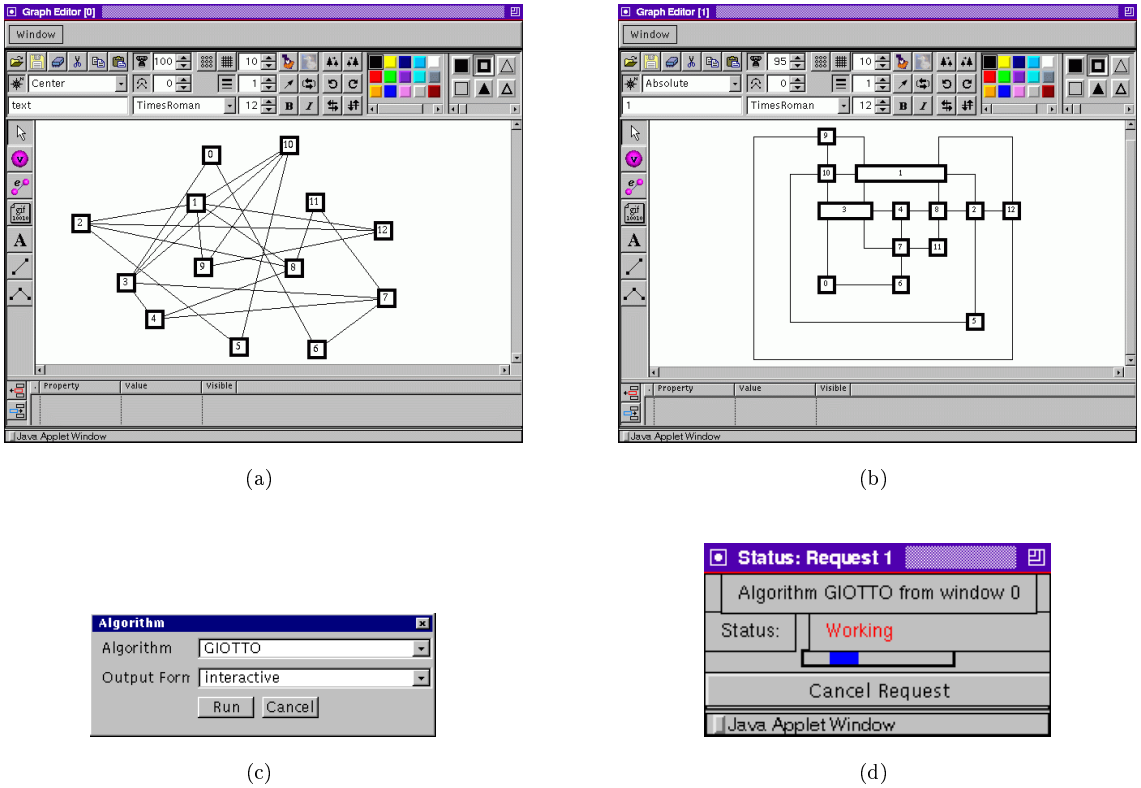
Suppose the user wants to draw a graph described in the Edges format using the Giotto algorithm, and wants the drawing to be displayed as a GIF. She loads the service's home page into her browser, clicks on the *forms interface* link, and gets an HTML form. She can specify a graph in one of two ways — either by giving the URL of the graph or by typing the description of the graph in the form itself. In Figure 5.4(a) she chooses the latter option and enters the graph (in Edges input format) into the text area provided in the form. She then selects the appropriate menu item to specify that the input format is Edges, the drawing algorithm is Giotto, and the output format is GIF, and submits the form to the server. Figure 5.4(a) shows a part of this user-filled form.

The Web-server receives the form and executes the CGI front end of the service. The CGI front end first invokes to extract the input graph, the type of service requested (DRAWING), the names of the input and output formats (Edges and GIF, respectively), and the drawing algorithm to use (Giotto) from the form. This information is then passed on to the request handler which first consults the graph drawer to determine the graph format required by the drawing algorithm (Giotto uses Malf) and then calls the translator to convert the graph from Edges to Malf. The graph description in Edges format is then sent to the graph drawer, which uses Giotto to construct a drawing of the graph.



(a) (b)

Figure 5.4: Scenario 1: Using the forms interface to draw a graph described in Edges format using Giotto, where the drawing is to be displayed as a GIF. (a) Part of the user-filled form; (b) Drawing in GIF format returned to the user.



(a) (b)

(c) (d)

Figure 5.5: Scenario 2: Using the graph-editor to draw a graph with Giotto, where the drawing is to be displayed in an editor window. (a) The interface of the editor and the input graph; (b) Drawing returned by the server; (c) Dialog box for algorithm and output format selection; (d) Status window displayed while the request is being handled by the server.

The drawing is returned to the request processor in Malf, the output format of Giotto, so it must once again invoke the translator to convert the drawing to GIF. The final drawing is saved to a file on the server and the URL of this file is returned to the client.

Back on the client, the user's browser receives the URL sent by the server and automatically loads the file, displaying it on the screen as shown in Figure 5.4(b) (assuming the browser is configured to automatically display GIF files; otherwise the user would be prompted to download the file).

Scenario 2. Using the Graph Editor (see Figure 5.5).

As before, the user loads the service's home page into her browser. This time she first clicks on the *interactive applet interface* and then on the *start client* link. This causes a graph editor window to appear. Figure 5.5(a) shows a graph editor window with a sample graph constructed interactively by the user. To obtain a drawing of the graph the user must specify the algorithm and the output format by making the appropriate selections on the pull-down menus at the top of the window. The example shows the algorithm Giotto and the output format "interactive". (Selecting "interactive" as

the output format will cause the drawing to be displayed in another editor window; all of the other formats supported by the forms interface, such as Parenthetic and GIF, are also supported by this interface and will be displayed in a new browser window.) Once the algorithm and output format have been selected, the user clicks on the "Run" button to send the request.

Sending the request causes a small status window to pop up, providing the user with information about the processing of the request and allowing her to cancel a running request (Figure 5.5(c)). The processing of the request on the server side follows the same basic pattern as for the forms interface (parsing, translation, drawing, translation); the differences are that the socket front-end is used for reading and parsing the request and some extra machinery is employed to ensure that responses destined for the same client are sent back in the proper order.

When the request completes, a new graph editor window appears (since the chosen output format was interactive) with the resulting drawing (Figure 5.5(b)), scaled to fit in the window. This drawing is fully editable, and can be modified and resubmitted to the server. It can also be saved, by sending a translation request to the server to convert the graph into some other format (presumably a text format), which is displayed in a browser window and can then be saved using the browser's own "Save As" mechanism.

5.7 Experience with the Service

Our service first became operational in June 1996. Monthly usage statistics are shown in Figure 5.6. Since that time it has been accessed by users around the world, as shown by Figure 5.7. A large fraction of these users are from educational and commercial sites (identified by the `.edu` and `.com` prefixes, respectively, in their Web-addresses). We have ourselves used it to generate drawings for inclusion in documents and presentation slides.

This service has been used by students within our department for studying and improving the algorithms supported by the service, sometimes in creative ways. For example, while developing this service we also designed a small command-line facility for sending drawing or translation requests to the server, to be used for debugging purposes. One student, Jody Fanto, working on improving Giotto, used this facility to design a novel distributed system for conducting experiments on Giotto. His system divides a test-set of graphs into smaller subsets, and for conducting experiments on each subset, spawns a new process that runs on a separate machine. Hence, while the drawing requests are satisfied by the server, the statistical analysis such as determining the area and number of bends of a drawing on the subsets is done on different machines. Another use for the service was in the development and testing of InteractiveGiotto [43], a version of Giotto modified to preserve the user's "mental map" when redrawing a layout after some changes have been made. Using the Graph Drawing Server meant that the graph editor applet could be used to construct test cases and to get immediate graphical feedback, a vast improvement over working with text representations of the graphs.

We also conducted an experimental study on the average response time of the service. Our

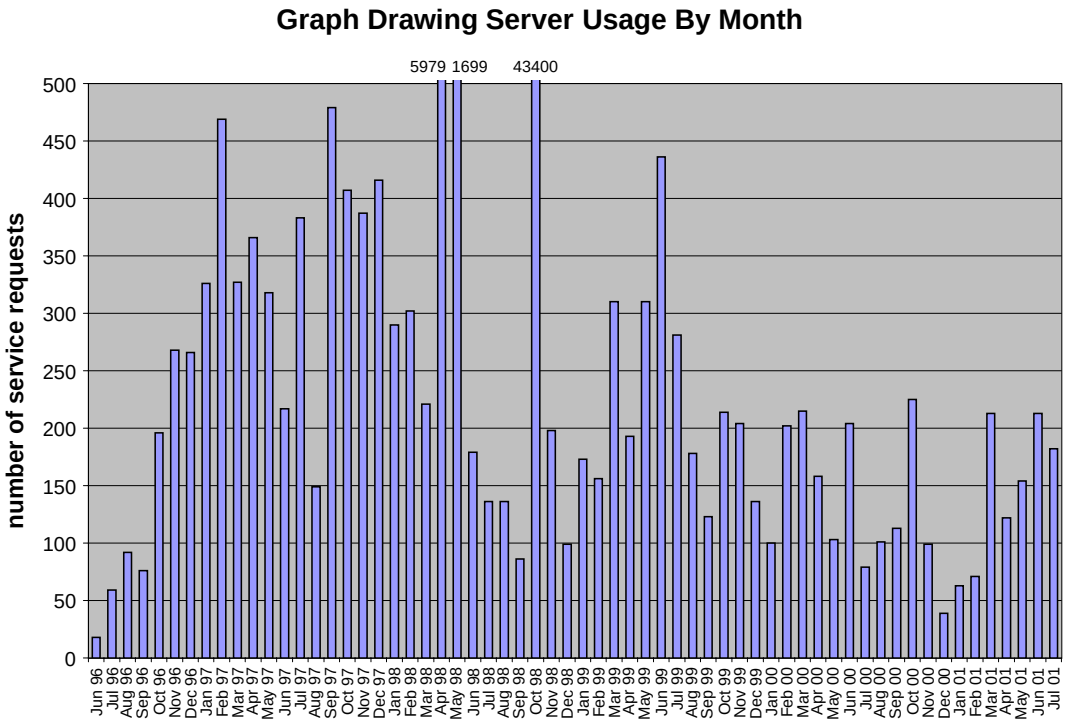


Figure 5.6: Distribution of over 63,000 service requests by month. Only drawing/translation requests are counted; the number of hits on the Graph Drawing Server's web pages is much higher (nearly 200,000 total).

motivation was not only to evaluate the performance of the service, but also to determine the running times of the drawing algorithms and translators supported by the service. These experiments also constitute a first step in our broader goal to provide a facility for conducting experiments on algorithms and translators and collecting statistics on them.

We used the following performance measures for our study:

- The average overall time needed to satisfy a request once it has been received by the server. We did not measure the time from the time it was submitted by a user because that is dependent on extraneous factors such as network traffic.
- The average time needed to satisfy a drawing request.
- The average time needed to satisfy a translation request.

For this study we used a set of 6,950 graphs with vertices and edges in the range 10 to 100 and 10 to 140, respectively. These graphs are a subset of a data set used in other experimental studies [57]; the full set was generated from a core of 112 graphs collected from various software engineering and database applications by using graph updating operations typical in these applications.

We have recently begun storing graphs submitted to the Graph Drawing Server to build up a library of graphs for future experiments. This data set is still quite small, but provides some

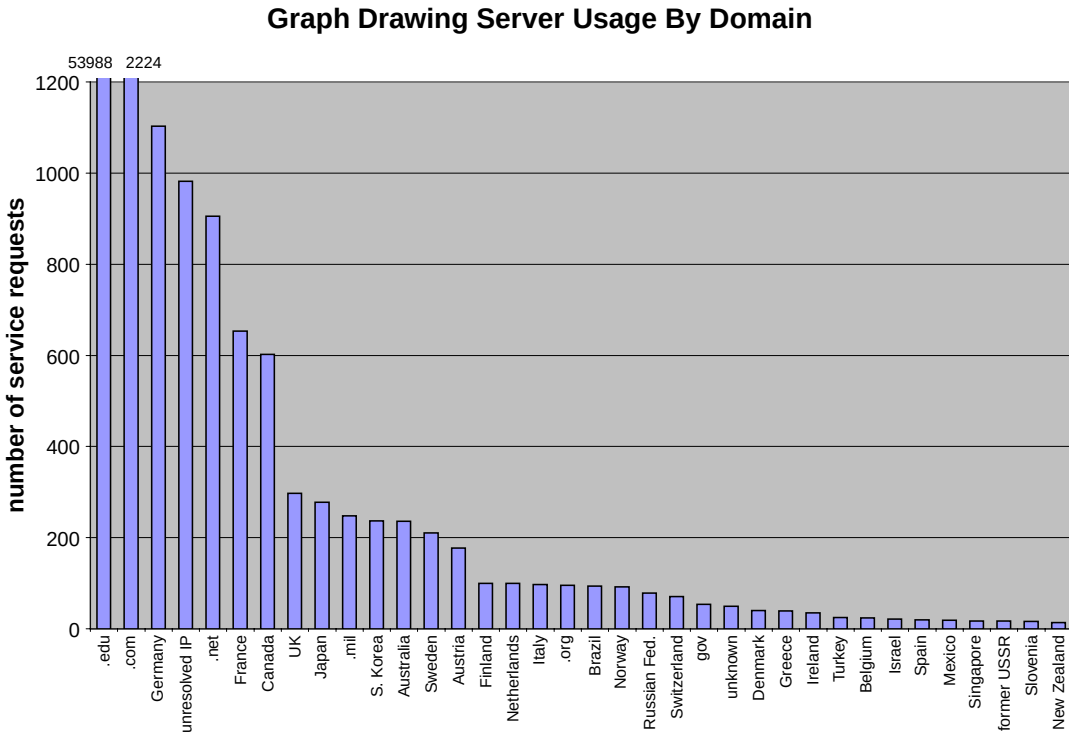


Figure 5.7: Distribution of service requests over various host domains from June 1996–July 2001. Only drawing/translation requests are included.

indication that the graphs used in the experimental study are reasonably "typical" as the processing time for the user graphs agrees with the data obtained from the study. We hope that over time we will be able to collect a set of larger graphs for use in experiments; so far 90% of the graphs submitted for drawing or translation have fewer than 80 vertices and only a handful have more than 100 vertices.

For the experimental study, the input graphs were in the formats Malf and Parenthetic. The output formats were categorized into two groups: *text oriented* formats that consisted of Parenthetic, Malf-Input, and Malf-Output, and the *diagram oriented* formats that consisted of Postscript, GIF, MIF, Gnuplot, Summary, and Fig. We assumed that each output format is equally likely to be requested by a user. Hence, in our study, 33% of the graphs were drawn or translated with text oriented output formats and 67% with diagram oriented formats. We also assumed that each drawing algorithm and explicit translation is equally likely to be requested by a user. Since we support seven drawing algorithms 86% of the requests were for drawing and the remaining for translation. All of the times measured are total elapsed times; they include any time the server spent running other processes. All of the experiments were run on the machine used as the server (an UltraSPARC 1) under normal conditions (though not dedicated solely to the Graph Drawing Server, the machine is not usually heavily loaded).

Figure 5.8 shows that the overall time needed to satisfy a request is in the range 8 to 12 seconds,

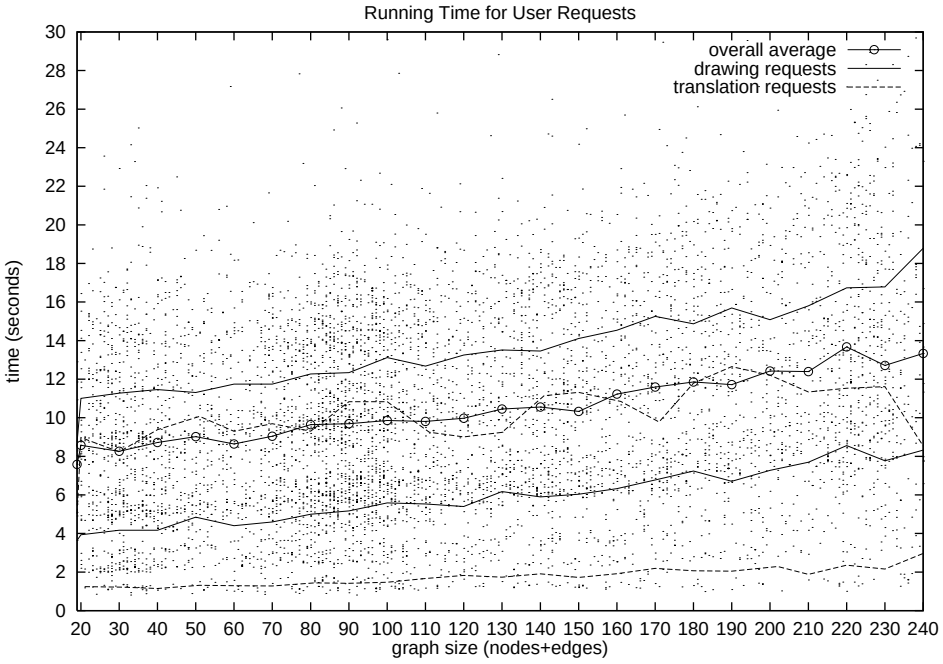


Figure 5.8: Average overall (elapsed) time needed to satisfy a request versus input graph size. The dots represent individual requests. The high and low lines for drawing requests and translation requests are averages for diagram-oriented output formats and text-oriented output formats, respectively.

and increases slowly with the graph size. The maximum time needed for a request was 45 seconds. (The few outlying points are the result of momentary server load.) Note this this is only the time used by the server; it does not include the time needed to send the data between client and server across the network. Our studies indicate that there is a perceptible difference between the average times needed for satisfying requests with the text oriented and diagram oriented formats. We believe that it is because the diagram oriented formats are all generated using Postscript (see Fig. 5.2), often requiring two (or more) translation steps, and because converting from Postscript to another format involves running a Postscript interpreter, a time-consuming task. Statistics such as these can be used to improve translator performance — if a particular multi-step sequence is common, a shortcut filter can be written to do the translation in a single step.

Figure 5.9 shows the time used by only the drawing algorithm step of the request. The average time is between 2 and 6 seconds and increases slowly with the graph size. As shown in the figure, many of the dots are clumped together in bands, illustrating the difference in running times among the supported algorithms.

Figure 5.10 compares the time needed for the translation step(s) of both drawing and translation requests. The average time needed is in the range 4 to 10 seconds and also increases slowly with an increase in graph size. The translation time for drawing requests is higher than for translation requests because typically two translations are needed to satisfy a drawing request — from the

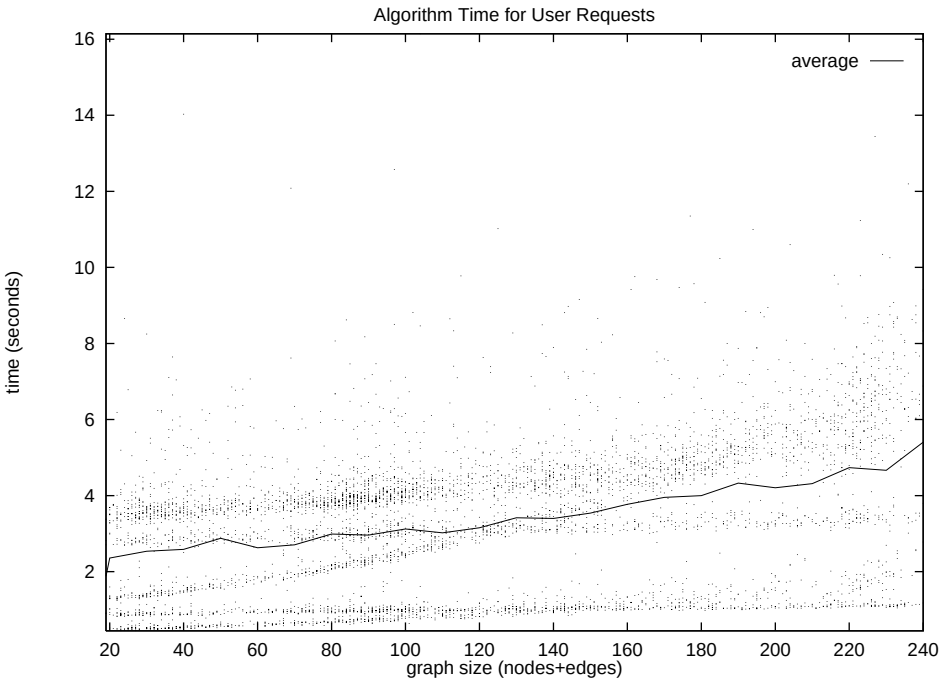


Figure 5.9: Average (elapsed) time needed by the drawing algorithms versus input graph size. The dots represent individual requests. The clumping of the dots into distinct bands illustrates the different running times of the various algorithms — the lower bands correspond to Pair and Planarizer, the middle band Column, and the upper bands Giotto, Bend-Stretch, and Sugiyama; the steeply increasing band is Giotto-With-Labels.

input graph's format to the drawing algorithm's input format and from the output format of the drawing algorithm to the desired output format. The difference is relatively small, however, because of the diagram oriented output — when a diagram oriented format is requested the time needed for this step dominates the total translation time for the drawing request. The lower band of dots in Figure 5.10 shows the translation times with the text-oriented output formats and the upper bands translation times with the diagram-oriented output formats.

5.8 Future Work

There are several directions for further work on the service:

- Improving the translator by adding support for difference files.
- Adding support for new formats such as GML [91].
- Adding new algorithms. We plan to start with implementations already available freely on the Web.
- Adding a programmer's interface, where a small Java or C++ library is provided to handle communications with the server, allowing users to make remote process calls to the server.

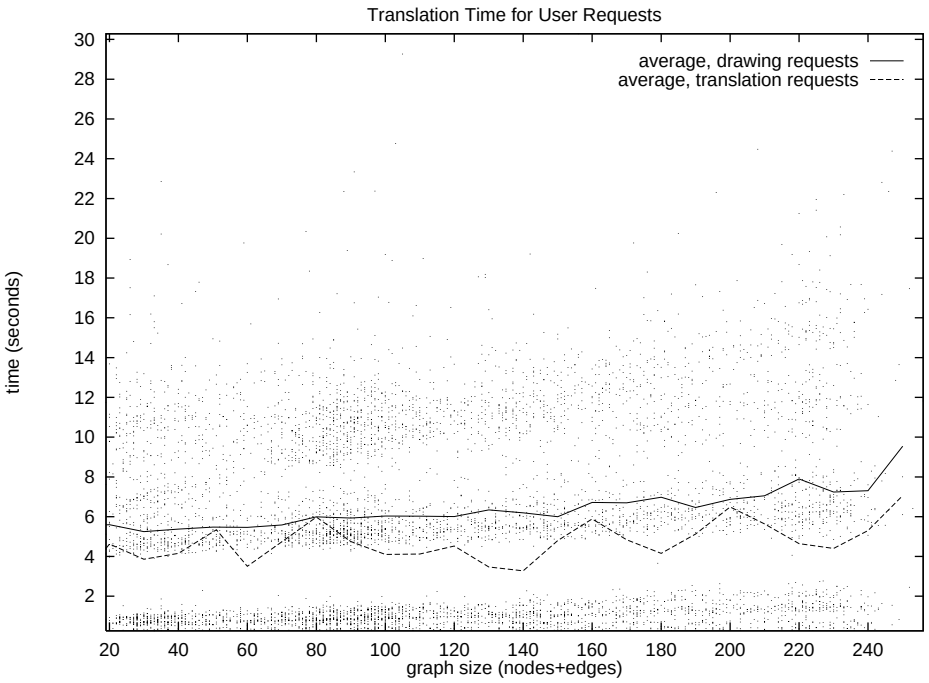


Figure 5.10: Average (elapsed) time needed by translations, both explicitly requested and implicitly done during drawing requests, versus input graph size. The dots represent individual requests. The lower band of dots (under 2 seconds) represents translation times for text-oriented output formats and the upper bands (over 4 seconds) represent translation times for diagram-oriented output formats.

- Adding support for incremental graph drawing.
- Providing a service (accessible through the Web) through which the users can conduct experimental studies on graph drawing algorithms.
- Exploring the use of distributed computing technologies (eg RMI or CORBA) for client-server communication.

Acknowledgments

We would like to thank Giuseppe Di Battista for making the Automatic Layout Facility of Diagram Server available to us, and Jody Fanto for allowing us to use his system to collect statistics on the service.

5.9 Appendix: Graph and Drawing Description Formats

In this appendix, we describe some of the graph and drawing description formats supported by the service in a greater detail. The formats we describe are: the Parenthetic format, Edges, Node/Edge List, and Malf.

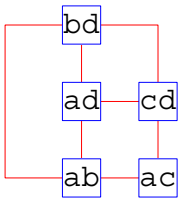


Figure 5.11: A drawing constructed by Giotto.

An *ASCII string* is a string consisting of ASCII characters. λ denotes an empty word. $[a_1, a_2, \ldots, a_n]$ denotes the occurrence of either none or one and only one of $a_1, a_2, \ldots, a_n$. $\{a_1, a_2, \ldots, a_n\}$ denotes the occurrence of $a_1, a_2, \ldots, a_n$ in any order. $\backslash n$ denotes the new-line character. $(a)^{n+}$ denotes n or more occurrences of a. $a|b$ denotes a choice between a and b. $(a)*$ denotes zero or more occurrences of a.

Parenthetic: The Parenthetic format [137] consists of nested lists of keyword-value pairs enclosed by parentheses. It provides a rich set of pre-defined keywords and allows the users to define their own keywords. The example given below, which describes the drawing of Figure 5.11, gives a flavor of this format. Notice that a keyword-value pair—fontsize 0.40, for example—can span more than one line, and several keyword-value pairs can be given on the same line. Also notice that for a vertex or an edge, only its id is mandatory, the other fields are optional and need not be specified.

```
(graph  (fontsize

                0.40)
        (vertex (name "ab")(id 1)
                (x 2.0)(y 1.0)
                (skeleton (height 0.50) (width 0.50))
                (incident-edges 6 7 8 ))
        (vertex (name "ac")(id 2)
                (x 3.0)(y 1.0)
                (skeleton (height 0.50) (width 0.50))
                (incident-edges 6 11 ))
        (vertex (name "ad")(id 3)
                (x 2.0)(y 2.0)
                (skeleton (height 0.50) (width 0.50))
                (incident-edges 8 9 12 ))
        (vertex (name "bd")(id 4)
                (x 2.0)(y 3.0)
                (skeleton (height 0.50) (width 0.50))
                (incident-edges 7 9 10 ))
        (vertex (name "cd")(id 5)
                (x 3.0)(y 2.0)
```

```
        (skeleton (height 0.50) (width 0.50))
        (incident-edges 10 11 12 ))
(edge   (type undirected)(id 6)
        (source 1 destination 2)
        (source-pt (x 2.00) (y 1.00))
        (destination-pt (x 3.00) (y 1.00)))
(edge   (type undirected)(id 7)
        (source 1 destination 4)
        (source-pt (x 2.00) (y 1.00))
        (destination-pt (x 2.00) (y 3.00))
        (shape West 1.00 North 2.00 East 1.00))
        (bends ((x 1.00) (y 1.00))((x 1.00) (y 3.00))))
(edge   (type undirected) (id 8)
        (source 1 destination 3)
        (source-pt (x 2.00) (y 1.00))
        (destination-pt (x 2.00) (y 2.00)))
(edge   (type undirected)(id 9)
        (source 4 destination 3)
        (source-pt (x 2.00) (y 3.00))
        (destination-pt (x 2.00) (y 2.00)))
(edge   (type undirected)(id 10)
        (source 4 destination 5)
        (source-pt (x 2.00) (y 3.00))
        (destination-pt (x 3.00) (y 2.00))
        (shape East 1.00 South 1.00)
        (bends ((x 3.00) (y 3.00))))
                ((x 3.00) (y 3.00))))
(edge   (type undirected)(id 11)
        (id 11)
        (source 5 destination 2)
        (source-pt (x 3.00) (y 2.00))
        (destination-pt (x 3.00) (y 1.00)))
(edge   (type undirected)(id 12)
        (id 12)
        (source 5 destination 3)
        (source-pt (x 3.00) (y 2.00))
        (destination-pt (x 2.00) (y 2.00))))
```

Edges: A file in the Edges format consists of a list of edges, one per line. There is no separate declaration for vertices as they are defined implicitly as part of the edge-definitions. A vertex appearing again refers to a vertex with the same name defined earlier. Edges can not describe drawings. It can only describe graphs. Formally, Edges has the following syntax (in BNF notation):

Graph ::= (Edge \n)*
Edge ::= SrcVertex EdgeDir DestVertex
EdgeDir ::= > | < | – | λ

Here, SrcVertex and DestVertex are ASCII strings denoting the names of the source and destination vertices, respectively, of an edge. SrcVertex > (<) DestVertex denotes a directed edge from (to) SrcVertex to (from) DestVertex SrcVertex DestVertex and SrcVertex – DestVertex both indicate an undirected edge between SrcVertex and DestVertex.

The graph of Figure 5.11 can be described using Edges as follows:

```
ab ac
ab bd
ab ad
bd ad
bd cd
cd ac
cd ad
```

Node/Edge List: The Node/Edge List format divides a file into the following sections:

// nodes: for defining vertices,

// edges: for defining edges,

// directed edges: for defining directed edges

// named edges: for defining edges with names, and

// directed named edges, **and** // named directed edges: for defining directed edges with names.

Sections may be omitted, but the // nodes section always precedes the other sections, which can be given in any order.

Formally, it has the following syntax (in the BNF notation):

Graph := NS {[ES] [DES] [NES] [DNES] [NDES]}
NS := "// nodes" \n (VertexName \n)*
ES := "// edges" \n (SrcVertex \n DestVertex \n)*
DES := "// directed edges" \n (SrcVertex \n DestVertex \n)*
NES := "// named edges" \n (EdgeName SrcVertex \n DestVertex \n)*
NDES := "// named directed edges" \n (EdgeName SrcVertex \n DestVertex \n)*
DNES := "// directed named edges" \n (EdgeName SrcvVrtex \n DestVertex \n)*

Here, VertexName, EdgeName, SrcVertex, and DestVertex are all ASCII strings denoting the name of a vertex, name of an edge, name of the source vertex of an edge, and name of the destination vertex of an edge, respectively.

The graph of Figure 5.11 can be described using Node/Edge List as follows:

```
// nodes
ab
ac
ad
bd
cd
//edges
ab
ac
ab
bd
ab
ad
bd
ad
bd
cd
cd
ac
cd
ad
```

Malf: The Malf format has two versions: Malf-Input and Malf-Output. Malf-Input can describe only graphs, whereas Malf-Output can describe both. A file in Malf format consists of a list of vertices, with one vertex per line, and a list of edges, again one edge per line. The two lists are separated by a line consisting of a single '#' character.

Formally, the syntax of Malf-Input is as follows (in BNF format):

Graph := VertexList "# \n" EdgeList
VertexList := (VertexId [VertexName]\n)*
EdgeList := (EdgeId EdgeDir SrcVertId DestVertId \n)*
EdgeDir := 0|1

The syntax of Malf-Output is as follows (in BNF format):

Graph := VertexList "#\n" EdgeList
VertexList := (VertexId VertexXc VertexYc VertexX| VertexY| [VertexName]\n)*
EdgeList := (EdgeId EdgeDir SrcVertId DestVertId NumBends (BendXc BendYc)$^{2+}$ \n)*
EdgeDir := 0|1

Here,VertexName and EdgeName are ASCII strings denoting the name of a vertex and an edge. VertexId and Edgeid are integers denoting the identity of a vertex and an edge, respectively.

The Malf format requires the VertexId (EdgeId) of a vertex (an edge) to be in the range 1 to N (M), where N (M) is the number of vertices (edges) in the graph. Also, vertices appearing consecutively in the vertex list should have consecutive VertexIds. SrcVertId and DestVertId are identities of the source and destination vertices of an edge. VertexXc, VertexYc, VertexXl, and VertexYl are integers denoting the co-ordinates of the center and the lengths along the x- and y- axes of the rectangle representing a vertex in a drawing. EdgeDir is 0 for an undirected edge and is 1 for an edge directed from vertex SrcVertex to vertex DestVertex. BendXc and BendYc, respectively, denote the integer x- and y- coordinates of the bend points of an edge. Numbends denotes the number of bend points of an edge. MALF-OUTPUT counts the end points of an edge also as bends points, and therefore, each edge at least two bends.

The graph of Figure 5.11 can be described using Malf-Input as follows:

```
1 ab
2 ac
3 ad
4 bd
5 cd
#
1 0 1 2
2 0 1 4
3 0 1 3
4 0 4 3
5 0 4 5
6 0 5 2
7 0 5 3
```

The graph of Figure 5.11 can be described using Malf-Output as follows:

```
1 1 0 0 0 ab
2 2 0 0 0 ac
3 1 1 0 0 ad
4 1 2 0 0 bd
5 2 1 0 0 cd
#
1 0 1 2  2   2 1  3 1
2 0 1 4  4   2 1  1 1  1 3  2 3
3 0 1 3  2   2 1  2 2
4 0 4 3  2   2 3  2 2
5 0 4 5  3   2 3  3 3  3 2
6 0 5 2  2   3 2  3 1
7 0 5 3  2   3 2  2 2
```

Chapter 6

GeomNet: Geometric Computing Over the Internet

Consider two real-life problems. First, suppose you have a collection of nails hammered into a board. If you were to stretch a rubber band around the inside of the nails and let it snap, what shape would the rubber band take? Second, imagine two planes whose trajectories and flight patterns show that they must, at one instance in time, fly dangerously close to one another — will they collide?

In geometric terminology, the shape sought in the first problem is called the convex hull of a set of points in the plane (nails on the board). This is the smallest convex polygon containing all the points. The solution sought in the second problem is an efficient algorithm for detecting potential collisions between objects.

Geometric computing emerged from algorithms developed to solve such problems. It has become a central building block in fields like computer graphics, artificial intelligence, CAD and GIS databases.

Although many representational formats for geometric data exist, only a few fully implement the combinatorial connectivity information that makes geometric data so interesting. Typically, each geometric software package is designed for a unique, incompatible data format. Designing a standard language for describing geometric data would necessitate the encoding of all possible relationships between the numerical and combinatorial components of geometric data — a Herculean task that doesn't seem likely.

In spite of difficulties, there are a number of robust implementations of geometric algorithms. Many of these are available on the Internet (see table 6.1, "Geometric Software on the Web").

Our group at the Center for Geometric Computing developed the GeomNet system to provide easy Internet access to geometric implementations via a plug-and-play environment, forming a link between software evaluation and production modes. By providing a progressive migration of software

Previously published as Gill Barequet, Stina S. Bridgeman, Christian A. Duncan, Michael T. Goodrich, and Roberto Tamassia. GeomNet: Geometric computing over the Internet. *IEEE Internet Computing*, 3(2):21–29, 1999.

Collections of geometric software can be found on the Internet at Amenta's "Directory of Computational Geometry Software" site (`http://www.geom.umn.edu/software/cglist/`), and a host of geometric computing applications can be found at Eppstein's "Geometry in Action" site (`http://www.ics.uci.edu/~eppstein/geom.html`). Erickson has an even more extensive collection of geometric software at his "Computational Geometry Software" site (`http://www.cs.duke.edu/~jeffe/compgeom/software.html`). In addition, there is a large collaborative effort under way to develop an extensive Computational Geometry Algorithms Library (`http://www.cs.ruu.nl/CGAL/`) [67, 123] which itself is built upon the successful Library of Efficient Data Types and Algorithms (`http://www.mpi-sb.mpg.de/LEDA/leda.html`) [44, 112]. This effort is directed at building a large collection of C++ routines for solving computational geometry problems. This collection should grow significantly in the years ahead.

Table 6.1: Geometric software on the Web.

from the host to the client, GeomNet attempts to simplify interfacing, one of the most significant problems in software engineering and in using software developed elsewhere. Users can invoke online, separately or in a pipeline, a rich collection of geometric computations for performing one-time or repeated tasks. The system is suitable for a wide variety of tasks, such as invoking an algorithm with specific input data, checking geometric structures or data for consistency, experimentally studying and/or comparing algorithms, designing new algorithms through the integration of existing algorithms, or demonstrating the course of an algorithm in an educational setting.

6.1 A Cooperative Computing Environment

We envisioned GeomNet as a family of geometric computing servers executing various geometric algorithms on behalf of remote clients. These clients can be users interfacing through a Web browser interface, or application programs connecting direction through sockets. We refer to negotiations between client and server as *cooperative computing*.

GeomNet is based on a layered object-oriented architecture (see Figure 6.1), with the highest level responsible for interacting with client processes and the lowest level responsible for interacting with specific geometric programs. This structure divides a GeomNet server's tasks into separate conceptual units and hides the details of specific implementations from components that don't need that information. This makes it easier both to maintain GeomNet and to incorporate new algorithms into the system.

GeomNet supports file formats commonly used in various communities. Formats for describing three-dimensional geometries include .OFF (used by the data visualization and manipulation Geomview [2], see `http://www.geom.umn.edu/software/download/geomview.html`), .WRL (for VRML files), and .STL (a manufacturing file format). The system also supports several graph formats. We are continuously incorporating more formats.

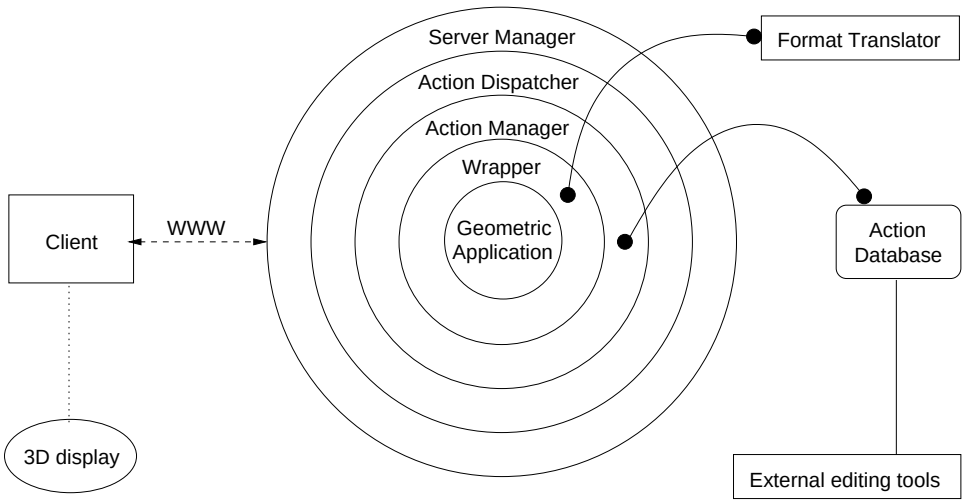


Figure 6.1: The GeomNet architecture. A client sends a request to one of the GeomNet servers, where the server manager adds it to its queue of incoming messages. The action dispatcher extracts messages from the queue and forwards them to the action manager, which decodes the message and, using the action database, invokes the wrapper of the application that will handle the request. After termination, the output is forwarded back through the layers and returned to the client.

Briefly, users submit data in one of the supported data formats and request a computation from a GeomNet server. GeomNet automatically converts the data to the format assumed by the requested computation, performs the computation, and returns the result in one of many popular formats, including interactive two- and three-dimensional visualization programs already plugged in to the user's Web browser or, alternatively, the user's application.

6.1.1 Client-Server Dialog Protocol

The dialog protocol between GeomNet's servers and clients was designed and implemented such that a client can call applications on the server's side. Messages exchanged between GeomNet clients and servers have a keyword-value structure. That is, every message is composed of a list of pairs consisting of a keyword, which identifies the name of a field in the message, and a value, which represents the contents of that field. This type of syntax is not only simple to parse and to

A first implementation of GeomNet is available on the Internet through the support of the Center for Geometric Computing at Johns Hopkins University (`http://www.cs.jhu.edu/labs/cgc/`) and at Brown University (`http://www.cs.brown.edu/cgc/`), with prototype servers available at the GeomNet sites (`http://www.cgc.cs.jhu.edu/geomNet/` and `http://loki.cs.brown.edu:8081/geomNet/`) at the respective universities. New releases are planned throughout 1999 and 2000.

Table 6.2: Where to access GeomNet.

understand, but also very powerful and expressive.

The client's application (either an interactive applet embedded in a Web page or a stand-alone program) sends a message to a GeomNet server, which expects the user to select a geometric operation (called an action) from the list of available operations (algorithms). Implicit in this selection are any parameters required for the specific action. Some parameters are mandatory, for example, the action name and the input filename. Some parameters are optional. If, say, the input file type is omitted, the system automatically derives it from the filename extension. Still other parameters are specific to the algorithm, for example, a tolerance for an approximation algorithm. Either the user or the client program could select the exact values assigned to these parameters. In either case, the message sent from the client encodes the requested action and the accompanying parameters in a Web-like syntax:

```
GN_action=convex-hull&GN_infile=large-set.off& \
GN_outfile=small-set.wrl&GN_intype=stl&mode=2-d&type=upper-hull
```

The message is further encoded by using the MIME type

```
application/x-www-form-urlencoded
```

so that special characters in the keyword and value strings are replaced by their ASCII hexadecimal values.

GeomNet keywords begin with the prefix GN_. Each application may also have its own set of additional keywords. In the example above, the application itself can use the keywords mode and type. Indeed, the same keyword may be used for different actions. Action-specific keywords are not necessarily unique; for example, keywords such as epsilon, mode, and angle are expected to be shared by many algorithms.

Note the similarity between this protocol and Java's remote-method-invocation mechanism: A keyword-value structure is simply a way to invoke a (remote) procedure by setting actual values to formal parameters. In addition, our protocol allows GeomNet to have default values, aggregate arguments (encoded appropriately in the message), and indirect referencing (by specifying a URL as an argument). In particular, arguments are passed by name, and their order is insignificant.

6.1.2 Computation Requests at the Server

GeomNet servers are responsible for decoding and fulfilling client requests for geometric computations. A GeomNet computation begins with the server manager, which tracks client requests and sends responses back to the appropriate clients. The server manager handles a queue of incoming messages to maintain the incoming requests. To process the next awaiting message, the server manager calls its action dispatcher, which extracts the message from the queue and forwards it to the action manager on that machine.

The action manager decodes the message according to the agreed-upon syntax and looks for the mandatory field GN_action. The action is then handled using the action database, where all

available actions are fully specified by their names and parameters. This database, saved in an ASCII control file written in Java, identifies the tool that should handle the action. More specifically, the action database maintains details concerning the availability of an action and the name of the Java wrapper class that should be used for performing that action. The action database does not contain information about the interface of the actual tool (usually a stand-alone program) that performs the action. It should know only what input and output formats are needed so that it can, if necessary, also schedule a format conversion routine.

The action manager performs file-type conversions, if necessary, before and after calling the wrapper level. Then the action manager invokes the wrapper of the application that should handle the requested action and passes it the necessary data. When the wrapper terminates, the action manager returns the action's output to the action dispatcher, which then forwards it to the server manager. This is done by passing back a locator (filename) for the result or, if it is small enough, by passing back the actual result encoded in a URL.

6.1.3 Wrapper-Application Interaction

The knowledge of what actual code or environment must be invoked to satisfy an action is at the bottom layer of the architecture, where we store the Java wrapper class that interacts directly with the application. The action manager invokes the serving application's wrapper class by name. It obtains this name at runtime (rather than during compilation) and binds the wrapper by using Java's runtime class loader. This allows for the addition of new algorithms (through new wrappers) without having to recompile, or even restart, the server.

An application's wrapper is the only system component that knows how to invoke the application. Thus, modifications made in the interface to the application, or even in the actual code for implementing an application, require only recompilation of the wrapper. No other GeomNet component is affected by changes made in the actual implementation of its geometric components. The wrapper performs all necessary preparations for properly invoking the associated application, then invokes it, receives the application's output, and returns the result to the action manager. The wrapper terminates an application running too long by sending a time-out interrupt.

Adding a new application requires coding a suitable wrapper (that agrees with the interface conventions), updating and recompiling the control file, and restarting the system. Entire system recompilation isn't needed; thus new applications can be added to every system installation without the system having the application's source code. The end user, however, can use only system-embedded applications.

Typically, geometric applications belong to one of three types: programming-language functions, stand-alone computer programs, and embedded applications. We have implemented wrappers for all three types.

A *programming language function* expects the input data in certain data structures to be passed to it as actual arguments. Optionally, an argument can contain the name of a data file. This is usually the case when the function is really a "roof" over a stand-alone program. When calling a

function, the wrappers first parse the temporary input file and prepare the data structures required by the function. They then call the function according to its calling sequence. Finally, they scan the data structures returned by the function and write the contents in a temporary output file with the format requested by the server.

A *stand-alone computer program* expects all the input in a command line to be extracted by the famous C argc/argv mechanism, or by Java's args[]. These arguments include the names of the input and output files and other arguments to the program. The program expects the input file to be in one of several predefined file formats and usually writes the output in the same file format. If the program does not recognize the syntax of the temporary input file, the wrapper converts the input file into a secondary input file with a syntax the program recognizes. The wrapper runs the program (executable or a shell script) according to its calling sequence. If the syntax of the temporary output file provided by the program is not the one the server requires, the wrapper converts the output file into a secondary output file with the requested format.

An *embedded application* operates only in a specific environment, such as a CAD system or a geometric database. The application expects the data to be preloaded into the environment that the application works in. The environment supplies the methods for accessing and modifying the geometric data, so that the application usually needn't be aware of the nature of the actual data structures that store these data. The output is the contents of the environment upon termination of the application. To invoke an embedded application, the wrappers first set the environment. This means they initialize the environment, parse the temporary input file, and load its contents into the environment. The wrappers then invoke the application according to its specifications. To unset the environment, the wrappers scan the environment and save its contents into a temporary output file with the syntax requested by the server. Lastly, they properly close the environment.

The user specifies an output syntax but not an output filename. GeomNet returns the output in a temporary file with the required syntax. The user may then choose a file to save the data into. Finally, GeomNet removes the temporary output file automatically.

6.2 Interfaces and Applications

GeomNet is representative of systems that enable distributed computing over the Internet in the sense that the scheme used in GeomNet can be used to develop similar systems in other disciplines. In this section, we give several examples of how our system can be applied in the field of geometry.

6.2.1 Classic Geometric Algorithms

A prime application for GeomNet is invoking implementations of fundamental geometric algorithms. Therefore the current GeomNet release includes implementations of several algorithms, such as d-dimensional convex hulls (including a random-box generator for code testing purposes), Voronoi diagrams, and Delaunay triangulations.

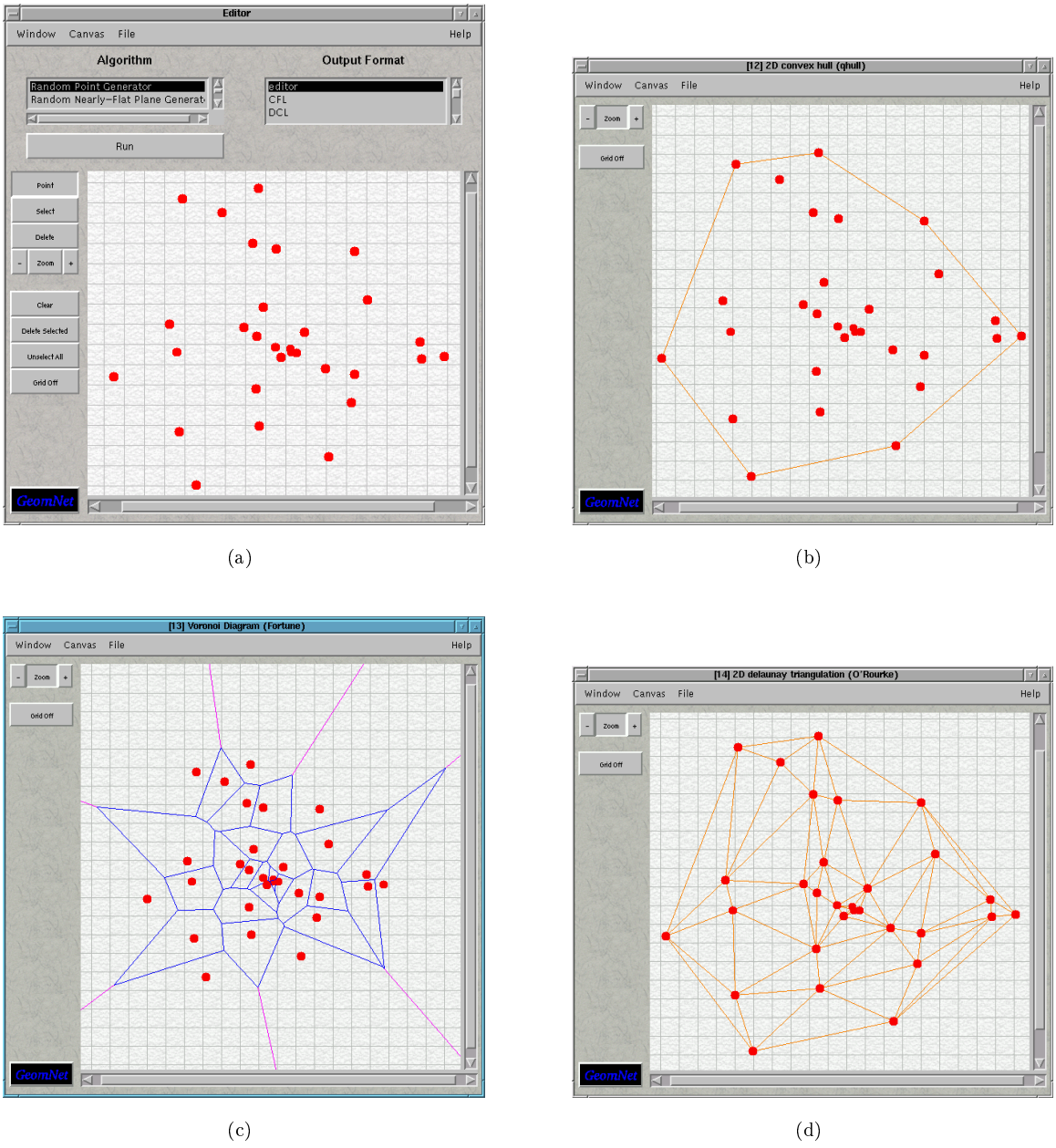


Figure 6.2: Applet interface for algorithms in two dimensions: (a) point set, (b) convex hull, (c) Voronoi diagram, (d) Delaunay triangulation.

We developed several interface mechanisms for invoking these routines. The simplest is the interactive applet interface. Figure 6.2(a) shows a Java point-set editor embedded in GeomNet.

The purpose of the interactive applet interface is clearly not to perform intricate large-scale computations but rather to interact with the system so as to gain intuition about geometric algorithms

and to easily test GeomNet servers. The user can then apply any algorithm for which an appropriate application is bound to the system. Figures 6.2(b), 6.2(c), and 6.2(d) show the convex hull, the Voronoi diagram, and the Delaunay triangulation, respectively, of the point set shown in Figure 6.2(a), as computed by the appropriate applications and displayed by GeomNet's applet. Some operations don't require graphical input, only parameter settings. For these operations GeomNet provides a general form for obtaining user input.

For production-level usage we provide a socket interface that can be invoked directly from inside an application program running on the client's site. This interface lets any program, not just Java applications, communicate with the server via the GeomNet protocol. We also provide a forms interface for downloading geometric data files. In both cases the server output can be sent immediately to an interactive display system running on the client, or it can be piped into a client application (including the calling application). Figure 6.3 shows such a response, the result of a client application that first requested GeomNet to generate a random sample of points in R^3 and then returned that set of points for GeomNet to compute its convex hull.

GeomNet's response was then sent immediately to a client invocation of the Geomview system. Here Geomview was used as a plug-in for the client's application. However, it could also have been used as a Netscape plug-in for someone using the forms interface or the applet editor.

Users may choose three-dimensional browsers to serve as plug-ins to the Web browser (when opening the GeomNet page) or as plug-ins to the user's client (if the socket interface is used). Often users can request that a GeomNet application's output be specified in a format suitable to the user's plug-ins or helper applications.

6.2.2 Drawing Abstract Graphs

Graph drawing is an evolving research area. The objective is to produce a graphical representation from a set of objects (nodes) and connections between them (edges). A graph is an abstract entity that has many possible drawings — deciding which drawing is of the highest "quality" is rather subjective and depends on the type of graph and the application. Common factors for determining quality include minimizing the number of edge intersections or the total area of the picture.

The Graph Drawing Server, a GeomNet component, can be used for drawing graphs from user applications, studying and comparing graph-drawing algorithms, converting between different formats for describing graphs and their drawings, creating a database of graphs occurring in user applications, and providing educational demonstrations.

The user must specify the graph style, the input format, the type of service desired (drawing or format conversion), the output format, and the drawing algorithm (if the graph is to be drawn). To provide the input graph in HTML form, the user can either give the URL of the file containing the graph or simply type the graph description in the form itself. The graph editor applet lets the user interactively draw the input graph on the screen. GeomNet currently supports four orthogonal-drawing algorithms for general graphs, as well as several algorithms for drawing specialized graphs. A prototype version of the Graph Drawing Server can forward requests as needed to other servers.

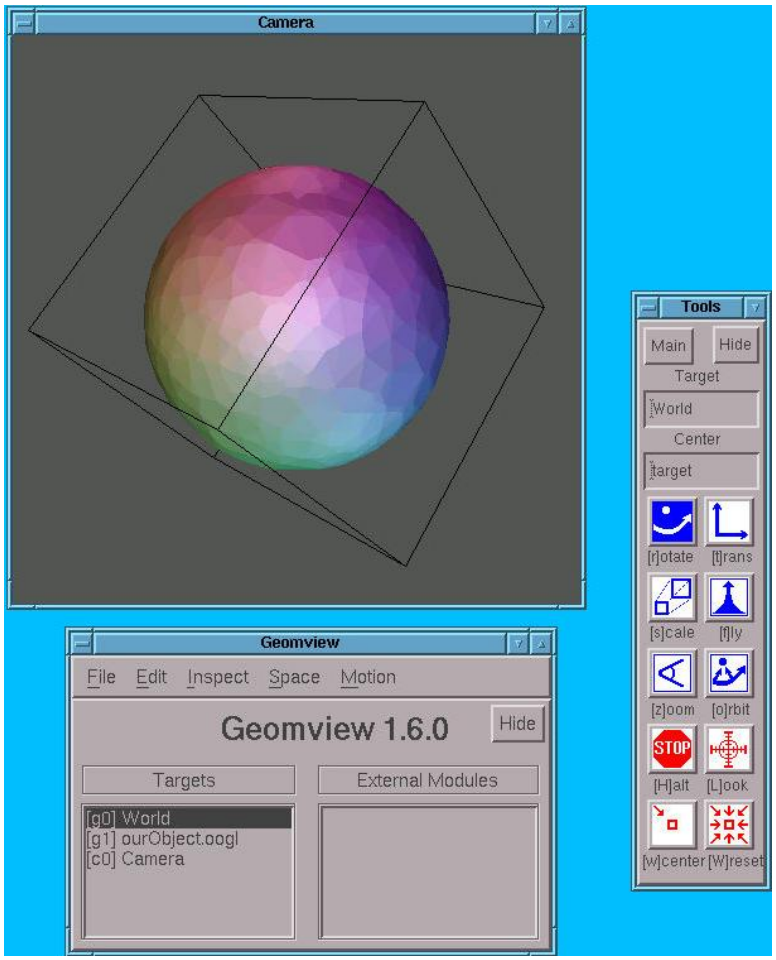


Figure 6.3: A three-dimensional display plugged into GeomNet.

Figure 6.4 compares the performance of several graph-drawing algorithms on a diagram taken from a flights database.

Figure 6.5 shows the application of a graph-drawing algorithm on the hierarchy of GeomNet's Web pages. The user specifies the URL of the hierarchy description in the input form (Figure 6.5(a)). Figure 6.5(b) shows the textual output of invoking the Sugiyama hierarchical graph-drawing algorithm. The output page provides a thumbnail image of the drawing (linked to the full-size image) and some statistics on the output, as well as convenient links for obtaining the output in a different format. Figure 6.5(c) shows the same input entered in the graph-editor applet, and Figure 6.5(d) shows the graphical output. The graphical output is used, for example, to demonstrate how many links the GeomNet user must follow to find a particular page.

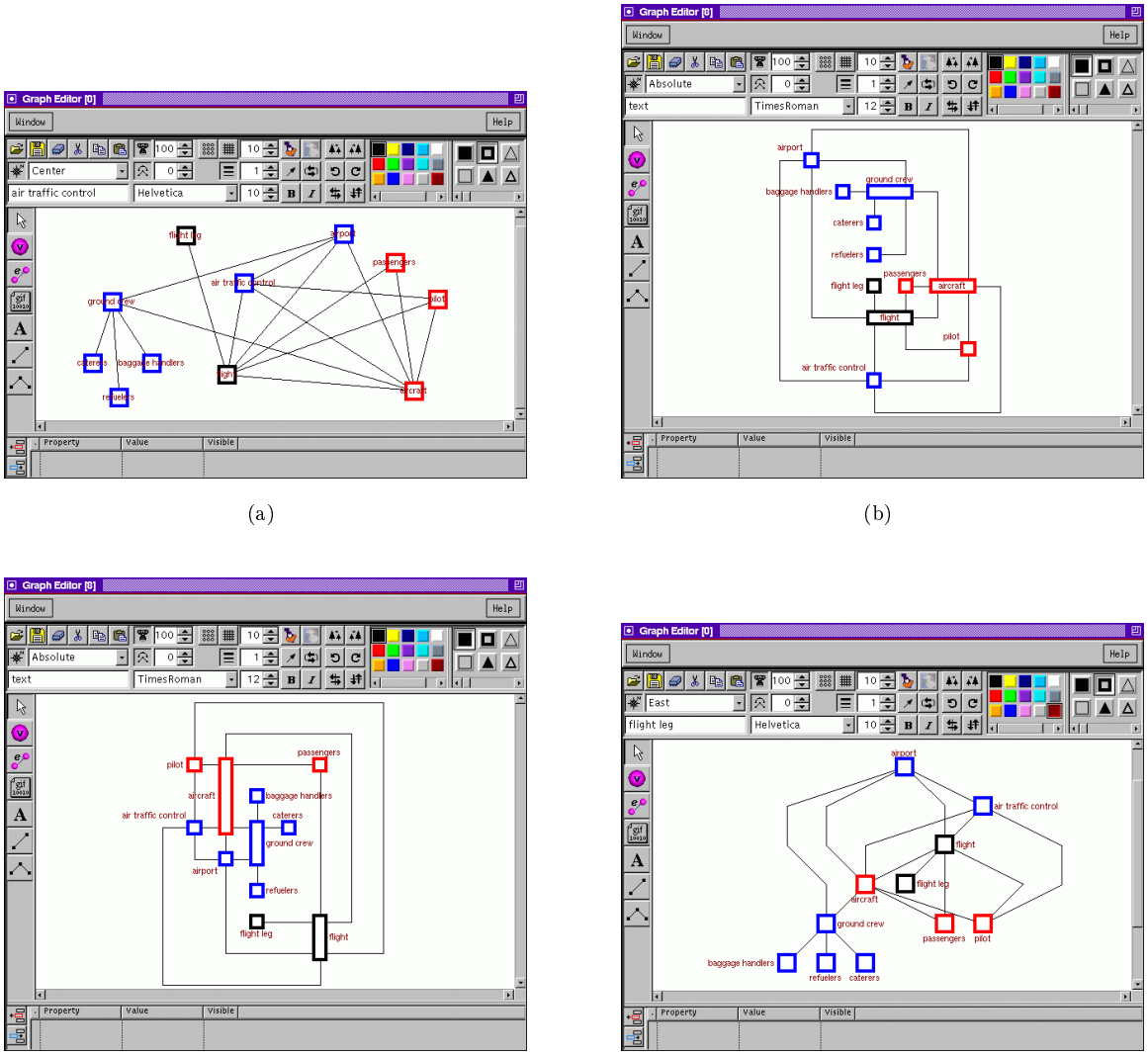
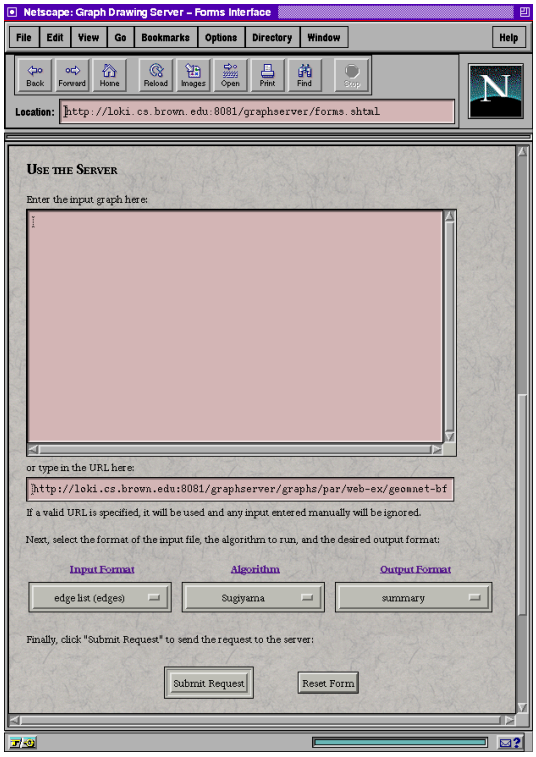


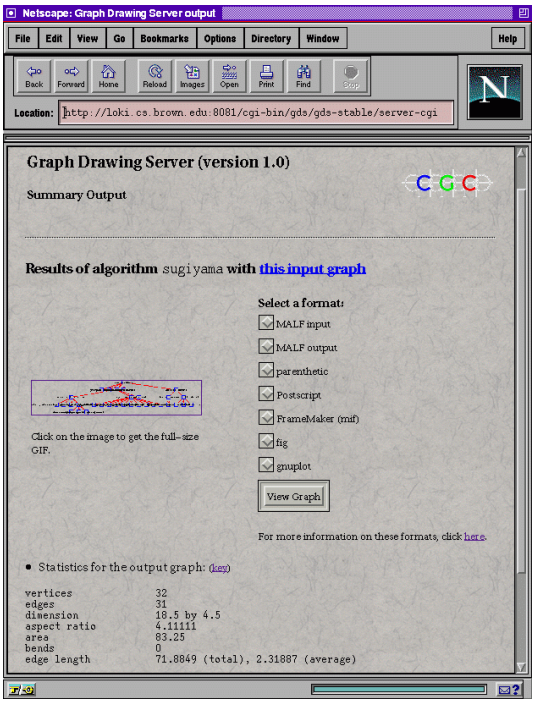
Figure 6.4: Some graph-drawing examples: (a) the flights database, (b) the Bend-Stretch algorithm, (c) the Giotto algorithm, (d) the Sugiyama algorithm.

6.2.3 Geometric Algorithm Animation

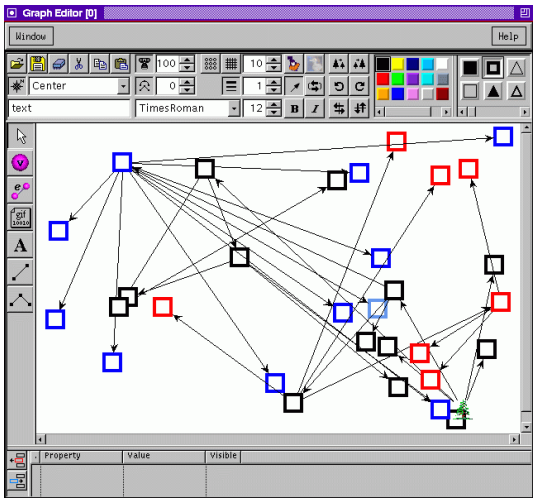
Animation can be used as a tool to understand an algorithm's operation. We are incorporating geometric algorithm animation capabilities into GeomNet by extending previous work on the Mocha model for Web-based algorithm animation [7]. In the Mocha model, algorithm animation consists of two components: the algorithm server, which executes the algorithm and produces a sequence of interesting events, called algorithm operations, and the animation component, which provides the multimedia visualization of the algorithm operations. The animation component can be further



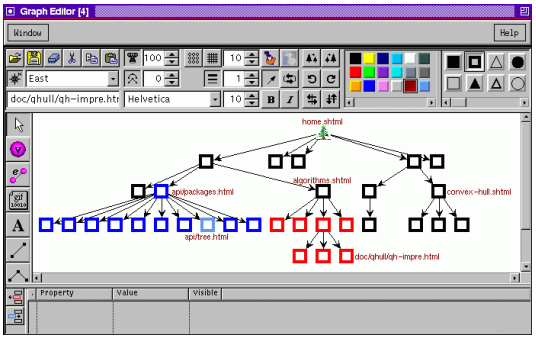
(a)



(b)



(c)



(d)

Figure 6.5: GeomNet's Web page hierarchy: (a) input form, (b) textual output, (c) graphical input, and (d) graphical output.

divided into the GUI, which handles the user's interaction with the interface and sends changes in the input to the algorithm server, and the animator, which receives algorithm operations from the algorithm server and updates the display accordingly.

Currently supported are animations of the convex hull, Voronoi diagram, and Delaunay triangulation algorithms provided by the Library of Efficient Data Types and Algorithms (LEDA), and the computation of a proximity graph of a set of points.

6.2.4 Experimental Results

We ran our geometric algorithms server on a Pentium Pro 200-MHz machine using Linux, and measured the running time of the algorithms for two- and three-dimensional Delaunay triangulations and three-dimensional convex hulls. For a reasonably sized input (tens of thousands of points) the system provided the output in less than 30 seconds (plus transportation time, which varied).

We have collected statistics on the performance of the graph-drawing server, which was run on a Sun UltraSparc workstation using Solaris 2.5/2.6. For graphs with up to 100 nodes and 140 edges, the typical running time of a drawing algorithm was less than 10 seconds, with less time spent on input and output format conversions and on other system overhead. A little more time was needed for conversion if the output format was graphical (for example, a GIF or PostScript file).

6.3 Future Work

Future GeomNet development will incorporate new types of servers, upgrade existing servers to control more algorithms, and expand the online help, which is now only partially available. These enhancements will make GeomNet functionality richer and appropriate to more research domains. Its modular structure makes these tasks simple.

A super-action mechanism, now under development, will give the user a scripting language for conditioning, pipelining, and iterating regular actions. This feature will enable further automation of complex sequences of operations performed on geometric objects.

We also plan to enhance the system to utilize distributed computing where appropriate. This means implementing a cluster of servers that can assign tasks or subtasks to one another. In a homogeneous environment, where most machines running the servers have the same resources, a task should be solved in parallel, to the extent that the underlying problem is parallelizable. Conversely, if the machines' computing capabilities differ, then each server should handle the problems it is best suited for.

We also intend to explore the possibility of downloading and running an application on the user's site, thus reducing the load on the server. This requires the application author's consent to Internet distribution of the application. Also, the user's machine would need the code type and machine resources to run the application. Once the system supports a cluster of servers, users can add their own applications by starting a server locally at their site. They can also submit applications to a server's owner for inclusion, although the current server assumes applications are trusted and

protects only against crashes and infinite loops. Additional security measures are needed to protect the server from applications that may cause accidental or malicious damage. Finally, we also plan to implement a mechanism (a metalanguage) that will let the user define the input and output format for the transferred data. This would remove the limitation that the user submit data in one of the formats the system recognizes.

6.4 Acknowledgments

We thank Yair Amir, Jim Baker, Ashim Garg, and Subodh Kumar for several useful discussions regarding geometric software and distributed computing.

Chapter 7

PILOT: An Interactive Tool for Learning and Grading

7.1 Introduction

Interactive World Wide Web (WWW)-based learning tools have become the focus of research for a large number of computer science educators [26, 27]. Interaction and animation in and out of the classroom offer the chance to actively engage students in the learning process. Several interactive educational tools have been developed over the last few years. Many of these, however, quickly become obsolete as hardware/software platforms and operating systems change. With the advent of platform-independent applications, there are far greater possibilities for creating more useful educational tools. While many computer science courses offer online access to handouts, syllabi, homework assignments solutions and other static documents, only a few have begun to exploit the full potential of the new technology available to us.

Online testing systems can be useful in distance learning, virtual universities, and online classes, and several systems that allow for online testing have been developed in the last decade (e.g., see [46]). Such systems tend to support multiple-choice questions, which provide a natural class of questions that can be automatically graded online. While such questions can be used to provide useful measures of student learning, we believe there are significant additional learning and testing opportunities available that have yet to be fully exploited. In particular, because of the ability to formally define input and output specifications, there are other more complex questions that should also allow for automatic online grading, at least in theory. Some of the immediate advantages of online grading for richer sets of questions are the ability to test students' answer *creation* abilities rather than simply their answer *choosing* abilities. In addition, online grading also provides fast and consistent grading, provably correct solutions, and pointers to information relevant to the question.

Previously published as S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *Proc. ACM Tech. Symp. on Computer Science Education (SIGCSE)*, pages 139–143, 2000.

Moreover, online grading also allows the assignment of different questions to the students, thus reducing issues of cheating and plagiarism.

We are therefore interested in interactive online automated grading tools that aid student learning and test answer creation abilities, not just answer choosing skills. In addition, we are interested in the visualization of questions, errors, and answers.

7.1.1 Previous Work

Several previous software systems have been designed with online testing in mind [110, 111]. Blackboard.com [22] provides automatic grading for quizzes with multiple choice and true/false questions. Systems such as QUIZIT [150], WebCT [153], and ASSYST [99] have been designed to perform online testing of answers whose correct syntax can be specified as regular expressions. Previous systems that allow for richer types of answers have needed assistance from the course graders and the instructor to perform the actual answer checking. In addition to the difficulty of dealing with sophisticated forms of answers, another area where these previous systems have trouble is in their lack of ability to provide partial credit to answers that are "almost" correct.

Intelligent tutoring systems have been an area of research in artificial intelligence for several decades. The primary application of these systems is providing feedback and tutoring the student rather than grading responses. Many systems, such as the LISP Tutor [4] and the Geometry Tutor [3], focus on abstract problem-solving skills and thus are more complicated than our aims here.

Since our notion of answer specification and checking involves a strong visualization component, it is also related to previous work on the visualization of algorithms and data structures. There is a rich literature that describes the benefits of concept visualization in education settings. Algorithm animation has been successfully used for visualizing graph algorithms, sorting, and searching, to name a few examples [140]. Similarly, program code animation also helps in the learning of new programming languages. Finally, concept animation has also been successful in in communicating difficult concepts such as finite state automata [26]. Tools for creating animations of data structures and algorithms have also been developed [130]. Interactive tutorials have been designed and their positive impact on student learning evaluated [11]. Electronic books have been proposed and developed, in which hypertext, interactive animations, audio and video parts are integrated in a web-based standalone educational resource [25].

7.1.2 Our Results

We have designed a Platform-Independent Learning Online Tool, PILOT, with several goals in mind. First, we would like to offer an interactive tool that can be used in class to aid in exposition. Furthermore, there are numerous problems that students learn best by example, and we would like a tool that can generate random instances of a problem and allow the student to create the solution online. Finally, we would like to allow for automated grading so that the student can receive immediate feedback on her work. Thus, PILOT allows for:

- WWW access and platform independence
- generation of interesting random instances of a problem
- user interaction to specify a solution
- online submission of solutions for evaluation
- evaluation of solutions, providing a score and comments
- generation of correct solutions to the problem

At this time, PILOT supports graph problems such as finding the minimum spanning tree (MST), tree search algorithms (breadth first search (BFS) and depth first search (DFS)), and shortest path algorithms.

One of the main advantages of PILOT is that it is a platform independent client/server based applet that can be run from a browser such as Netscape or Internet Explorer. Another even more important feature is its capability to successfully interact with the student by providing detailed feedback. For example, in creating a MST, if an edge is chosen incorrectly, PILOT will highlight the edge and suggest how to correct it.

With additional security, PILOT can be used for grading by allowing graders to input both problems and students' solutions. (In the current system, PILOT can only be used to check problems generated on the spot. This is to prevent students from entering their homework problems and using PILOT's problem-solving capabilities to obtain solutions.) Such security could take the form of password protection or encryption, to allow only authorized users to connect. Additional security would also allow the use of PILOT in testing situations, where it is important to ensure that each student only submits one version of the answer.

7.2 Using PILOT

In the current scenario, the user chooses a problem type from a pull-down menu and clicks the "generate" button to create a random instance of that problem. Figure 7.1(a) shows the result of generating an instance of MST-Prim — a minimum spanning tree problem to be solved using Prim's algorithm. PILOT easily allows testing of both general concepts ("find a minimum spanning tree of the given graph") and specific algorithms ("find a minimum spanning tree of the given graph, using Prim's algorithm"). For MST-Prim, the user is to execute Prim's algorithm, starting with the vertex marked "start"; the solution is a numbering of the edges in the order in which they were added to the MST. To indicate the solution, the user clicks on the edges belonging to the MST. The order can be adjusted in the "Edge Ordering" window; by default, the edges are listed in the order in which they are selected.

Once the user is satisfied that she has entered the correct solution (Figure 7.1(b)), clicking the "check" button will correct and grade the solution. The system will display the graph with the incorrect edges highlighted, along with a score and an explanation of the errors made; see Figure 7.1(c). Note that the actual solution is not displayed — this is because the checker may not actually compute the solution in the process of grading the user's input. A solution can be obtained

at any point by clicking the "solve" button; see Figure 7.1(d).

7.3 PILOT Architecture

PILOT uses a client-server architecture, and is built on top of GeomNet [9]. In the GeomNet model, the client is responsible for maintaining the user interface and all of the algorithm-related computation is done on the server. For PILOT, the client is implemented as a Java applet and the server side contains the problem generators, checkers, and solvers, currently also implemented in Java. The main motivation for choosing the client-server architecture was flexibility — the server is not constrained by the security restrictions placed on applets and is not limited to running Java programs, making it possible to take advantage of existing tools. The graph generator, for example, uses the Graph Drawing Server [35] component of GeomNet to compute a layout for the automatically generated graphs. The modularity of the GeomNet system also makes it easy to add new components — both interfaces and problem generators/checkers/solvers — to PILOT.

We now look at the graph generator and problem checker components of PILOT in more detail, focusing on minimum spanning tree problems as an example; the problem solvers are straightforward implementations of the appropriate problem solving algorithms and are not considered further.

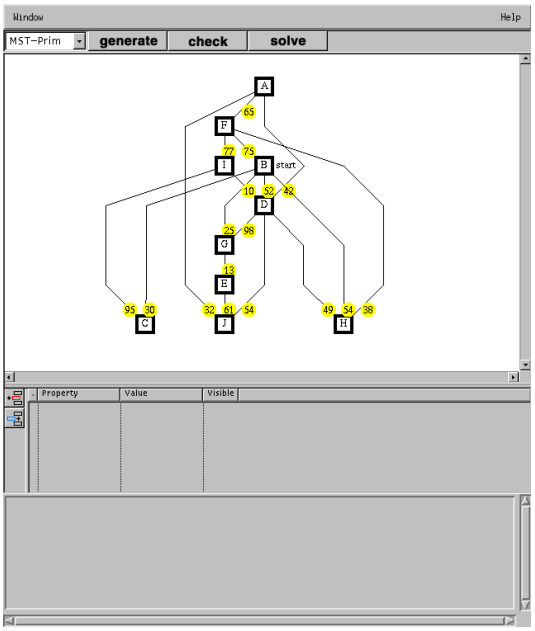
7.3.1 Graph Generator

The graph generator uses a method similar to that of [57] to generate "realistic" graphs for experimental purposes. Graphs are built from a single vertex by repeatedly applying three operations — (1) insertion of a vertex and a random number of adjacent edges, (2) insertion of an edge between two existing vertices, and (3) splitting of an existing edge by replacing it with a new vertex and two new edges. Graph properties such as the ratio of edges to vertices can be controlled by adjusting probabilities assigned to each of the operations and the degree of newly inserted vertices.

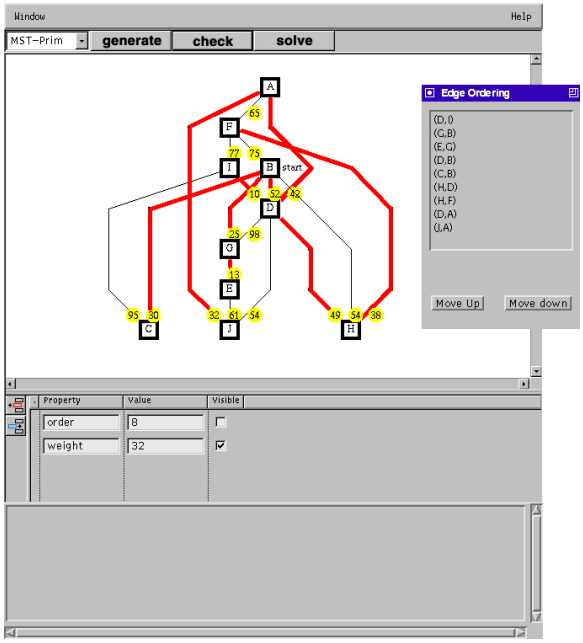
7.3.2 Problem Checkers

There are four main challenges in designing problem checkers: determining what constitutes a solution, handling non-unique solutions, assigning appropriate partial credit, and returning meaningful comments.

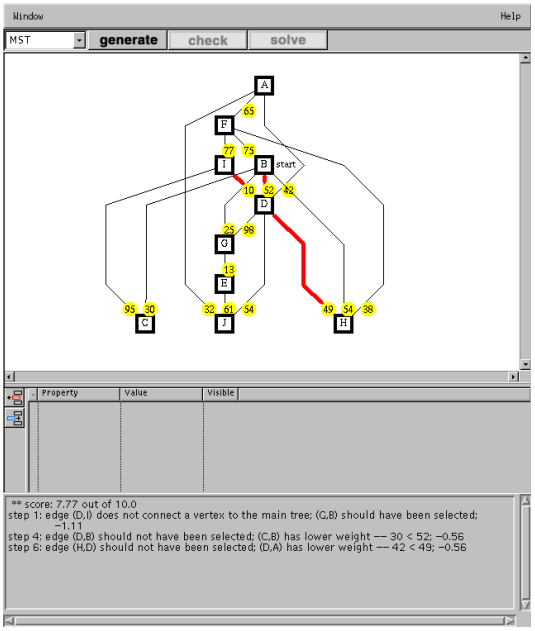
The format of the solution fundamentally affects the structure of the checker. For example, the MST problem simply tests whether or not the user can construct a minimum spanning tree, and so the solution is a list of the edges belonging to the MST. The checker simply verifies whether or not the right edges were selected. In MST-Prim, the goal is to test the user's knowledge of a specific algorithm and so more information is needed in the solution. In this case, the order in which the edges are added to the MST is sufficient to verify that the user executed the algorithm correctly, and the checker must check this order.



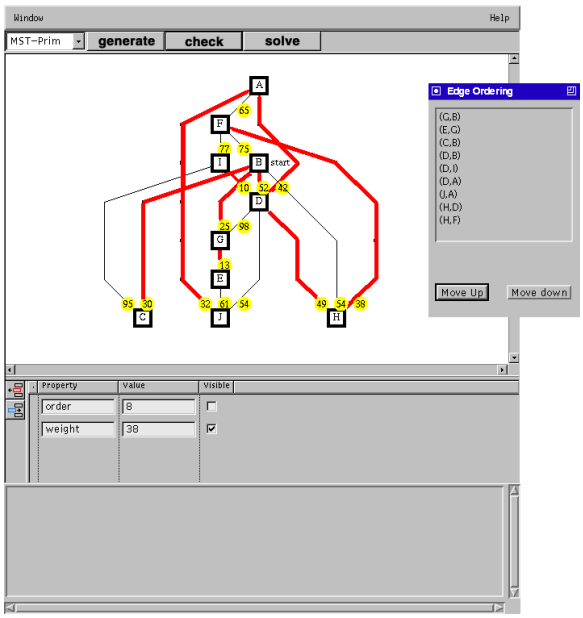
(a) random instance of MST-Prim



(b) user's solution



(c) automatically corrected solution, with incorrect edges highlighted



(d) system-generated solution

Figure 7.1: Example of user interaction with PILOT.

The last three problems are related. It is relatively easy to compute a solution and compare the user's input to it, simply returning "correct" or "incorrect" (or "full credit"/"no credit"). However, this unfairly penalizes a student who understands the concept but makes a small mistake, and is of limited use to a student who is trying to master a concept. More appropriate responses for MST problems, for example, would be something like "Edge (a,c) should be replaced by the lower-weight edge (a,b)" and a 1-point penalty for each incorrect edge. The solve-and-compare approach also runs into problems when the solution is not unique, since the user may have a correct solution but be marked wrong because the system generated a different one. Non-unique solutions can easily occur in MST problems when multiple edges have the same weight.

One approach is to verify properties of the user's solution, to ensure that it is valid. This is the approach taken in the MST checker — for each edge in the MST, that edge should be the lowest-weight edge of any connecting the two vertex partitions created by the removal of the edge from the spanning tree. Each time an edge violates this property, it is marked incorrect and the appropriate replacement edge can be indicated to the user. Partial credit can be assigned according to the number of incorrect edges. (If the user's input is not a spanning tree, cycles are broken by removing the highest-weight edge in the cycle and trees are joined by adding the lowest-weight edge between the trees. The checker then proceeds with the spanning tree produced, adding an additional penalty for non-tree input.)

This approach partially addresses the problem of meaningful comments and partial credit, but is not appropriate for problems where an early mistake can be compounded. For example, if the user chooses the wrong edge in the first step of Prim's algorithm but otherwise executes the algorithm properly, the one mistake may cause several other edges to be selected incorrectly. It is unfair to penalize the user for every edge that is wrong since it was actually only one mistake, and the system's comments may be similarly misleading. A checker can solve this problem by taking an incremental approach and stepping through the solution of the problem, taking into account the user's choices as they happen. The MST-Prim checker considers the user's edges in order, testing each edge to determine if it is valid as the next choice. An edge is valid if it connects a new vertex to the spanning-tree-in-progress and has the lowest weight of any edge connecting a new vertex to the tree. A penalty is assessed if the user's edge is not valid, with a higher penalty if the edge does not connect to the tree. The internal data structures are then updated to include the new edge, and the checker continues with the next edge.

7.4 Future Work

The current PILOT system can be extended in many ways. Of particular use in a teaching tool would be to allow greater interactivity — as the user works through the problem, the system can immediately provide feedback as to whether or not the user is doing the right thing. Also, if the user is unsure of what step to take next, the system can provide hints or outright statements about what to do.

Another issue is the generation of problems of approximately equal difficulty (and, related to this, the generation of appropriate special cases). For example, in Prim's algorithm the addition of an edge and vertex to the spanning tree may result in a new, lower-weight connection for an unconnected vertex and thus change the best choice for the next vertex/edge pair added to the tree. Problems with many instances of this case may be viewed as harder than problems without, since they require knowledge of particular cases in the algorithm. This is particularly relevant if PILOT is used in a testing situation, since it is undesirable for one student to get an easy case when another is faced with a much harder example. Dealing with this involves looking more carefully at the properties of the graphs produced by the graph generator.

Problem checkers can pose challenging problems of their own. The issues are the same as those mentioned in Section 7.3.2 — determining an appropriate format for the solution and handling partial credit and comments. Partial credit is one of the most "human" tasks of grading, and one that is very subjective, and so determining appropriate ways to handle it automatically is an important task. Implementing checkers to assign partial credit can be significantly harder than the corresponding problem solvers.

Finally, PILOT can be extended to handle additional problem types and algorithms. The mechanism for doing this is straightforward — many other graph problems, such as maximum flow, can be supported by the current interface so all that is required are additional checkers and solvers. Adding new problem types, such as sorting, requires more work to create a new interface in addition to generators/checkers/solvers. In both cases, however, the server remains the same so adding new components is only a matter of plugging in a new front- or back-end.

7.5 Acknowledgements

Thank you to Ryan Baker for useful discussions regarding PILOT.

Bibliography

[1] H. Alt, O. Aichholzer, and Günter Rote. Matching shapes with a reference point. *Internat. J. Comput. Geom. Appl.*, 7:349–363, 1997.

[2] N. Amenta, S. Levy, T. Munzner, and M. Philips. Geomview: A system for geometric visualization. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C12–C13, 1995.

[3] J. R. Anderson, C. F. Boyle, and G. Yost. The geometry tutor. *J. Math. Behavior*, pages 5–20, 1986.

[4] J. R. Anderson and B. J. Reiser. The LISP tutor. *Byte*, 10:159–175, 1985.

[5] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[6] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. Algorithm animation over the World Wide Web. In *Proc. Int. Workshop on Advanced Visual Interfaces*, pages 203–212, 1996.

[7] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. The Mocha algorithm animation system. In *Proc. Int. Workshop on Advanced Visual Interfaces*, pages 248–250, 1996.

[8] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. Visualizing geometric algorithms over the Web. *Comput. Geom. Theor. Appl.*, 12:125–152, 1999.

[9] G. Barequet, S. Bridgeman, C. Duncan, M. Goodrich, and R. Tamassia. GeomNet: Geometric computing over the Internet. *IEEE Internet Computing*, 3(2):21–29, 1999.

[10] G. Barequet, S. S. Bridgeman, C. A. Duncan, M. T. Goodrich, and R. Tamassia. Classical computational geometry in GeomNet. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 412–414, 1997.

[11] L. Barnett, J. Casp, D. Green, and J. Kent. Design and implementation of an interactive tutorial framework. In *Proc. 29th SIGCSE Tech. Symp.*, pages 87–91, 1998.

[12] Wojciech Basalaj and Karen Eilbeck. Straight-line drawings of protein interactions. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 259–266. Springer-Verlag, 1999.

[13] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.

[14] M. Beccaria, P. Bertolazzi, G. Di Battista, and G. Liotta. A tailorable and extensible automatic layout facility. In *Proc. IEEE Workshop on Visual Languages*, pages 68–73, 1991.

[15] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 6(12):476–497, 1994.

[16] Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo. Computing orthogonal drawings with the minimum number of bends. In Frank Dehne, Andrew Rau-Chaplin, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proc. 5th Workshop Algorithms Data Struct.*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 331–344. Springer-Verlag, 1997.

[17] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings. *Comput. Geom. Theory Appl.*, 9:159–180, 1998.

[18] T. C. Biedl and M. Kaufmann. Area-efficient static and incremental graph drawings. In R. Burkard and G. Woeginger, editors, *Algorithms (Proc. ESA '97)*, volume 1284 of *Lecture Notes Comput. Sci.*, pages 37–52. Springer-Verlag, 1997.

[19] Therese Biedl, Joe Marks, Kathy Ryall, and Sue Whitesides. Graph multidrawing: Finding nice drawings without defining nice. In S. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 347–355. Springer-Verlag, 1998.

[20] Therese C. Biedl. New lower bounds for orthogonal drawings. *J. Graph Algorithms Appl.*, 2(7):1–31, 1998.

[21] Therese C. Biedl, Brendan P. Madden, and Ioannis G. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 391–402. Springer-Verlag, 1997.

[22] Blackboard Inc. `www.blackboard.com`.

[23] J. D. Boissonnat, F. Cazals, and J. Flötotto. 2D-structure drawings of similar molecules. In Joe Marks, editor, *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 115–126. Springer-Verlag, 2001.

[24] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Macmillan, London, 1976.

[25] C. Boroni, F. Goosey, M. Grinder, J. Lambert, and R. Ross. Tying it all together: Creating self-contained, animated, interactive, web-based resources for computer science education. In *Proc. 30th SIGCSE Tech. Symp.*, pages 7–11, 1999.

[26] C. Boroni, F. Goosey, M. Grinder, and R. Ross. Weblab! A universal and interactive teaching, learning, and laboratory environment for the World Wide Web. In *Proc. 28th SIGCSE Tech. Symp.*, pages 199–203, 1997.

[27] C. Boroni, F. Goosey, M. Grinder, and R. Ross. A paradigm shift! The Internet, the Web, browsers, Java, and the future of computer science education. In *Proc. 30th SIGCSE Tech. Symp.*, pages 145–152, 1999.

[28] U. Brandes and D. Wagner. Dynamic grid embedding with few bends and changes. In *Proceedings of the 9th Annual International Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *Lecture Notes in Computer Science*, pages 89–98. Springer-Verlag, 1998.

[29] Ulrik Brandes, Patrick Kenis, and Dorothea Wagner. Centrality in policy network drawings. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 250–258. Springer-Verlag, 1999.

[30] Ulrik Brandes, Galina Shubina, Roberto Tamassia, and Dorothea Wagner. Fast layout methods for timetable graphs. In Joe Marks, editor, *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 127–138. Springer-Verlag, 2001.

[31] Ulrik Brandes and Dorothea Wagner. A bayesian paradigm for dynamic graph layout. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 236–247. Springer-Verlag, 1997.

[32] Ulrik Brandes and Dorothea Wagner. Using graph layout to visualize train interconnection data. In Sue H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[33] S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Computational Geometry: Theory and Applications*, 16(1):53–93, 2000.

[34] S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 45–52. Springer-Verlag, 1997.

[35] S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. *Internat. J. Comput. Geom. Appl.*, 9(4/5):419–446, 1999.

[36] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *Proc. ACM Tech. Symp. on Computer Science Education (SIGCSE)*, pages 139–143, 2000.

[37] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In Sue H. Whitesides, editor, *Graph Drawing (Proceedings of GD '98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag.

[38] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *Journal of Graph Algorithms and Applications*, 4(3):47–74, 2000.

[39] Stina Bridgeman, Giuseppe Di Battista, Walter Didimo, Guiseppe Liotta, Roberto Tamassia, and Luca Vismara. Turn-regularity and optimal drawings of orthogonal representations. In *Abstracts 15th European Workshop Comput. Geom.*, pages 161–164. INRIA Sophia-Antipolis, 1999.

[40] Stina Bridgeman, Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Roberto Tamassia, and Luca Vismara. Optimal compaction of orthogonal representations. In *CGC Workshop on Geometric Computing*, 1998.

[41] Stina Bridgeman, Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Roberto Tamassia, and Luca Vismara. Optimal compaction of orthogonal representations. In *Graph Drawing (Proc. GD '99)*, Lecture Notes Comput. Sci. Springer-Verlag, 1999.

[42] Stina Bridgeman and Roberto Tamassia. A user study in similarity measures for graph drawing. In Joe Marks, editor, *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2001.

[43] Stina S. Bridgeman, Jody Fanto, Ashim Garg, Roberto Tamassia, and Luca Vismara. InteractiveGiotto: An algorithm for interactive orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 303–308. Springer-Verlag, 1997.

[44] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[45] Andrea Carmignani, Giuseppe Di Battista, Walter Didimo, Francesco Matera, and Maurizio Pizzonia. Visualization of the autonomous systems interconnections with HERMES. In Joe Marks, editor, *Graph Drawing (Proc. GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 150–163. Springer-Verlag, 2001.

[46] Jacobo Carrasquel. Teaching CS1 on-line: the good, the bad, and the ugly. In *Proc. 30th SIGCSE Tech. Symp.*, pages 212–216, 1999.

[47] L. P. Chew, M. T. Goodrich, D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. *Comput. Geom. Theory Appl.*, 7:113–124, 1997.

[48] R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar ST-digraphs. *SIAM J. Comput.*, 24(5):970–1001, 1995.

[49] M. P. Consens, F. C. Eigler, M. Z. Hasan, A. O. Mendelzon, E. G. Noik, A. G. Ryman, and D. Vista. Architecture and applications of the HY+ visualization system. *IBM Syst. J.*, 33:458–476, 1994.

[50] M. P. Consens, A. O. Mendelzon, and A. G. Ryman. Visualizing and querying software structures. In *Proc. 14th Intl. Conference on Software Engineering*, pages 138–156, 1992.

[51] E. Dengler, M. Friedell, and J. Marks. Constraint-driven diagram layout. In *Proc. IEEE Sympos. on Visual Languages*, pages 330–335, 1993.

[52] Edmund Dengler and William Cowan. Human perception of laid-out graphs. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 441–443. Springer-Verlag, 1998.

[53] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.

[54] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing.* Prentice Hall, Upper Saddle River, NJ, 1999.

[55] G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed acyclic graphs: An experimental study. *Int. J. Comput. Geom. Appl.*, 10(6):623–648, 2000.

[56] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of three graph drawing algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 306–315, 1995.

[57] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–325, 1997.

[58] G. Di Battista, A. Giammarco, G. Santucci, and R. Tamassia. The architecture of Diagram Server. In *Proc. IEEE Workshop on Visual Languages*, pages 60–65, 1990.

[59] G. Di Battista and G. Liotta. Upward planarity checking: Faces are more than polygons. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 72–86. Springer-Verlag, 1998.

[60] G. Di Battista, G. Liotta, and F. Vargiu. Diagram Server. *J. Visual Lang. Comput.*, 6(3):275–298, 1995. (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).

[61] G. Di Battista, G. Liotta, and F. Vargiu. Spirality and optimal orthogonal drawings. *SIAM J. Comput.*, 27(6):1764–1811, 1998.

[62] W. Didimo and G. Liotta. Computing orthogonal drawings in a variable embedding setting. In K.-Y. Chwa and O. H. Ibarra, editors, *Algorithms and Computation (Proc. ISAAC '98)*, volume 1533 of *Lecture Notes Comput. Sci.*, pages 79–88. Springer-Verlag, 1998.

[63] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.

[64] Ala Eddine Barouni, Ali Jaoua, and Nejib Zaguia. Visualizing algorithms for the design and analysis of survivable networks. In J. Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 232–241. Springer-Verlag, 1999.

[65] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[66] U. Erlingson and M. Krishnamoorthy. Interactive graph drawing on the World Wide Web. manuscript, Dept. of Comput. Sci., Rensselaer Polytechnic Institute, 1996. `http://www.cs.rpi.edu/projects/pb/graphdraw`.

[67] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996. `http://www.cgal.org`.

[68] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. Semnet: Three-dimensional graphic representation of large knowledge bases. In R. Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction*, pages 201–233. Lawrence Erlbaum Associates, 1988.

[69] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1–2):193–213, 1995.

[70] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, June 1991.

[71] U. Fößmeier, C. Hess, and M. Kaufmann. On improving orthogonal drawings: the 4M-algorithm. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 125–137. Springer-Verlag, 1998.

[72] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 254–266. Springer-Verlag, 1996.

[73] Ulrich Fößmeier. Interactive orthogonal graph drawing: Algorithms and bounds. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 111–123. Springer-Verlag, 1997.

[74] Ulrich Fößmeier and Michael Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 134–145. Springer-Verlag, 1997.

[75] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.

[76] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. Graph visualization in software analysis. In *Proc. IEEE Symposium on Assessment of Quality Software Development Tools*, May 1992.

[77] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

[78] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds. In *Proc. 2nd Annu. European Sympos. Algorithms*, volume 855 of *Lecture Notes Comput. Sci.*, pages 12–23. Springer-Verlag, 1994.

[79] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 286–297. Springer-Verlag, 1995.

[80] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 201–216. Springer-Verlag, 1997.

[81] N. Gelfand and R. Tamassia. Algorithmic patterns for orthogonal graph drawing. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 138–152. Springer-Verlag, 1998.

[82] J. E. Goodman and R. Pollack. Multidimensional sorting. *SIAM J. Comput.*, 12(3):484–507, August 1983.

[83] M. T. Goodrich, Joseph S. B. Mitchell, and M. W. Orletsky. Approximate geometric pattern matching under rigid motion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(4):371–379, April 1999.

[84] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[85] Leonidas Guibas and David Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 190–199, 1995.

[86] Leonidas J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[87] Y. Hara, A. Keller, P. Rathmann, and G. Wiederhold. Implementing hypertext database relationships through aggregations and exceptions. Technical Report STAN-CS-91-1381, Stanford University, 1981.

[88] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[89] M. Hasan, D. Vista, and A. Mendelzon. Visual Web surfing with HY+. In *Proc. CASCON'95*, 1995. Available from `http://www.db.toronto.edu:8020/`.

[90] M. Himsolt. GraphEd: a graphical platform for the implementation of graph algorithms. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 182–193. Springer-Verlag, 1995.

[91] M. Himsolt. GML: Graph modelling language. Manuscript, Universität Passau, Innstraße 33, 94030 Passau, Germany, 1996. `http://infosun.fmi.uni-passau.de/Graphlet/GML/`.

[92] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 106–117, 1988.

[93] F. Hoffmann and K. Kriegel. Embedding rectilinear graphs in linear time. *Inform. Process. Lett.*, 29:75–79, 1988.

[94] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[95] M. Y. Hsueh. *Symbolic Layout and Compaction of Integrated Circuits.* PhD thesis, Berkeley, CA, 1980.

[96] M. Y. Hsueh and D. O. Pederson. Computer-aided layout of LSI circuit building-blocks. In *Proc. IEEE Int. Symp. on Circuits and Systems*, 1979.

[97] K. Imai, S. Sumino, and H. Imai. Minimax geometric fitting of two corresponding sets of points. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 266–275, 1989.

[98] S. Isoda, T. Shimonmura, and Y. Ono. Vips: A visual debugger. *IEEE Softw.*, 4(3):8–19, 1987.

[99] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Tech. Symp.*, pages 335–339, 1997.

[100] D. Kimelman, B. Leban, T. Roth, and D. Zernik. Reduction of visual complexity in dynamic graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 218–225. Springer-Verlag, 1995.

[101] D. Kimelman and G. Sangudi. Program visualization by integration of advanced compiler technology with configurable views. In J. J. Dongarra and B. Tourancheau, editors, *Proc. CNRS-NSF Collaboration Workshop on Environments and Tools for Parallel Scientific Computing, Saint Hilaire du Touvert, France*, pages 73–84. Elsevier Science, September, 1992.

[102] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In G. Cornuejols, R. E. Burkard, and G. J Woeginger, editors, *Integer Programming and Combinatorial Optimization (Proc. IPCO '99)*, volume 1610 of *Lecture Notes Comput. Sci.*, pages 304–319. Springer-Verlag, 1999.

[103] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. Technical Report MPI-I-98-1-031, Max Planck Institut für Informatik, Saarbrücken, Germany, December 1998.

[104] Gunnar W. Klau and Petra Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max Planck Institut für Informatik, Saarbrücken, Germany, 1998.

[105] E. Koutsofios and S. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ., 1995. Available from `http://www.research.bell-labs.com/dist/drawdag`.

[106] M.S. Krishnamoorthy, F. Oxaal, U. Dogrusoz, D. Pape, A. Robayo, A. Koyanagi, Y. Hsu, D. Hollinger, and A. Hashemi. GraphPack: Design and features. *World Scientific: Scientific Visualization*, 1996.

[107] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. B. G. Teubner, 1990.

[108] Kelly A. Lyons, Henk Meijer, and David Rappaport. Algorithms for cluster busting in anchored graph drawing. *J. Graph Algorithms Appl.*, 2(1):1–24, 1998.

[109] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, Sept. 1993.

[110] David Mason and Denise Woit. Integrating technology into computer science examinations. In *Proc. 29th SIGCSE Tech. Symp.*, pages 140–144, 1998.

[111] David Mason and Denise Woit. Providing mark-up and feedback to students with online marking. In *Proc. 30th SIGCSE Tech. Symp.*, pages 3–6, 1999.

[112] Kurt Mehlhorn and Stefan Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.

[113] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[114] K. Miriyala, S. W. Hornick, and R. Tamassia. An incremental approach to aesthetic graph layout. In *Proc. Internat. Workshop on Computer-Aided Software Engineering*, 1993.

[115] K. Misue, Peter Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *J. Visual Lang. Comput.*, 6(2):183–210, 1995.

[116] S. Moen. Drawing dynamic trees. *IEEE Softw.*, 7:21–28, 1990.

[117] S. Mukherjea. Visualizing the information space of hypermedia systems. Technical report. `http://www.cc.gatech/edu/gvu/people/Phd/sougata/Nvb.html`.

[118] S. Mukherjea, J. Foley, and S. Hudson. Visualizing complex hypermedia networks through multiple hierarchicial views. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, 1995.

[119] S. North. 5114 directed graphs, 1995. Manuscript.

[120] S. North. Incremental layout in DynaDAG. In *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 409–418. Springer-Verlag, 1996.

[121] S. North and E. Koutsofios. Applications of graph visualization. Technical report, AT&T Bell Laboratories, Murray Hill, NJ., 1995.

[122] R. H. J. M. Otten and J. G. van Wijk. Graph representations in interactive layout design. In *Proc. IEEE Internat. Sympos. on Circuits and Systems*, pages 914–918, 1978.

[123] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, May 1996.

[124] A. Papakostas and I. G. Tollis. Improved algorithms and bounds for orthogonal drawings. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 40–51. Springer-Verlag, 1995.

[125] A. Papakostas and I. G. Tollis. Interactive orthogonal graph drawing. *IEEE Trans. Comput.*, C-47(11):1297–1309, 1998.

[126] Achilleas Papakostas, Janet M. Six, and Ioannis G. Tollis. Experimental and theoretical results in interactive graph drawing. In Stephen North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 371–386. Springer-Verlag, 1997.

[127] Achilleas Papakostas and Ioannis G. Tollis. Algorithms for area-efficient orthogonal drawings. *Comput. Geom. Theory Appl.*, 9(1–2):83–110, 1998. Special Issue on Geometric Representations of Graphs, G. Di Battista and R. Tamassia, editors.

[128] M. Patrignani. On the complexity of orthogonal compaction. In F. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors, *Algorithms and Data Structures (Proc. WADS '99)*, volume 1663 of *Lecture Notes Comput. Sci.*, pages 56–61. Springer-Verlag, 1999.

[129] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. 8th Annu. ACM Conf. Object-Oriented Program. Syst. Lang. Appl.*, volume 28 of *SIGPLAN Notices*, pages 326–337, 1993.

[130] W. Pierson and S. Rodger. Web-based animation of data structures using JAWAA. In *Proc. 29th SIGCSE Tech. Symp.*, pages 267–271, 1998.

[131] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 435–446. Springer-Verlag, 1996.

[132] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 248–261. Springer-Verlag, 1998.

[133] S. P. Reiss. 3-D visualization of program information. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 12–24. Springer-Verlag, 1995.

[134] S. P. Reiss. An engine for the 3D visualization of program information. *J. Visual Lang. Comput.*, 6(3):299–323, 1995. (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).

[135] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Softw. – Pract. Exp.*, 17(1):61–76, 1987.

[136] K. Ryall, J. Marks, and S. Shieber. An interactive system for drawing graphs. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 387–393. Springer-Verlag, 1997.

[137] S. Singh. Documentation for Paren-to-GDS. Manuscript, Dept. of Comp. Sci., Brown University, 1991.

[138] J. M. Six, K. G. Kakoulis, and I. G. Tollis. Refinement of orthogonal graph drawings. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 302–315. Springer-Verlag, 1998.

[139] L. Spratt and A. Ambler. Using 3D tubes to solve the intersecting line representation problem. In *Proc. IEEE Symp. on Visual Languages, 1994*, pages 254–261, 1995.

[140] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[141] Björn Steckelbach, Till Bubeck, Ulrich Fößmeier, Michael Kaufmann, Marcus Ritt, and Wolfgang Rosenstiel. Visualization of parallel execution graphs. In Sue H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[142] L. Stockmeyer. Optimal orientation of cells in slicing floorplan design. *Inform. Control*, 57:91–101, 1983.

[143] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[144] R. Tamassia. New layout techniques for entity-relationship diagrams. In *Proc. 4th Internat. Conf. on Entity-Relationship Approach*, pages 304–311, 1985.

[145] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[146] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

[147] R. Tamassia, G. Liotta, and F. P. Preparata. Robust proximity queries in implicit Voronoi diagrams. In *Proc. 8th Canad. Conf. Comput. Geom.*, page 1, 1996.

[148] R. Tamassia and I. G. Tollis. Planar grid embedding in linear time. *IEEE Trans. on Circuits and Systems*, CAS-36(9):1230–1234, 1989.

[149] R. Tamassia, I. G. Tollis, and J. S. Vitter. Lower bounds for planar orthogonal drawings of graphs. *Inform. Process. Lett.*, 39:35–40, 1991.

[150] L. Tinoco, E. Fox, and D. Barnette. Online evaluation in WWW-based courseware. In *Proc. 28th SIGCSE Tech. Symp.*, pages 194–198, 1997.

[151] A. van Hoff, S. Shaio, and O. Starbuck. Hooked on Java. 1996.

[152] G. Vijayan and A. Wigderson. Rectilinear graphs and their embeddings. *SIAM J. Comput.*, 14:355–372, 1985.

[153] WebCT, Inc. `www.webCT.com`.

[154] Wayne A. Wickelgren. *Cognitive Psychology.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

[155] W. Wimer, I. Koren, and I. Cederbaum. Floorplans, planar graphs and layouts. *IEEE Trans. Circuits Syst.*, CAS-35:267–278, 1988.

[156] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40(1):2–18, 1990.