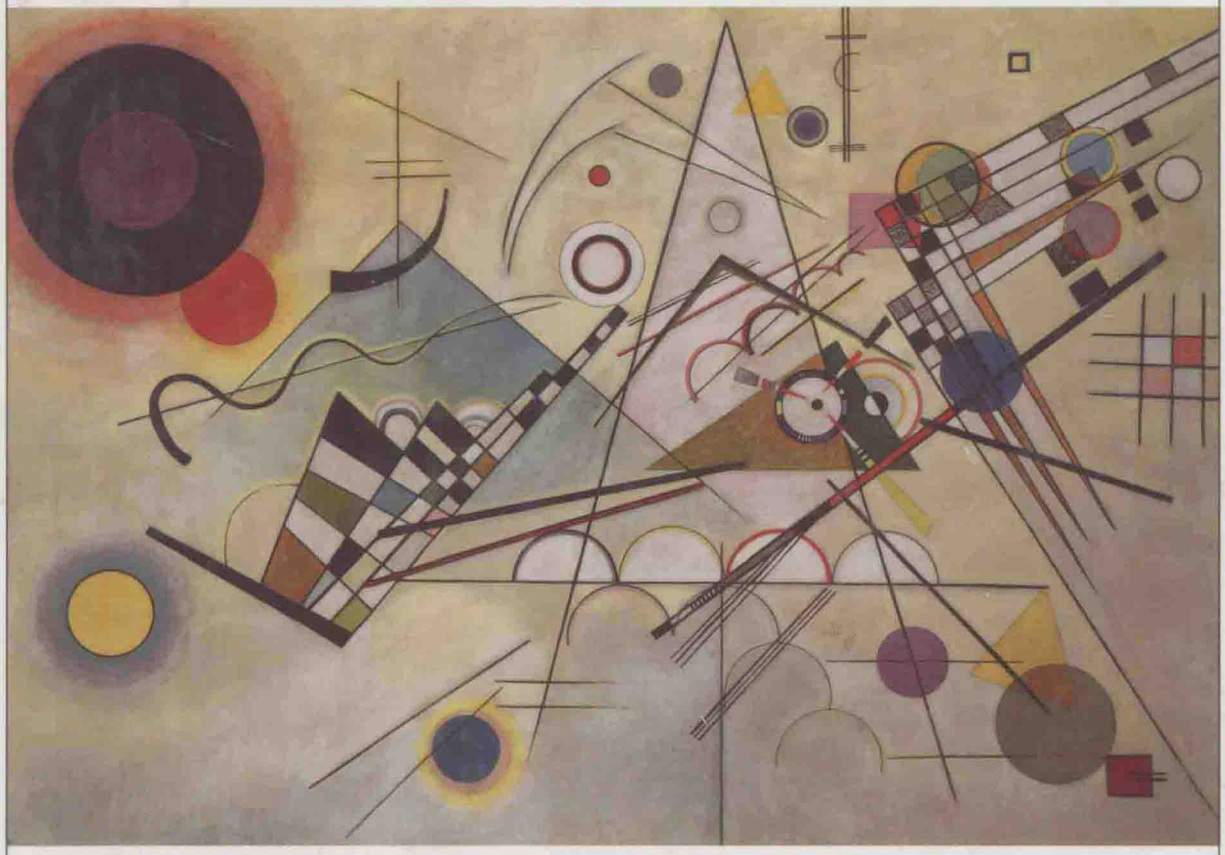
GRAPH DRAWING

ALGORITHMS FOR THE VISUALIZATION OF GRAPHS



GIUSEPPE DI BATTISTA

PETER EADES

ROBERTO TAMASSIA

IOANNIS G. TOLLIS

Graph Drawing

Algorithms for the Visualization of Graphs

Giuseppe Di Battista
Dipartimento di Informatica e Automazione
Università degli Studi di Roma Tre

Peter Eades
Department of Computer Science
University of Newcastle

Roberto Tamassia
Department of Computer Science
Brown University

Ioannis G. Tollis
Department of Computer Science
The University of Texas at Dallas

An Alan R. Apt Book

PRENTICE HALL, *Upper Saddle River, New Jersey 07458*

Library of Congress Cataloging-in-Publication Data

Graph drawing: algorithms for the visualization of graphs /
Giuseppe Di Battista . . . [et al.]
p. cm.
"An Alan R. Apt Book"
Includes bibliographical references and index.
ISBN: 0-13-301615-3 (alk. paper)
1. Computer graphics. 2. Graph theory. I. Di Battista, Giuseppe.
T385.G6934 1999
511'.3—dc21 98-19177
CIP

Publisher: Alan Apt
Editor: Laura Steele
Production Editor: Edward DeFelippis
Editor-In-Chief: Marcia Horton
Managing Editor: Eileen Clark
Assistant Vice President of Production and Manufacturing: David W. Riccardi
Art Director: Jayne Conte
Cover Designer: Anthony Gemmellaro
Copy Editor: Jerri Uzzo
Manufacturing Buyer: Pat Brown
Editorial Assistant: Kate Kaibni

Printed in the United States of America

10 9 8 7 6 5 4 3 2

ISBN: 0-13-301615-3

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Preface

The visualization of complex conceptual structures is a key component of support tools for many applications in science and engineering. A graph is an abstract structure that is used to model information. Graphs are used to represent information that can be modeled as objects and connections between those objects. Hence, many information visualization systems require graphs to be drawn so that they are easy to read and understand. In this book, we describe algorithms for automatically generating clear and readable diagrams of complex conceptual structures.

Graph Drawing

Graph drawing addresses the problem of constructing geometric representations of graphs, networks, and related combinatorial structures. Geometric representations of graphs have been investigated by mathematicians for centuries, for visualization and intuition, as well as for the pure beauty of the interplay between graph theory and geometry. In the 1960s, computer scientists began to use graph drawings as diagrams to assist with the understanding of software. Knuth's 1963 paper on drawing flowcharts [Knu63] was perhaps the first paper to present an algorithm for drawing a graph for visualization purposes.

Today, the automatic generation of drawings of graphs finds many applications. Examples include software engineering (data flow diagrams, subroutine-call graphs, program nesting trees, object-oriented class hierarchies), databases (entity-relationship diagrams), information systems (organization charts), real-time systems (Petri nets, state-transition diagrams), decision support systems (PERT networks, activity trees), VLSI (circuit schematics), artificial intelligence (knowledge-representation diagrams), and logic programming (SLD-trees). Further applications can be found in other science and engineering disciplines, such as medical science (concept lattices), biology (evolutionary trees), chemistry (molecular drawings), civil

engineering (floorplan maps), and cartography (map schematics).

Because of the combinatorial and geometric nature of the problems investigated, and the wide range of the application domains, research in graph drawing has been conducted within several diverse areas, including discrete mathematics (topological graph theory, geometric graph theory, order theory), algorithmics (graph algorithms, data structures, computational geometry, VLSI), and human-computer interaction (visual languages, graphical user interfaces, software visualization). A bibliography on graph drawing algorithms [DETT94] cites more than 300 papers. In addition, a large body of related nonalgorithmic literature exists on geometric graph theory, topological graph theory, and order theory.

Various graphic standards are used for drawing graphs. Usually, vertices are represented by symbols such as points or boxes, and edges are represented by simple open Jordan curves connecting the symbols that represent the associated vertices. However, the graphic standards may vary depending upon the application. For example, mathematicians seem to prefer straight-line drawings, where edges are straight-line segments, while circuit and database designers tend to use orthogonal drawings, where edges consist of horizontal and vertical segments. Within a graphic standard, a graph has infinitely many different drawings. The usefulness of a drawing of a graph depends on its readability, that is, the capability of conveying the meaning of the graph quickly and clearly. Readability issues can be expressed by means of aesthetic criteria, such as the minimization of crossings between edges, and the display of symmetries.

When drawing a graph, we would like to take into account a variety of aesthetic criteria. For example, planarity and the display of symmetries are often highly desirable in visualization applications. In general, in order to improve the readability of a drawing, it is important to keep the number of bends and crossings low. Also, to avoid wasting space on a page or a computer screen, it is important to keep the area of the drawing small, subject to resolution rules. In this scenario, many graph drawing problems can be formalized as multi-objective optimization problems (e.g., construct a drawing with minimum area and minimum number of bends and crossings). Trade-offs are often necessary in order to solve these problems.

The purpose of this book is to describe fundamental algorithmic techniques for constructing drawings of graphs.

Organization of the Book

This book is organized as follows:

In Chapter 1, we review the terminology of graphs and their drawings.

Chapter 2 presents general graph drawing methods which use the algorithms presented in the following chapters as building blocks. It provides guidelines for employing the technical material of the book in the design of graph drawing algorithms and systems.

Divide and conquer is an evergreen paradigm in computer science. In Chapter 3, we apply this technique to draw trees and series-parallel digraphs. Further, we show how to test the planarity of a graph using the divide and conquer paradigm.

Chapter 4 presents techniques for constructing various types of drawings of planar graphs. These techniques can also be used for drawing nonplanar graphs by means of a preliminary planarization step.

In Chapter 5, we present methods based on network flow. These methods construct a planar orthogonal drawing of an embedded planar graph, with the minimum number of bends.

Flow techniques are used again in Chapter 6 to address the upward planarity testing problem for digraphs. The study of upward planarity has fascinating connections with fundamental graph-theoretic and order-theoretic properties.

Incremental techniques are presented in Chapter 7. We apply these techniques to the graph planarization problem, and also use them to design algorithms suitable for interactive systems.

In Chapter 8, we focus on constructing orthogonal grid drawings of nonplanar graphs. The presented techniques are based first on orienting a given graph, and then drawing it one vertex at a time, following the order of the orientation.

Chapter 9 presents the hierarchical approach for creating polyline drawings of digraphs with vertices arranged in horizontal layers. This approach is highly intuitive and can be applied to any digraph.

Chapter 10 presents several techniques that take a graph as input and simulate a system of forces reflecting user preferences. A straight-line drawing results from an equilibrium configuration of the force system.

In Chapter 11, we present techniques for investigating the intrinsic limits of graph drawing algorithms, both in terms of the quality of the output and in terms of computational resources required.

In Appendix A, we tabulate upper and lower bounds on properties of drawings of graphs, and discuss trade-offs between such properties.

Use of the book

The reader is expected to be familiar with basic algorithms and data structures. This book is primarily written for three audiences.

- It can be used as a text in an advanced undergraduate or graduate course in graph drawing. It can also provide material for courses that devote part of their attention to graph drawing; these include computational geometry, graph algorithms, and information visualization.

- It provides researchers with techniques that cover the main themes of the graph drawing area. Also, the chapters are relatively self contained, allowing for independent reading.

- Engineers involved in creating user interfaces can use this book as a fundamental source for effective and practical graph drawing methods.

Most chapters end with several exercises and problems. Many of them are devoted simply to testing the level of knowledge of the material contained in the chapter. Some exercises require more thought, and some are suitable for advanced courses.

Acknowledgements

In 1987, when we started planning to write this book, we had no idea that research in graph drawing would grow as fast as we have seen in recent years. We also had no idea of the effort required to write a systematic description of such a dynamic field.

Many people gave us feedback on earlier versions of this book. Special thanks go to: Juergen Branke, Stina Bridgeman, Isabel Cruz, Walter Didimo, Joel Fenwick, Arne Frick, Tom Frisinger, Patrick Garvan, Mike Goodrich, Seokhee Hong, Konstantinos Kakoulis, Michael Kaufmann, Joseph LaViola, Xuemin Lin, Giuseppe Liotta, Brian Lucena, Kazuo Misue, Justin Monti, Petra Mutzel, Achilleas Papakostas, Maurizio Patrignani, Maurizio Pizzonia, Aaron Quigley, Galina Shubina, Janet Six, Robin Stacey, Jennifer Stewart, Chensu Sun, Jiankuan Sun, David Thomson, Francesco Vargiu, Luca Vismara, Paola Vocca, Sue Whitesides, and Weisheng Xu.

We acknowledge the support received from the Australian Research Council, the Esprit Project Alcom-IT, the National Institute of Standards and Technology, the National Science Foundation, the Texas Advanced Research

Program, the US Army Research Office, Fujitsu Laboratories, Brown University, the University of Newcastle, the University of Texas at Dallas, and the Third University of Rome.

Finally, this work would not have been possible without the encouragement and support of Alexandra, Caitriona, Cristina, Elvira, Isabel, Manfredo, Patrick, Pietro, Rossana, and Shelly.

Giuseppe Di Battista,
Peter Eades,
Roberto Tamassia,
Ioannis G. Tollis.

Contents

Chapter 1

Graphs and Their Drawings

Relational structures, consisting of a set of entities and relationships between those entities, are ubiquitous in computer science. Such structures are usually modeled as *graphs*: the entities are *vertices*, and the relationships are *edges*. For example, most tools in software engineering use graphs to model the dependency relationships between modules in a large program. A module is represented as a vertex in a graph, and the dependency of module a on module b is represented by an edge from a to b. These graphs are typically drawn as diagrams with text at the vertices and line segments joining the vertices as edges. The example in Figure 1.1 represents the dependencies between some of the modules in `Xwindows`. In the example in Figure 1.2, the vertices represent documents in a hypertext system and the edges represent hyperlinks between the documents.

Visualizations of relational structures are only useful to the degree that the associated diagrams effectively convey information to the people that use them. A good diagram helps the reader understand the system, but a poor diagram can be confusing and misleading.

For example, consider the two diagrams in Figure 1.3. Both diagrams represent a simple class hierarchy; vertices represent classes of geometric shapes, and edges describe the *is-a relation*. Here each vertex represents a class, and a directed edge between two vertices represents the class-subclass relationship. Figure 1.3.a is more difficult to follow than Figure 1.3.b. This book is about *graph drawing algorithms*, that is, methods to produce graph drawings which are easy to follow.

The main purpose of this introductory chapter is to define the basic concepts for graphs and graph drawings. Related material is available in many textbooks:

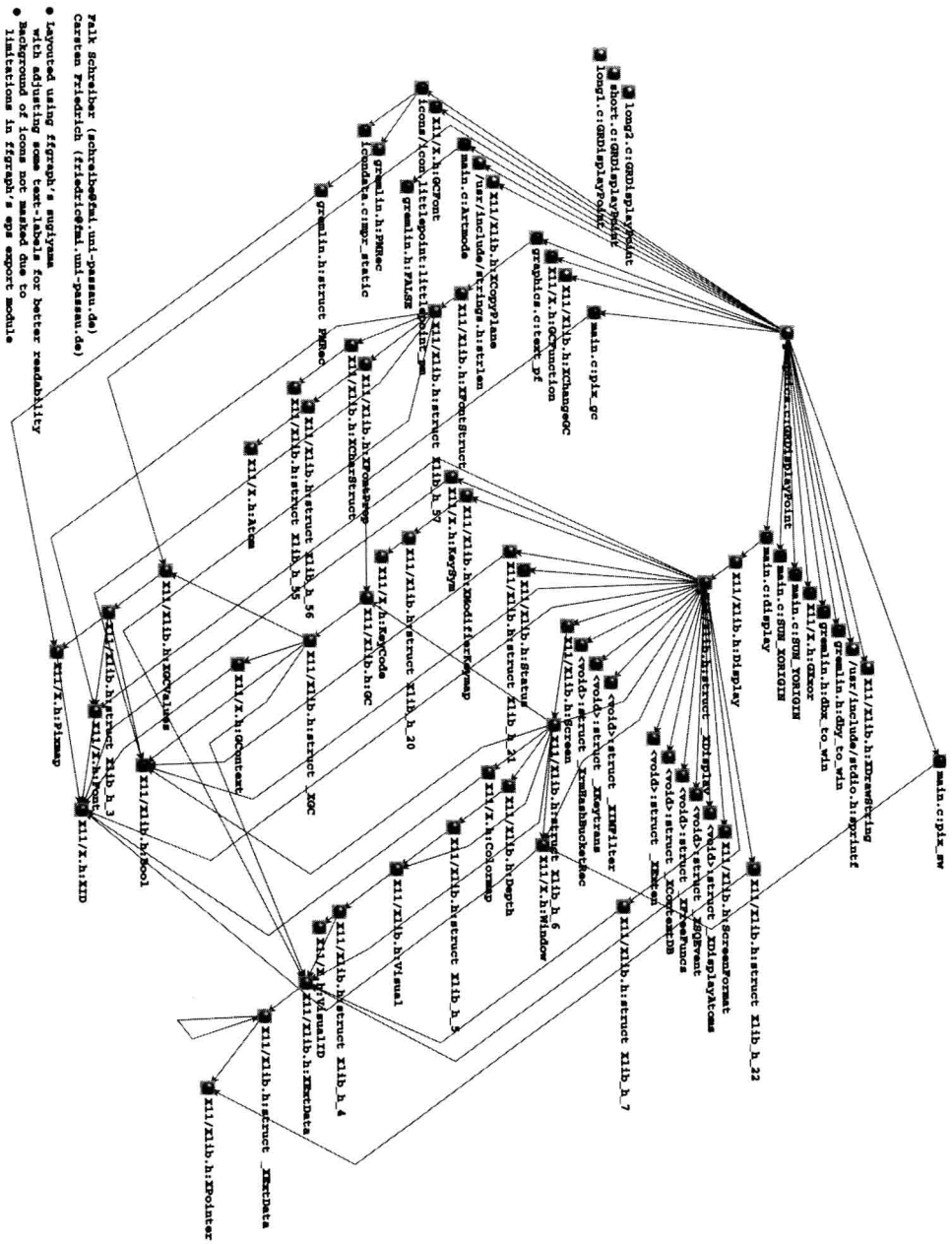


Figure 1.1: Dependency graph for some of the modules of **Xwindows**. (Courtesy of F. Schreiber and C. Friedrich.)

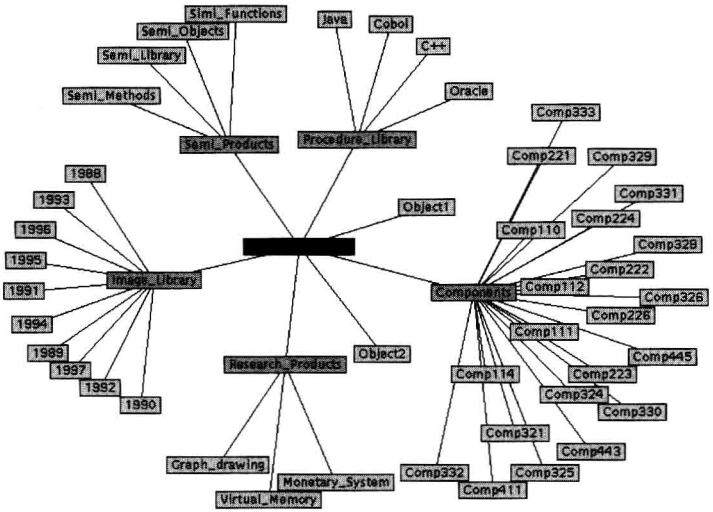


Figure 1.2: A graph representing hypertext documents and links between them. (Courtesy of M. Huang.)

- Graph theory is described in [BM76, Har72].
- Graph algorithms are illustrated in [Eve79, Gib80, Meh84, NC88, Tar83].
- There are many textbooks describing basic data structures and algorithms, for example, [CLR90, GT98]. Also, a reference book for computational complexity is [GJ79].
- Computational geometry provides a good background for many graph drawing methods (see [PS85]).

A *graph* $G = (V, E)$ consists of a finite set V of *vertices* and a finite multiset E of *edges*, that is, unordered pairs (u, v) of vertices. The vertices of a graph are sometimes called *nodes*; edges are sometimes called *links*, *arcs*, or *connections*.

An edge (u, v) with $u = v$ is a *self-loop*. An edge which occurs more than once in E is a *multiple edge*. A *simple graph* has no self-loops and no multiple edges. Most of this book deals with simple graphs, and unless otherwise specified, we assume that graphs are simple.

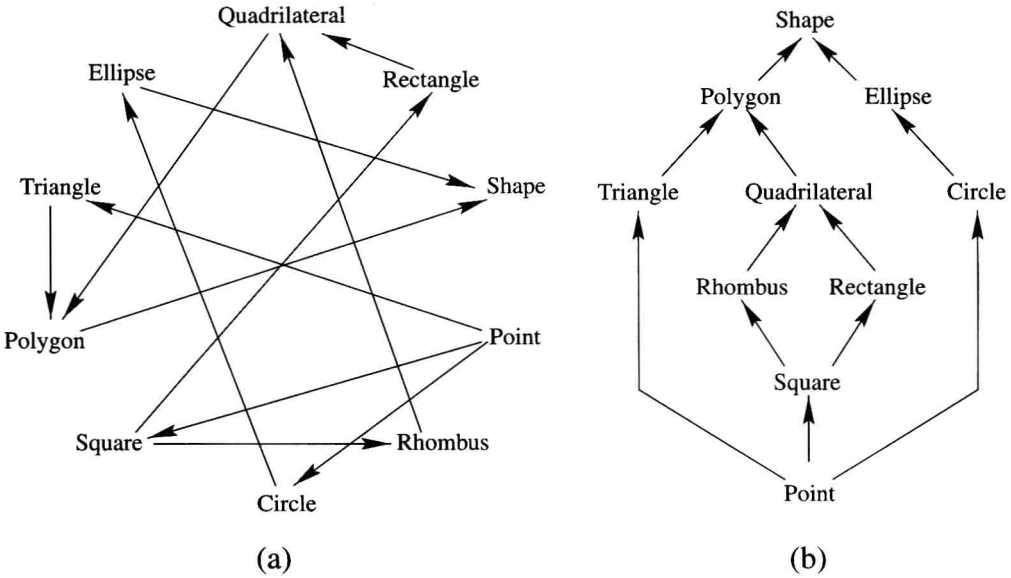


Figure 1.3: Two drawings of a class hierarchy.

The *end-vertices* of an edge $e = (u, v)$ are u and v; we say that u and v are *adjacent* to each other and e is *incident* to u and v. The *neighbors* of v are its adjacent vertices. The *degree* of v is the number of its neighbors.

A *directed graph* (or *digraph*) is defined similarly to a graph, except that the elements of E, called *directed edges*, are ordered pairs of vertices. The directed edge (u, v) is an *outgoing edge* of u and an *incoming edge* of v. Vertices without outgoing (resp. incoming) edges are called *sinks* (resp. *sources*). The *indegree* (resp. *outdegree*) of a vertex is the number of its incoming (resp. outgoing) edges.

A (directed) *path* in a (directed) graph $G = (V, E)$ is a sequence $(v_1, v_2, \ldots, v_h)$ of distinct vertices of G, such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq h - 1$. A (directed) path is a (directed) *cycle* if $(v_h, v_1) \in E$. A directed graph is *acyclic* if it has no directed cycles.

An edge (u, v) of a digraph is *transitive* if there is a directed path from u to v that does not contain the edge (u, v). The *transitive closure* G' of a digraph G has an edge (u, v) for every path from u to v in G. In many applications, a digraph conveys the same information as its transitive closure. For example, since class inheritance is transitive, the class diagram in Figure 1.4 contains the same information as the one in Figure 1.3. However,

as Figure 1.4 shows, transitive edges can clutter a graph drawing and cause confusion. In general, for many digraphs it is better to draw a *reduced digraph* (also called *transitive reduction*), that is, a digraph with no transitive edges. Figure 1.3 shows the reduced digraph of the digraph in Figure 1.4. Many of the algorithms in this book deal with reduced digraphs.

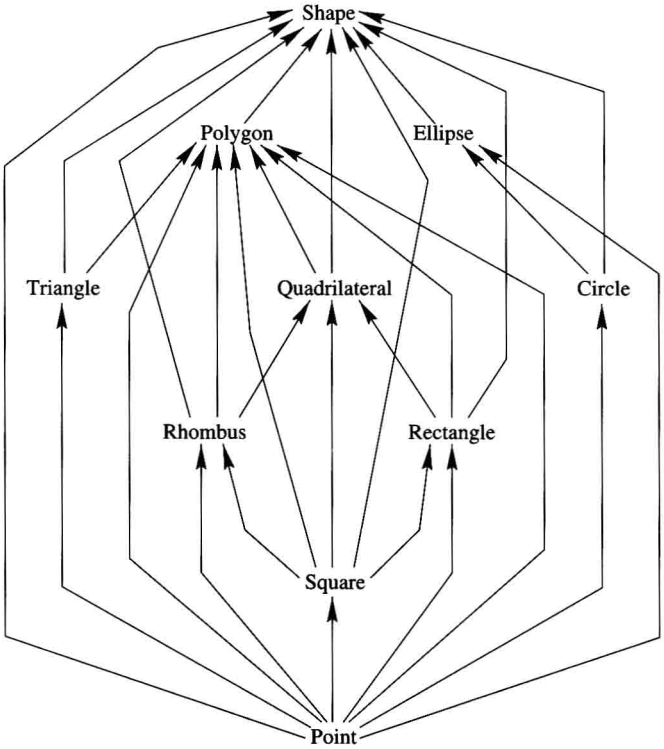


Figure 1.4: The transititive closure of the class hierarchy in Figure 1.3.

A graph $G' = (V', E')$, such that $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$, is a *subgraph* of graph $G = (V, E)$. If $E' = E \cap (V' \times V')$ then G' is *induced* by V'.

A graph $G = (V, E)$ with n vertices may be described by a $n \times n$ *adjacency matrix* A whose rows and columns correspond to vertices, with $A_{uv} = 1$ if $(u, v) \in E$ and $A_{uv} = 0$ otherwise. Table 1.1 gives a description of a graph (let us call it G_1) as an adjacency matrix.

Another way to describe a graph is by giving a list L_u of edges incident to vertex u for each $u \in V$. A description of a directed graph (let us call it G_2) in this format appears in Table 1.2.