Università degli Studi di Roma “La Sapienza”
Dottorato di Ricerca in Ingegneria Informatica
XIII Ciclo – 2001–3

Visualization of Large Graphs

Maurizio Patrignani
Università degli Studi di Roma “La Sapienza”

Dottorato di Ricerca in Ingegneria Informatica

XIII Ciclo - 2001–3

Maurizio Patrignani

Visualization of Large Graphs

Thesis Committee

Prof. Giuseppe Di Battista (Advisor)
Prof.ssa Paola Bertolazzi
Prof. Marco Schaefer
AUTHOR’S ADDRESS:
Maurizio Patrignani
Dipartimento di Informatica e Automazione
Università degli Studi di Roma “Roma Tre”
Via della Vasca Navale 79, 00146 Roma, Italy
E-MAIL: patrigna@dia.uniroma3.it
WWW: http://www.dia.uniroma3.it/~patrigna
Contents

I Problem Analysis 5

1 Motivations 7

2 Graph Drawing Background 11
   2.1 Graphs ................................................................. 11
   2.2 Drawings of Graphs .............................................. 14
   2.3 Orthogonal Drawing Convention ................................. 17
   2.4 Straight-Line Drawing Convention ............................... 19

3 Efficiency Problems 21
   3.1 The Topology-Shape-Metric Approach ......................... 21
   3.2 Force Directed Approaches ..................................... 23

4 Effectiveness Problems 25
   4.1 Drawing Conventions and Effectiveness Considerations .... 26

II Methodologies and
State of the Art 29

5 General Principles 31
   5.1 Hiding Principle .................................................... 31
   5.2 Stressing Principle .............................................. 32

6 Design Tradeoffs 35
   6.1 The Relationship Between Efficiency and Effectiveness .... 35
   6.2 General and Domain Specific Solutions ....................... 35
   6.3 Static Versus Interactive Scenarios ............................. 36
   6.4 Radical Versus Smooth Approaches ............................. 36

7 Strategies 39
   7.1 Windowing, Panning, and Zooming ............................. 39
7.2 Multiple Windows and Rapid Zooming .......................... 40
7.3 Warped Visualizations ............................................. 41
7.4 Clustering .......................................................... 42
7.5 Spanning Tree Reduction ........................................... 43
7.6 Pruning ............................................................. 43
7.7 Compacting ......................................................... 44
7.8 Exploiting the Three-Dimensional Environment ............ 44

8 Analysis of Existing Systems ....................................... 47
8.1 Cone Trees ......................................................... 47
8.2 Site Manager ....................................................... 48
8.3 NicheWorks ........................................................ 49
8.4 NestedVision3D ..................................................... 50
8.5 Other Research Projects and Prototypes ...................... 51

III The Efficiency Limits ................................................. 53

9 The Complexity of Orthogonal Compaction ....................... 55
9.1 Orthogonal Compaction ............................................ 55
9.2 NP-hardness of the OAC Problem ................................ 57
  9.2.1 Sliding-Rectangles Gadget .................................. 58
  9.2.2 Instance \((H_A,K_A)\) Construction Rules ............... 62
  9.2.3 Correctness .................................................. 66
9.3 The OAC Problem is in NP ....................................... 70

10 Related Problems ..................................................... 75
10.1 NP-Completeness of the OTELC and OMLEC Problems .... 75
10.2 Final Remarks ..................................................... 77

IV Stressing and Hiding in Three-Dimensional Graph Drawing .... 79

11 A Split&Push Approach to Three-Dimensional Orthogonal Drawing ......................................................... 81
  11.1 A Strategy for Constructing 3D Orthogonal Drawings .... 81
  11.2 Feasibility of the Approach .................................. 85

12 The Reduce-Forks Algorithm .......................................... 93
  12.1 Vertex Scattering .............................................. 93
  12.2 Direction Distribution ........................................ 95
  12.3 Vertex-Edge Overlap Removal and Crossing Removal ...... 97
13 Experimental Comparison of Three-Dimensional Orthogonal Drawing Methods 101
13.1 Algorithms Description ......................................................... 101
13.2 Graph Base Generation ......................................................... 104
13.3 Experimental Comparison .................................................... 105
13.4 Practical Usability of Three-Dimensional Orthogonal Drawing Algorithms ......................................................... 106

V Information Hiding in Specific Application Contexts 113

14 The Visualization of Network Partitions 115
14.1 The Network Partitioning Problem ........................................... 115
14.2 Interactive Partitioning .......................................................... 116
14.3 Visualizing Network Partitions ............................................... 117
     14.3.1 Choosing the Drawing Convention .................................. 117
     14.3.2 Induced Graph Visualization ........................................ 118
     14.3.3 Pairwise Representation ............................................... 119

15 A Visual Network Partitioning System 121
15.1 Interface and Interactive Exploration ..................................... 121
     15.1.1 Visualization Tuning .................................................. 121
15.2 User Operations ............................................................... 122
     15.2.1 Manual Editing ....................................................... 122
     15.2.2 Refinement Heuristics ............................................... 123

16 Drawing Infinite Trees 125
16.1 Introduction ........................................................................ 125
16.2 The Framework and the Drawing Strategies .............................. 127
16.3 Experimental Results ........................................................... 130
16.4 Analysis .............................................................................. 132
     16.4.1 Analysis of Leftmost-Embedding Algorithm ....................... 133
     16.4.2 Analysis of Central-Embedding Algorithm ......................... 136

17 Conclusions 139
Introduction

Two growing trends suggest a critical role for the information visualization research field in the near future: on one hand, due to the ubiquity of computer technologies, more and more massive data sets are available in electronic form; on the other hand, an increasing fraction of the users accessing these data are non-specialists.

Information visualization has precisely the key role of translating abstract information into geometric realities, allowing the user to easily understand, browse, and manipulate them in a natural and intuitive way, and its task becomes particularly critical as the amount of the data to be visualized increases.

The information can often be modeled as a graph. In fact, graphs are expressive enough to represent a wide range of structured information and they have a natural and intuitive visual representation which is effectively perceived by the user (evidence indicates that the relationship between a pair of objects is better perceived when it is represented through adjacency, rather than through size, proximity, color or shape [134]).

This work, devoted to scalability issues in graph drawing, is organized in five parts. The first part contains the motivations (Chapter 1), the essential graph drawing background (Chapter 2), and a description of the problems occurring when large data sets are involved in graph drawing. In particular, problems are classified into the two main categories of efficiency problems (Chapter 3) and effectiveness problems (Chapter 4).

The second part is devoted to the study of the possible solutions to the problems previously introduced and classified. Solutions are analyzed at various levels of abstraction: first, two general principles are given in Chapter 5; secondly, a set of trade-offs that should be considered in the visualization design is presented in Chapter 6; finally a collection of techniques exploiting in different ways one or both the general principles is presented in Chapter 7 (Strategies).

Chapter 8 closes this part by presenting an analysis of several existing applications, all corresponding to fully fledged industrial or commercial software systems. The analysis stresses how the general principles are used, how the
different techniques are adopted, and how the trade-offs are resolved to obtain visualization tools reaching far above the threshold of usability.

In the third part we show several NP-hardness results, indicating that pursuing the effectiveness of the representation by reducing the size of the drawing is not an efficient strategy, at least for orthogonal drawings. In particular, we show that compacting an orthogonal drawing with respect to the area is an NP-complete problem. Also, we show that the similar problems of compacting the orthogonal drawing with respect of the maximum edge length or total edge length are NP-complete problems too. Finally we show that even approaching the optimal solutions to these problems is as hard as actually finding them, or, more formally, that the three problems are not in PTAS, i.e., they don’t admit a polynomial time approximation scheme.

Since exploiting a three-dimensional environment seems to be a good strategy to visualize huge amount of data, in the fourth part we consider orthogonal drawings in three-dimensions. There is a rich and recent literature about this field but algorithms are generally devised with the purpose of meeting lower bounds, and tend, consequently, to produce drawings unsuitable for information visualization purposes. The approach Split&Push described in Chapter 11, was devised with the opposite purpose of producing readable three-dimensional orthogonal drawings, disregarding bounds and asymptotic performance. The method is based on generating the final drawing through a sequence of steps, starting from a “degenerate” drawing in which all the objects are collapsed in a single point. At each step the drawing “splits” into two pieces and finds a structure more similar to its final version.

In Chapter 13 we perform an experimental comparison of most of the three-dimensional orthogonal drawing methods found in literature. The purpose, other than evaluating their usability in information visualization applications, is that of validating the Reduce&Forks algorithm presented in Chapter 12 that follows the Split&Push approach. The comparison shows that no algorithm tested can meet both the requirements of minimum area and reduced number of bends, confirming the existence of a trade-off between the two measures, a concept that was suggested by several researchers, but still waits to be rigorously proven. Unfortunately, the drawings produced by the Split&Push algorithm, extremely readable when the graph is small, seem to loses their good qualities when the size of the input graph increases.

In the fifth part it is described how the hiding principle can be used to visualize large data set in two specific application contexts. The first application requires the representation of large networks partitions. The purpose is that of allowing the user to refine the partition by focusing the computer computational power towards intermediate goals, in the framework of a novel human-computer interaction paradigm called Human Guided Simple Search.
Chapter 14 describes the partitioning problem, the visualization requirements introduced by the interactive approach, and the solutions we devised, while Chapter 15 contains a description of the software system developed for the Mitsubishi Electric Research Laboratories.

The second application context that we consider is the exploration of a potentially infinite tree, of which only a neighborhood of the vertex on which the user attention is currently focused is represented. The visualization of the navigation of the Internet, which is considered to be a highly hierarchical graph, is the natural application of this model. When the user shifts the focus from a vertex to an adjacent one, the drawing is recomputed and vertices common to the two subsequent drawings may have different positions. Our study is meant to evaluate how changes on a simple drawing method (we use the Reingold and Tilford algorithm) improve the preservation of the user mental map by producing subsequent drawings in which common vertices have similar positions. The results we found are that even simple strategies may have considerable impact on the quality of the drawing method when used in a dynamic framework.

Our conclusions point out that, when visualizing large graphs, general and canonical approaches, such as the topology-shape-metric one, seem unsuitable for the purpose, and the adoption of the orthogonal drawing convention itself, otherwise so effective when small graphs are involved, has to be discouraged both in the two and in the three dimensions. On the other hand, solutions to domain specific problems exist and can be found, moving along the lines of general principles, using suitable techniques, and resolving trade-offs.

The problem of drawing large graphs is not susceptible to be solved in a simple and unique way, but it seems to have many different solutions as many application domains are considered, and each solution is an original recipe implying several choices regarding the use of general methodologies and techniques, and challenging the designer creativity and imagination.
Part I

Problem Analysis
Chapter 1

Motivations

Two growing trends suggest a critical role for the information visualization research field in the near future: on one hand, due to the ubiquity of computer technologies, more and more massive data sets are available in electronic form; on the other hand, an increasing fraction of the users accessing these data are non-specialists.

Information visualization has precisely the key role of translating abstract information into geometric realities, allowing the user to easily understand, browse, and manipulate them in a natural and intuitive way, and its task becomes particularly critical as the amount of the data to be visualized increases.

So strategic is considered the research on information visualization to receive attention in the Implementation Plan proposed in the President’s FY 2000 Budget National Science and Technology Council [101], stressing the relationship with large data sets by stating explicitly that “the goal of visualization research is to significantly improve our ability to see, understand, and manipulate huge quantities of data”.

Also, the research on how visualization techniques can be applied when large data set are concerned answers a recent call of the computational geometry community, which has expressed an interest in strengthening the original link with applications by modeling the complexity and imperfections of the physical world and by coping with the limitations of realistic computing devices [128].

A limited list of the applications that could take advantage of the research on visualization of large data sets includes the following:

- The representation of computer networks for diagnostic, design, and didactic purposes. Computer networks, in the local as well as in the wide area perspective, defy automatic representation techniques, featuring recurrent topological schemes, and a natural clustering counterposed to global interconnection.
• Support tools for Web browsing and representation of Web sites. The Web can be viewed as an ever-changing interconnection graph whose maintenance, as well as exploration, requires large amounts of data to be considered.

• Data Structures and function calls representation. Large software engineering projects imply a large number of designers sharing a common framework. The underlying data structures are largely common to several software modules and independently accessed and updated. A graphic representation of the data and of the relations between the function accessing them may provide a better understanding of their inner structure and of the dynamics involved in the running process, both in the synthesis and in the debugging phase of the project.

• The representation of chains of biological processes or molecular transformation. The Genome project is only an example of how the biological research area is producing large amounts of data, that need to be explored, analyzed, classified and compared. Such tasks can benefit from specific automatic visualization tools, provided that huge amounts of data could be managed.

• Representation of industrial plants schemes. Industrial plants are currently represented by handmade diagrams that could be modeled as strongly constrained block representations, in which each block corresponds to a device or component of the industrial plant, while links correspond to control signals or actual movements of materials between the plant facilities. The capability of producing constrained drawings of large amount of data would allow us to automatically produce such diagrams.

• Representation of integrated circuits for VLSI design. Different phases of the design of integrated circuits may take advantage of human interaction if a proper representation could be available. Candidate phases are the logical debugging phase, in which a visual inspection could be helpful, or the partitioning phase, in which an exact algorithm can not be applied due to the NP-hardness of the underlying problem.

The information to be represented can often be modeled as a graph. If this is not the case, sometimes simple transformations can be applied in order to obtain graphs as simplified models of structurally more complex information. In Chapter 14, for example, following a usual technique, we apply a clique transformation in order to obtain a weighted graph starting from a network representing a huge circuit.
Several are the reasons why graphs are so commonly used: first they are expressive enough to represent a wide range of structured information, secondly graphs have so a natural and intuitive visual representation to be generally identified with it, and finally the user perception of such pictorial representation is highly effective, since evidence indicates that the relationship between a pair of objects is better perceived when it is represented through adjacency, rather than through size, proximity, color or shape [134].

The graph drawing research field offers several general solutions to different visualization problems. The scalability of such solutions, though, is generally not guaranteed, and the visual and computational effects of huge quantities of data is seldom considered. This work explores the impact of large data sets on the graph drawing problem, registering the efficiency limits of current widely adopted drawing approaches and the potentials of alternative ones.
Chapter 2

Graph Drawing Background

In this chapter we recall well-known concepts and notations about graphs and their drawings that are needed in the rest of this thesis.

2.1 Graphs

The purpose of the graph drawing research field is to investigate automatic methods for the visual representation of graphs.

An undirected graph (or graph) is a pair $G = (V, E)$, where $V$ is a finite and non-empty set of elements called vertices (or nodes) of $G$, and $E$ is a finite (possibly empty) set of elements called edges (or arcs) of $G$. An edge $e \in E$ is an unordered pair $\{u, v\}$ of vertices of $G$; vertices $u$ and $v$ are called end-vertices of $e$, or simpler vertices of $e$. We say that $e$ connects $u$ to $v$. Vertices $u$ and $v$ are said to be adjacent, while $e$ and any of its end-vertices are incident. Two edges are adjacent if they have a common vertex. The number of edges that are incident on a vertex $v$ is called the degree of $v$ and denoted as $\text{deg}(v)$.

An edge of a graph $G$ is a self-loop if its end-vertices coincide. Also, $G$ contains multiple edges if it has two or more edges with the same end-vertices. A graph $G$ is simple if it has neither self-loops nor multiple edges.

A path $p$ between two vertices $v_0$ and $v_k$, $k \geq 1$, of a graph $G$ is an alternating sequence $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$ of vertices and edges of $G$, where $e_i = (v_{i-1}, v_i), (i = 1, \ldots, k)$. Vertices $v_0$ and $v_k$ are the end-vertices of $p$. We often write $p = (v_0, v_1, \ldots, v_k)$ to denote path $p$. If all vertices of $p$ are distinct, $p$ is said to be a simple path. The length of a path is the number of its edges. If the end-vertices of $p$ coincide (i.e. $v_0 = v_k$) the path is a cycle. A simple cycle is a cycle such that only its end-vertices coincide, while all its other vertices are distinct. A graph is said to be acyclic if it does not contain any cycle. A graph that is not acyclic is also called cyclic.

A subgraph $G'$ of a graph $G = (V, E)$ is a graph $G' = (V', E')$, such that
$V' \subseteq V$ and $E' \subseteq E$. If $V'$ is a subset of $V$, the subgraph of $G$ induced by $V'$ is the graph $G' = (V', E')$, where $E' \subseteq E$ is the set of all edges of $G$ connecting any two vertices of $G$ that are in $V'$ (see Figure 2.1). Note that the subgraph induced by a subset of vertices is uniquely determined.

![Image](image-url)

**Figure 2.1:** (a) A graph $G$ with 7 vertices. (b) The subgraph of $G$ induced by vertices $v_1, v_2, v_3, v_6$ and $v_7$.

A *subdivision* of a graph $G = (V, E)$ is a graph $G'$ obtained from $G$ by replacing a subset $E'$ of edges of $G$ with paths, where two paths may share only end-vertices. Namely, if $e = \{u, v\} \in E'$, it is replaced with a path $p = \{u = v_0, v_1, \ldots, v_k = v\}$, $k > 1$, where the vertices $v_1, \ldots, v_{k-1}$ do not belong to $V$. Subdivision $G'$ is said to be *homeomorphic* to graph $G$. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a one-to-one correspondence between sets $V_1$ and $V_2$ and a one-to-one correspondence between sets $E_1$ and $E_2$, such that vertex $v_1 \in V_1$ is incident to edge $e_1 \in E_1$ if and only if the corresponding vertex $v_2 \in V_2$ and edge $e_2 \in E_2$ are incident. In practice, two graphs that are isomorphic can be considered as the same object.

A graph is *complete* if it has an edge connecting every pair of vertices. A complete graph with $n$ vertices is usually denoted as $K_n$. A graph $G$ is *bipartite* if the set of its vertices can be partitioned into two sets $V_1$ and $V_2$ such that every edge of $G$ connects a vertex in $V_1$ to a vertex in $V_2$. A graph $G$ is complete bipartite if it is bipartite and every vertex in $V_1$ is adjacent to all vertices in $V_2$. A complete bipartite graph such that $|V_1| = n_1$ and $|V_2| = n_2$ is usually denoted as $K_{n_1,n_2}$.

A *directed graph* $G$ is defined similarly to a graph, except that every edge $e$ of $G$ is now an *ordered* pair $(u, v)$ of vertices of $G$; vertices $u$ and $v$ are still called *end-vertices* of $e$, or simpler *vertices* of $e$. We also say that $e = (u, v)$ leaves vertex $u$ and enters vertex $v$. An edge that leaves (enters) a vertex $v$ is an *outgoing* (*incoming*) edge of $v$, and it is *directed* from $v$ to $u$. 
2.1. **Graphs**

If $G$ is a directed graph, ignoring for every edge of $G$ the order of its end-vertices (i.e. its direction), we obtain an undirected graph that is called **underlying graph** of $G$. A vertex $v$ and an edge $e$ of $G$ are **incident** if they are incident in the underlying graph of $G$. Two vertices or two edges of $G$ are **adjacent** if they are adjacent in the underlying graph of $G$.

A **directed path** $p$ from a vertex $v_0$ to a vertex $v_k$, $k \geq 1$, of a digraph $G$ is an alternating sequence $v_0, e_1, v_1, e_1, \ldots, e_k, v_k$ of vertices and edges of $G$, where $e_i = (v_{i-1}, v_i), (i = 1, \ldots, k)$. Path $p$ is a directed cycle if $v_0 = v_k$. Directed paths and cycles of a digraph $G$ are **simple** if they are simple in the underlying graph of $G$. A digraph $G$ is **acyclic** if it does not contain any directed cycle.

A vertex $v$ of a digraph $G$ is a **source** (sink) if it has no incoming (outgoing) edges. An acyclic digraph with exactly one source $s$ and one sink $t$ is called an **st-digraph**.

The definitions that follow are given for graphs. In the case of a digraph they are referred to the underlying graph.

A **connected component** $G'$ of a graph $G$ is a maximal subgraph of $G$ such that for every pair $\{u, v\}$ of vertices of $G'$ there is a path between $u$ and $v$ in $G'$. A graph that has exactly one connected component is said to be connected (or 1-connected). A connected subgraph $G'(V', E')$ of $G(V, E)$ such that $V' = V$ is a **spanning subgraph** of $G$. A separating $k$-set, $k \geq 1$, of a graph $G$ is a set of $k$ vertices whose removal increases the number of connected components of $G$. Separating 1-sets and 2-sets are called **cutvertices** and **separation pairs**, respectively. A connected graph is **biconnected** (or 2-connected) if it has no cutvertices. A biconnected graph is **triconnected** (or 3-connected) if it has no separation pairs. More in general, a graph is $k$-connected, $k \geq 2$, if it has no separating $(k-1)$-sets. Obviously, if a graph is $k$-connected, it is also $(k-1)$-connected. Figure 2.2 examples of connected, biconnected, and triconnected graphs are given.

In the rest of this work we simply use the term “graph” to mean “connected graph”.

A graph that is connected and acyclic is called **tree**. A collection of trees (i.e. an acyclic graph) is a **forest**. A **rooted tree** is a tree $T$ in which a vertex is chosen as root. Such a choice immediately induces a hierarchy on the set of all vertices of $T$. Namely, with each vertex of $T$ we can associate an integer number called **depth** (or level) of the vertex. The depth of the root is 0. For any vertex $\mu$, distinct from the root, the depth of $\mu$ is the length of the unique path between the root and $\mu$ in $T$; every vertex $\beta \neq \mu$ on this path is an ancestor of $\mu$, and $\mu$ is a descendant of $\beta$. Also, if $(\beta, \mu)$ is an edge of the path between the root and vertex $\mu$, $\beta$ is the parent of $\mu$, and $\mu$ is a child of $\beta$. A vertex without children is a leaf. The depth of $T$ is the maximum depth of a leaf of $T$.

Let $T$ be a rooted tree. If an ordering of the children of every vertex of $T$
is given, we say that $T$ is an ordered rooted tree.

Let $G$ be a connected graph. A biconnected component (or block) of $G$ is a maximal biconnected subgraph of $G$. Observe that any cut vertex of $G$ belongs to at least two distinct blocks. It is possible to construct a tree $T$, called block cutvertex tree (and often denoted as BC-tree) that describes the set of all blocks of $G$ and the relationships among them. This tree, introduced by F. Harary [63], can be computed in linear time. It is defined as follows.

- With each block of $G$ there is an associated $B$-node of $T$.
- With each cut vertex of $G$ there is an associated $C$-node of $T$.
- There is an edge in $T$ connecting a $B$-node $\beta$ to a $C$-node $\chi$, if and only if the cut vertex associated with $\chi$ belongs to the block associated with $\beta$.

Observe that in a BC-tree there are neither edges connecting two $B$-nodes nor edges connecting two $C$-nodes. In Figure 2.3 it is shown a graph and the corresponding BC-tree.

### 2.2 Drawings of Graphs

The definitions of this section are given for graphs. In the case of a digraph they are referred to the underlying graph.

Let $G$ be a graph. A two-dimensional drawing of $G$ maps each vertex of $G$ into a point of the plane, and each edge $\{u, v\}$ of $G$ into a simple Jordan
curve between the two points corresponding to $u$ and $v$, respectively. A three-dimensional drawing of $G$ maps each vertex of $G$ into a point of the space, and each edge $\{u, v\}$ of $G$ into a simple curve between the two points corresponding to $u$ and $v$, respectively. We simply refer to a drawing of a graph $G$ anytime the two-dimensionality or three-dimensionality of it is apparent from the context.

The theory of planarity investigates the properties of graphs and of their two-dimensional drawings. A two-dimensional drawing is planar if no two distinct edges intersect. A graph is planar if it admits a planar drawing. Planar graphs play a crucial role in the graph drawing field and in the whole of graph theory [30, 102].

A planar drawing $\Gamma$ subdivides the plane into topologically connected regions called faces. Exactly one of these faces is an unbounded region, and it is called external face. All the other faces are said to be internal. The degree of a face $f$, denoted as $\text{deg}(f)$, is the number of edges encountered while walking on the border of $f$ clockwise. In particular, if the graph is biconnected $\text{deg}(f)$ always coincides with the number of edges that belong to the border of $f$. In the rest of this work, in order to simplify the notation, we often speak of edges of a face $f$ to mean the edges that belong to the border of $f$. In this way, we can also consider a face as a cycle, and describe it as a circular sequence of vertices and edges.

There is a simple necessary condition for connected planar graphs, known as the Euler’s formula.

**Theorem 1 (Euler 1750)** Let $G = (V, E)$ be a planar graph and let $F$ be
any set of faces of a planar drawing of $G$. Then:

$$|E| = |V| + |F| + 2.$$ 

Euler’s formula allows us to assert that the number of faces of any planar drawing of a planar graph $G$ does not depend on the choice of the drawing itself.

An result due to Kuratowski provides a characterization of the set of planar graphs.

**Theorem 2 (Kuratowski 1930)** A graph is planar if and only if it does not contain any subdivision of $K_5$ or $K_{3,3}$.

It is easy to prove that a graph is planar if and only if all its connected components are planar; further a connected graph is planar if and only if all its blocks are planar.

We now observe that a planar drawing $\Gamma$ of a planar graph $G$ induces a circular clockwise ordering of the edges incident on each vertex. Two planar drawings $\Gamma_1$ and $\Gamma_2$ of $G$ are equivalent if they induces the same circular clockwise ordering of edges around vertices, and if they have the same external face. Such binary relationship is clearly an equivalence relationship. We call planar embedding of $G$ an equivalence class of planar drawings of $G$. A planar graph $G$ with an associated embedding $\phi$ is an embedded planar graph, and it is often denoted as $G_\phi$. Note that all the drawings in the embedding $\phi$ have the same set of faces and the same external face. For this reason, we can speak without ambiguity of faces of $G_\phi$. To specify that a drawing $\Gamma$ belongs to an embedding $\phi$ of $G$, we can say that $\Gamma$ preserves $\phi$, or that $\Gamma$ is a drawing of $G_\phi$.

In order to simplify the terminology, when working with planar graphs, we often use the term embedding instead of planar embedding.

Many algorithms in graph drawing are specific for planar graphs. Testing if a graph is planar can be done in linear-time [69, 12, 28, 86]. Planarity testing algorithms can be also modified to determine a planar embedding if the graph is found to be planar [20, 92]. However, all current linear time algorithms for planarity testing are based on sophisticated techniques or data-structures that are difficult to realize. A description of the Hopcroft-Tarjan planarity testing algorithm [69], that studies important implementation details, is given in [92]. Also, an efficient on-line planarity testing algorithm is described in [34].

When a graph is not planar, a pre-processing step can be performed in which the graph is made planar by adding a suitable set of dummy vertices to replace crossings. Such a step is usually called planarization. Reducing the number of dummy vertices (i.e. of crossings) added by a planarization
2.3. ORTHOGONAL DRAWING CONVENTION

algorithm is an important target for obtaining more readable drawings. For a survey on planarization heuristic algorithms see [30]. Finding the minimum number of crossings and finding a maximum planar subgraph are both NP-hard problems. Combinatorial optimization techniques for the maximum planar subgraph problem have been investigated in [73].

A drawing convention is a basic rule that the drawing must satisfy in order to be admissible. A drawing convention of a real-life application can be very complex and can involve many details of the drawing. In the following we will examine two widely used drawing conventions: the orthogonal drawing convention and the straight-line convention.

2.3 Orthogonal Drawing Convention

The orthogonal drawing convention is recognized to be suitable for several types of diagrams, including data flow diagrams, entity-relationship diagrams, state-transition charts, circuit schematics, and many others. Such diagrams are extensively used in real-life applications spanning from software engineering, to databases, real-time systems, and VLSI.

A drawing of a graph $G$ such that all the edges are mapped to polygonal chains of segments parallel to the axes is an orthogonal drawing. A turn on a polygonal chain of an orthogonal drawing is called bend. An orthogonal grid drawing of $G$ is an orthogonal drawing such that vertices and bends have integer coordinates.

![Figure 2.4: (a) A planar orthogonal drawing with 6 bends. (b) A planar orthogonal grid drawing with 6 bends. (c) A bend-optimal orthogonal drawing in the same class as the drawings in (a) and (b). The two orthogonal drawings in (a) and (b) are shape equivalent.](image)

A $k$-planar graph is a planar graph such that every vertex has degree at most $k$. The following theorem gives a characterization of the existence of two-dimensional orthogonal drawings.
Theorem 3 [127] A planar graph has a two-dimensional orthogonal drawing if and only if it is 4-planar.

The corresponding theorem for three-dimensional orthogonal drawings is simpler, since planarity is not needed:

Theorem 4 A graph has a three-dimensional orthogonal drawing if and only if every vertex has degree at most 6.

Let $\Gamma$ and $\Gamma'$ be two two-dimensional orthogonal drawings of a planar graph $G$, preserving the same embedding. We say that $\Gamma$ and $\Gamma'$ are shape equivalent if:

- For each vertex $v$ of $G$, consecutive edges incident on $v$ form the same angle in the two drawings, and
- For each edge $\{u,v\}$ of $G$ we have the same (possibly empty) sequence of left and right turns in the two drawings when walking from $u$ to $v$.

The two orthogonal drawings in Figure 2.4 (a) and Figure 2.4 (b) are shape equivalent.

An orthogonal representation $H$ of a planar graph $G$ is a class of shape equivalent orthogonal drawings of $G$. We say that $H$ preserves an embedding $\phi$ of $G$ when the drawings in $H$ preserve $\phi$. In this case we say that $H$ is an orthogonal representation of $G_\phi$. By the definition, all orthogonal drawings in $H$ have the same number of bends.

Roughly speaking, an orthogonal representation defines a class of orthogonal drawings that may differ only for the length of the segments of the edges. Any orthogonal representation $H$ can be completely described by specifying the values of the angles around every vertex and the ordered sequence of left and right turns on each edge. The following result gives a characterization of planar orthogonal representations.

Theorem 5 [127] Let $G_\phi$ be an embedded planar graph. $H$ is an orthogonal representation of $G_\phi$ if and only if the following properties hold:

- For every vertex $v$ of $G_\phi$, the sum of the angles around $v$ in $H$ is equal to 360 degree.

- Let $f$ be a face of $G_\phi$. Walk clockwise on the border of $f$ in $H$ and consider the sum $S_r$ of right turns and the sum $S_l$ of left turns, where a right turn can either a right bend or an angle of 90 degree inside $f$, and a left turn can be either a left bend or an angle of 270 degree inside $f$; an angle of 360 degree inside $f$ is considered as 2 left turns. Then

$$S_r - S_l = 4.$$
2.4 STRAIGHT-LINE DRAWING CONVENTION

Given an orthogonal representation \( H \) of a planar graph \( G \), a planar orthogonal drawing \( \Gamma \) of \( H \) can be computed in linear time \([127, 30]\) with a compaction algorithm. This algorithm assigns coordinates to the vertices and bends of the representation trying to reduce “as much as possible” either the area or the total edge length of the final drawing. Unfortunately, both this two problems are shown to be NP-hard \([112]\).

We say that an orthogonal representation is optimal when it has the minimum number of bends. Tamassia showed that an orthogonal representation with the minimum number of bends of an embedded planar graph can be computed in polynomial time \([127]\), by a very elegant reduction to a flow problem. Figure 2.4 (c) shows an optimal orthogonal representation in the same class as the drawings in Figure 2.4 (a) and Figure 2.4 (b). The problem of computing an optimal orthogonal representation of a planar graph, that does not necessarily preserve a given embedding, has been proved to be NP-hard \([61]\).

The length (height, depth, respectively) of an orthogonal grid drawing \( \Gamma \) is the maximum difference between the \( x \) (\( y \), \( z \), respectively) coordinates of its vertices.

A \((l, h)\)-compactable orthogonal representation is an orthogonal representation such that \( l \) is the minimum length of all its orthogonal grid drawings, and \( h \) is the minimum height between all orthogonal grid drawings with length \( l \).

The area of a two-dimensional orthogonal grid drawing is the product of its length and height. The volume of a three-dimensional orthogonal grid drawing is the product of its length, height, and depth. The total edge length of an orthogonal grid drawing is the sum of the lengths of its edges. The maximum edge length of an orthogonal grid drawing \( \Gamma \) is the maximum value of all its edge lengths.

2.4 Straight-Line Drawing Convention

A straight-line drawing is such that all the edges are drawn as straight line segments.

Every planar graph admits a two-dimensional straight-line grid drawing, as independently established by Wagner, Fary and Stain ([133], [49], and [126], respectively). Furthermore, every plane graph of \( n \) vertices has a two-dimensional straight-line grid drawing that fits into a rectangle of area \( O(n \times n) \), and this bound is asymptotically tight ([27] and [123]).

For three-dimensional straight-line grid drawings a complexity of \( O(n \times n \times n) \) of volume occupation was proven in [23].
Chapter 3

Efficiency Problems

Several problems hinder the use of standard graph drawing techniques when large data sets are involved. These problems can be classified in two categories: efficiency problems and effectiveness problems. This chapter is devoted to the first category, while the latter category is considered in the next chapter.

Computational efficiency is an important parameter of any graph drawing algorithm ([30]) and it is a critical measure to take into account if large graph have to be represented. In fact, when large amounts of data are involved, some phases of the graph drawing process may become computationally too expensive to be performed. Although asymptotic complexity cannot be the definitive measure of the practical usability of an algorithm, assessing the intrinsic complexity of a drawing method is the first step for the comprehension of its scalability.

In the following we will discuss the efficiency of the most common drawing methods to produce two-dimensional drawings in the orthogonal and in the straight-line conventions.

3.1 The Topology-Shape-Metric Approach

A well known and widely used approach to produce two-dimensional orthogonal drawings is the topology-shape-metric approach (see, for example, [7, 127, 54, 9, 80, 30]), in which the graph drawing process is organized in three algorithmic steps (see Fig. 3.1):

Planarization In this step an embedding for \( G \) is computed. If \( G \) is not planar, a set of dummy vertices is added to replace crossings in order to compute an embedding \( \phi \) for \( G \). The aim of this step is to obtain a planar embedding by introducing a small number of dummy vertices.
Figure 3.1: the three steps of the topology-shape-metric approach: (a) abstract description of a graph; (b) planar embedding produced by the planarization step; (c) orthogonal representation produced by the orthogonalization step; and (d) final drawing produced by the compaction step. The dummy vertex (black) is introduced by the planarization step and removed at the end of the compaction step.

**Orthogonalization** During this step, an orthogonal representation $H$ of $G_{\phi}$ is computed within the embedding $\phi$ aiming at obtaining an orthogonal representation with a small number of bends.

**Compaction** In this step a final geometry for $H$ is determined. Namely, coordinates are assigned to vertices and bends of $H$. The purpose of this step is the reduction of the size (area, total edge length, maximum edge length) of the final drawing.

The Topology-Shape-Metrics approach allows us to deal with topology, shape, and geometry of the drawing separately, so simplifying the whole drawing process. On the other hand, decisions taken in early steps cannot be changed, thus overall optimization is not achieved in general. For instance, introducing dummy vertices forces crossings to appear on specific edge pairs, thus the total number of bends of the final drawing may be not optimal.

The distinct phases of the Topology-Shape-Metrics approach have been extensively studied in literature. If $G$ is planar an embedding $\phi$ of $G$ is determined in linear time, by applying a well-known planarity testing algorithm [69, 20]. If $G$ is not planar, the minimum number of dummy vertices introduced may be $\Omega(n^4)$. However, in practice this number is usually much smaller. Minimizing the number of crossings is in general NP-hard. See [30] for a survey on planarization techniques.

A popular algorithm for constructing an orthogonal representation of an embedded graph with vertices having at most four incident edges was presented by Tamassia [127]. Such algorithm computes an orthogonal representation that has the minimum number of bends within the given embedding.
3.2. Force Directed Approaches

Force directed approaches are nondeterministic layout techniques using a physical analogy to obtain straight-line drawings of undirected graphs. They originate from the first spring embedder algorithm introduced by Eades in [38]. Edges are modeled as springs opposing any stretching and compression with a force proportional to their elastic constant. Furthermore, vertices repel each other as if electrically charged, with a force quadratically increasing as their distance decreases. A variety of numerical techniques can be used to find an equilibrium configuration. A simple algorithm works as follows. Vertices are initially placed at random locations. At each iteration, the force on each vertex is computed, and each vertex is moved in the direction of the force by a small amount proportional to its magnitude.

The output of the algorithm is highly unpredictable, as the elastic constant, the electric constant, and the natural length of the springs needs to be finely tuned for the specific problem or graph, since the initial configuration is usually a random one, and since vertex overlapping is generally handled with random perturbations of the system. Nevertheless, force directed approaches are very popular for several reasons:

- The physical analogy makes them easy to understand and relatively simple to code.
- No hypothesis is made on the input graph. Planarity, for example, is not requested. Connectivity properties of the graph are not concerned
by the drawing process.

- The drawing approach is highly tailorable: not only the electrical and elastic constants can be finely tuned, but the behavior of attractive and repulsive forces themselves can be changed. For example, logarithmic springs can be used instead of Hooke’s law springs [38].

- Constraints, as position constraints or fixed-subgraph constraints can be naturally be handled by the model by adding additional forces to account for particular requirements of the applications.

- The results can be good.

Unfortunately, force-directed methods are notorious for using considerable computational resources. Each iteration involves a visit of all pairs of vertices in the graph and the quality of the layout depends on the number of full iterations. Several attempts have been made to improve their efficiency. These include:

- In [57] the use of force functions that are more amenable to efficient algorithms for finding local minimal is proposed. Brandenburg, Himsolt, and Rohrer, in their extensive empirical analysis of force directed and randomized graph drawing methods [14], point out that this algorithm is fast on small graphs (less than about 60 vertices).

- The use of some randomization in the style of simulated annealing.

- The use of sophisticated methods for numerical analysis to solve equations that arise from the various models. For example, the recognition and the special treatment of stiff equations.

- The adoption of preprocessing algorithm to place the vertices in a sensible starting position that is presumably near the equilibrium one.

Although the above strategies may be helpful in some cases, the computational expense of a force directed method is still an unsolved problem for very large graphs. Even one of the best variants [56] is still estimated to work with a complexity of $O(n^3)$, where $n$ is the number of vertices of the graph ([67]). A drastic solution to cut down the complexity of the algorithm is that of eliminating from the model repulsive forces between not adjacent vertices, but in this way vertices that are topologically far one from the other in the graph tend to overlap, and the drawing wraps on itself.
Chapter 4

Effectiveness Problems

Large graphs imply a large number of objects to be shown to the user. The ability of the visualization process to convey information to the user is affected in two different ways:

- Growing the size of the graph, the graphical resolution limits of the visualization devices jeopardize the discernibility of vertices and edges. The number of pixels of the screen of the device offers an upper bound on the number of objects that can be visualized.

- The user’s comprehension of the semantic roles and mutual relationships of the objects presumably decrease as their number increases.

The second phenomenon is much more difficult to measure. In fact, although it is well known that comprehension and detailed analysis of data in graph structures is easier when the size of the displayed graph is small, very few findings in cognitive science have practical applications at this time and very few usability studies have been done [67]. It follows that, even if usability may become an issue before the problem of graphic discernibility arises, only the latter seems susceptible to be measured, while an objective measure of information overload is difficult to conceive.

On the other hand, studies about how readability is influenced by graphic properties of the drawing (the so called aesthetics) exist ([6, 117, 115, 116]), and one could reasonably expect that an aesthetic particularly critical for the comprehension of the drawing of a small graph is even more critical when the size of the graph is increased.

In this respect, the main result emerging from the experiment seems to be that the minimization of the number of intersections is by far the most important aesthetic for the human comprehension of the drawing. Unfortunately enough, the problem of minimizing the crossing number is NP-complete both
for general graphs [60], and for particular cases as layered drawings [47] or upward rectilinear drawings [61].

Thus, since pursuing the minimization of the number of intersections is unfeasible (only the recognition of planar graphs can be reasonably advocated) and since the experiments till now performed usually concern graphs of extremely contained size (the number of vertices of the test graphs is seldom greater than 20), very few are the findings of the research about graph drawings aesthetics that can be applied to large graphs.

4.1 Drawing Conventions and Effectiveness Considerations

Different graph drawing conventions show different properties with respect to graphic resolution problems. In the orthogonal grid drawing convention, for example, the distance between unrelated objects (that is non adjacent vertices, non adjacent edges, and non incident vertices and edges) is bounded by the grid distance. The convention rules guarantee that, if we consider a partial view showing an area of a fixed size of the drawing, unrelated objects will stay distinguishable even if the size of the input graph (and of the whole drawing) increases. Alternatively, imagining the area of the drawing to be constrained by the screen size, a rough measure of how much unrelated objects tend to collapse when the number of the vertices of the graph increases can be easily found. Namely, since the area occupation of a two-dimensional orthogonal grid drawing of a planar and of a non planar simple graph, is $\Theta(n) \times \Theta(n)$ [130, 11], growing the graph to be represented, the grid distance has to be linearly reduced in order for the drawing to fit into the screen.

In contrast, consider the straight-line drawing convention. In a two-dimensional straight-line grid drawing with side length $f(n)$, the distance between a vertex and a non incident edge may be as short as $\frac{1}{f(n)}$ (see Figure 4.1), while in a three-dimensional straight-line grid drawing of size $f(n)$, the same distance may be $\frac{1}{f(n)^2}$ (see Figure 4.2).

It follows that adopting a two-dimensional (three-dimensional) straight-line convention, the risk of overlaps in a partial view of the drawing grows linearly (quadratically) with the number of vertices of the graph, since both for two-dimensional and three-dimensional straight-line grid drawings $f(n)$ is $O(n)$. Considering a picture of the whole drawing fitting into the available screen, the smallest distance between two unrelated objects diminishes quadratically (cubically).

An immediate consequence of the remarks above is that vertices of an orthogonal grid drawing may be assigned an area different from zero without generating intersections with non-incident edges, while the size of the vertices
4.1. DRAWING CONVENTIONS AND EFFECTIVENESS CONSIDERATIONS

Figure 4.1: In a two-dimensional straight-line grid drawing of size $f(n)$, the distance between an edge and a vertex may be as short as $\frac{1}{f(n)}$.

of a two-dimensional straight-line grid drawing should be inversely proportionate to the number of the vertices of the graph to guarantee the absence of intersections with non-incident edges, even if the drawing method per se guarantees no intersections.

Thus, orthogonal and straight-line grid drawings show very different behaviors with respect to the problem of large graph representations: orthogonal grid drawings tend to be less comprehensible in a global view of the graph, due to the introduction of bends, but, on the other hand, allow a clearer focused analysis of the drawing, since the convention rules guarantee a fixed distance between unrelated objects. Conversely, straight-line drawings may be clearer when the whole graph is represented, since edges have no bends, but tend to be less comprehensible for a focused analysis, since edges tend to overlap with vertices, even when vertices are constrained to be on the grid.
Figure 4.2: In a three-dimensional straight-line grid drawing of size $f(n)$, the distance between two edges may be as short as $\frac{1}{f(n)}$. 
Part II

Methodologies and State of the Art
Chapter 5

General Principles

Two main principles may be adopted to approach the efficiency and effectiveness problems described in Chapters 3 and 4. The hiding principle consists of simplifying the information to be represented, while the stressing principle, consists of emphasizing the objects and their relationships to help the user understanding the information they represent. Both the principles may be used by a single complex strategy (Chapter 7), finding an equilibrium point between different tradeoffs (Chapter 6).

5.1 Hiding Principle

The hiding principle consists of concealing the parts of the graph upon which the attention of the user should not focus. In fact, both from a cognitive perspective, and from a mere device-resolution one, it does not even make sense to present the user with a very large amount of data.

The concept of hiding is more general than the concept of filtering [104] since it does not necessarily imply the complete removal of the objects to hide, and includes every attempt to de-emphasize some object by partially covering or occluding it, giving it some transparency, shrinking it, etc.

The hiding operation may happen at two different levels:

- it may be performed on the input graph (semantic hiding), or
- it may be performed on the drawing itself (geometric hiding).

Although both semantic and geometric hiding are capable of increasing the effectiveness of the representation, since they both reduce the amount of information to be conveyed to the user, only semantic hiding has beneficial effects on the efficiency of the drawing method, by cutting down the size of the graph to be displayed before the drawing process is started, while geometric hiding needs to operate on a drawing of the whole graph.
Also, the visualization process may be affected by semantic hiding in a second way: by producing a graph that is not only reduced in size with respect of the original one, but that has acquired some additional graph theoretic properties which significantly ease the layout process (for example, it may produce a graph that is connected and acyclic).

Further, the hiding principle necessarily implies the interaction with the user, who is usually allowed to operate on the interface and to ask for the visualization of the parts of the drawing that are hidden in the current view. Thus, navigation and interaction facilities are essential in conjunction with hiding techniques.

For its beneficial effects on efficiency, semantic hiding should be preferred to geometric hiding. The main advantage of the latter, though, is that it is intuitively handled by the user in an interactive setting, a characteristic that is not guaranteed by semantic hiding, and that needs to be explicitly pursued in that case. The user, in fact, accepts that objects that are geometrically “far” from the focused part of the drawing are concealed in the current view, and expects them to maintain their geometric positions when a shift of the focus introduces some new vertices in the drawing and hides some others. Interacting with a static geometric world is a comforting feeling that helps the user managing the interaction primitives and is straightforward to realize when the hiding is a geometrical one, that is, when an underlying static geometric description of the whole graph actually exists.

In the case of semantic hiding a coherent geometric description of the whole graph is not necessarily available at any moment, and only the subgraph that has to be shown is drawn by the application. The user mental map has to be preserved in some way when the focus is shifted.

5.2 Stressing Principle

The stressing principle consists of emphasizing the objects presented to the user and their relationships with the purpose of helping the user perceiving the information they represent. Applying the stressing principle never reduces the information to be shown. On the contrary, the information is often increased, as graphic features may be added to the drawing, and in any case its representation is suitably modified in order to make it more expressive. Different kinds of stressing can be recognized depending on the operations that are performed and on the object that are concerned:

- in additive stressing some graphic features are added as decorations to the objects. This enriching operation may regard the color, shape, or size of the object, or the presence of labels attached to the edges or vertices of the graph. This corresponds to adding information to the diagram,
5.2. STRESSING PRINCIPLE

Figure 5.1: Examples of additive stressing: using colors (b), shapes (c), sizes (d), and labels (e).

rather than reducing it. Vertices with the same role, for example, may have similar graphic features in order to help the user perceive their common nature (see Fig. 5.1).

- In **positional stressing** some objects or relations between them are emphasized by their relative positions. Vertices may be constrained to be at the center of the drawing, or at the top of it, or on the external border of the drawing. A set of vertices may share the same internal face, or be placed into the same area. Edges incident to the same vertex may have a prescribed and meaningful circular order around it. Intersections may be precluded for some edge, and some particular path (for example a notable shortest path) may be constrained to be drawn on a straight line (see Fig. 5.2).

- In **global stressing** a global property of the drawing is pursued so to increase the comprehension of the whole drawing. Several are the properties capable of increasing the readability of a drawing, and they correspond to the aesthetic criteria mentioned in Chapter 4. A limited list includes the number of crossing, the number of bends, the orthogonality of the edges, the angular resolution, etc. A typical property capable of
Figure 5.2: Examples of positional stressing; a vertex constrained to be at the center (a) or at the top (b) of the drawing, vertices constrained to share the external face (c), vertices constrained to be near one to the other (d), intersections precluded for some edges (e), and a significant path drawn on the same straight line (f).

Increasing the readability of the drawing is the minimum area (or volume) of the drawing itself, which may allow the whole diagram to be encompassed by a single view of the user (the problem of reducing the area of an orthogonal drawing is discussed in Chapter 9). Other measures that may be reduced are the maximum edge length, or the average edge length (Chapter 10).
Chapter 6

Design Tradeoffs

This chapter contains a list of directions that can be explored in order to cope with large data sets. Each of them represents a design opportunity, and corresponds to sacrificing some desirable characteristic of the drawing or drawing method in order to increase the scalability of the solution.

6.1 The Relationship Between Efficiency and Effectiveness

In Chapters 3 and 4 we discussed how large data sets impact both the efficiency and the effectiveness of the representation. The relationship between the two, though, seems to be more subtle and complex than that, since improving the effectiveness sometimes implies considerable computational costs.

Thus, increasing the efficiency of a drawing algorithm may allow us to invest some computational time in pursuing effectiveness, and, conversely, finding an extremely effective representation may induce the designer to start from a lesser quality drawing, eliminating an expensive algorithmic step.

6.2 General and Domain Specific Solutions

A common choice in the effort to achieve greater scalability is relaxing the constraint for total generality to devise solutions for domain-specific problems [100]. Two are the ways in which this relaxation may take place:

- the class of graphs susceptible to be handled by the drawing method may be narrowed by assuming some particular property to be satisfied, as, for example, the property of being acyclic or quasi-hierarchical;

- it may be assumed that some additional information is available to the
drawing method, such as a partition on the vertices of the graph, or a significant weight for the edges.

6.3 Static Versus Interactive Scenarios

For interactive scenario we mean one in which the user is asked to cooperate in the displaying process, by selecting the objects that need to be shown or hidden, choosing viewpoints and zooming factors, or simply affecting the display process by setting some visualization parameter. Static scenarios, in contrast, are those that request from the user only to specify an input and collect the output of the drawing algorithm.

In computer science the word “interactive” has a special fascination, and, definitely, a positive flavor. Interactivity is generally seen as a great opportunity. Thus, it is unusual to talk about the disadvantages of interaction. As a matter of fact, though, static scenarios are easier to realize and easier to understand. The user barely needs some training to use a static visualization tool. Furthermore, the drawings produced have a wider scope of application. They may be, for example, printed on paper.

Interactive scenarios constrain the solutions to the computer-based visualization domain, narrowing their field of application. Furthermore, they need somehow more learned users, trained to interact with the machine, and demand, usually, the protracted attention of such sophisticated users.

Undoubtedly, interaction has beneficial effects on the human’s understanding of the information visualized. Interacting with the objects, for example, helps memorizing them. Thus, interaction is welcome when it is not indispensable and the solution maintains both the advantages of interactive and static scenarios. But when large data sets are involved, interaction tends to become indispensable and the advantages of static scenarios are progressively traded off for the opportunities of interaction.

As we saw in Chapter 5 the use of the hiding principle necessarily implies an interaction with the user, who needs to communicate which part of the graphs needs to be hidden or shown. Thus, using the hiding principle corresponds to constraining the solution to an interactive system (and user) and to narrowing the scope of application of the drawing method.

6.4 Radical Versus Smooth Approaches

A drastic distinction between what is shown and what is concealed of the input graph, of what is emphasized and of what is de-emphasized, is a clear and simple discipline both for the designer to implement and for the user to relate with.
6.4. RADICAL VERSUS SMOOTH APPROACHES

When dealing with large graphs, though, radical dichotomies are not so effective as more flexible approaches, in which different levels of gradation can be appreciated, and intermediate solutions between the extreme ones are allowed.

Most of the systems for the visualization of large graphs depart from the simplicity of radical and drastic approaches, moving towards more sophisticated scenarios, in which what is shown continuously shades into what is hidden, what is emphasized into what is de-emphasized, etc.
Chapter 7

Strategies

The two basic principles of hiding and stressing introduced in the Chapter 5 find a number of strategies exploiting them in different and particular ways. These strategies, in their turn, are used, alone or in combination, by existing systems.

7.1 Windowing, Panning, and Zooming

Windowing is an elementary hiding strategy. It consists of presenting the user with a partial view of the whole drawing falling inside a given rectangular area. Since the objects shown are selected on the basis of their geometric position, this hiding technique belongs to the category of geometric hiding, and has no effect on the efficiency of the visualization.

The windowing strategy is traditionally used in information visualization at large. The technique is, for example, effectively used for the visualization of textual information, where a scrollbar allows the user to scroll up and down the text, while the horizontal line length usually, but not necessarily, fits into the screen width.

In graph drawing the graphic information is not organized sequentially as in text displaying. As a consequence, the only “initial view” that could be conceived is a view of the whole graph. For the same reason, there is not a preferred vertical or horizontal scrolling direction and, since the translation of the window has to be performed with respect to an arbitrary direction, it is generally referred to as panning, rather than scrolling.

Other than panning, the user may zoom on the points of interest, changing the scale of the objects represented, focusing on a particular area, and sacrificing the boundaries of the current view. The operation of rotating the window is usually not allowed, for its disorienting effects on the user.

A peculiarity that makes zooming especially well-suited for graph draw-
ing is that, due to the simplicity of the graphics that represent vertices and edges (usually lines and elementary geometric forms), zooming can be easily performed by adjusting screen transformations and redrawing the screen’s contents from an internal representation, in contrast with complex aliasing problems arising any time a zoom operation is performed on a pixel image [67] such as text fonts.

7.2 Multiple Windows and Rapid Zooming

A well-known problem with the windowing technique lays with the radicality of how the hiding principle is applied: objects that are not directly shown to the user, are definitely hidden from the user’s view. The undesired consequence is that zooming on a focus results in a loss of all the contextual information falling outside the current window, and such loss of context may become a considerable usability obstacle. This problem is sometimes referred to as the problem of focus and context, that is: trying to reconcile an exploration at an arbitrary level of detail with a high level view of the whole information displayed.

Windowing offers only poor solutions to the problem of focus and context. A first solution consists in providing the user with multiple windows. Typically, a secondary window contains a navigation chart: a shrunk overview of the whole graph, in which the position of the primary window is shown as a rectangular frame. From the chart the user can infer which part of the drawing is currently shown in the detailed view.

Figure 7.1: A snapshot of the Astra system, showing a navigation of a hierarchical graph with multiple windows.
This solution has two main inconveniences: first, the adoption of a second drawing divides the user's attention between two different targets and may be disorienting [100], and, secondly, the information regarding the immediate neighborhood of the current window, presumably the most relevant for the user to infer the context, is available as much (or as less) as the information regarding the farthest part of the drawing. A second solution to the focus and context problem is offered by rapid zooming [111]. In rapid zooming the user is given the ability to rapidly zoom in and out of points of interest, so that, even if focus and context are not simultaneously available, the user can rapidly and smoothly move from focus to context and back, and integrate the information.

7.3 Warped Visualizations

A set of strategies have been devised in order to better approach the focus and context problem. These techniques are sometimes addressed as focus-context techniques [85, 67, 100], since they aim to simultaneously conveying focus and context information in a single view.

The basic idea is that of spatially distorting the drawing. Distortion gives more room to designated points of interest and decreases the space given to those objects away from these points. Some techniques work with a single focus [85], others allow multiple foci to be simultaneously expanded [104].

![Figure 7.2: A fisheye view of a hierarchical graph.](image)

Such warped visualizations have several flavors and several names, including focus-context [118, 85], nonlinear magnification [75], fisheye views [122, 58], and pliable surfaces [18].

In hyperbolic layout the graph is laid out on a hyperbolic plane or space, and the drawing so obtained is projected on the Euclidean equivalent to ob-
Figure 7.3: A clustered graph: in (a) the common nature of red, green, and blue vertices, is stressed by drawing them near one to the other. In (b) they are collapsed in a single vertex (the squares) representative of all the vertices of the cluster.

7.4 Clustering

The clustering strategy assumes that additional information about the graph is available. Such additional information describes a significant homogeneity between subsets of vertices. This concept can be generalized with the notion of clustered graph [42, 40, 51], in which a recursive partitioning of the vertices can be recognized. More formally, a clustered graph \( C = (G, T) \) consists of an undirected graph \( G \) and a rooted tree \( T \), such that the leaves of \( T \) are exactly the vertices of \( G \) (a similar definition of “nested graph” can be found in [103, 104]).

Since the clustering strategy needs additional information, it corresponds to the pattern of sacrificing generality to push effectiveness, and since the homogeneity of the vertices is emphasized by drawing them inside the same area, it belongs to the class of positional stressing.

The information about homogeneity may be used to leverage the hiding principle as well, since vertices belonging to the same cluster may be collapsed into the same graphic object. This may be particularly effective in an interactive setting, in which the user is allowed to specify which cluster has to be drawn expanded or collapsed, favoring a top-down approach to the graph. This notion is a generalization of the one formalized in [41, 51], in which the view at level \( i \) of a clustered graph \( C = (G, T) \) consider all the vertices at the same level to be expanded, and is defined as a graph \( G_i \) whose vertices are the clusters at level \( i \) of \( C \) and featuring an arc between two vertices only if there is an arc in \( G \) between two vertices belonging to the corresponding clusters.

With respect to the tradeoff between radical and smooth solutions, the
7.5 Spanning Tree Reduction

The spanning tree reduction strategy consists in simplifying the input graph by considering only a spanning tree of the graph itself. Such simplification makes sense only when the input graph is very near to a hierarchical graph (quasi-hierarchical graph) or when additional information is available about the weight of the edges in order to produce a spanning tree that significantly reflects the internal structure of the input graph. Spanning tree reduction is a semantic hiding technique. Some system optionally display the hidden edges, which however are not considered when computing the layout of the graph.

7.6 Pruning

The pruning strategy consists of a semantic hiding of part of the graph based on the topological distance of the vertices from a specific vertex chosen as the center of the visualization. If the graph is a tree, the initial center of the visualization may be the root of the tree, considered as the point of view, i.e., the center of the attention of the user.

The user is allowed to shift the focus from the current point of view to an adjacent vertex. As the point of view changes, the topological neighborhood changes too, since a certain number of vertices are eliminated from it and a
certain number of vertices are inserted. The purpose of the drawing methods is to preserve the user mental map by minimizing the changes in the common parts of two subsequent views. To this purpose a look-ahead mechanism can be exploited, taking into account a larger neighborhood of the current point of view, and considering in advance the effects of the moves of the user. This problem is considered in detail in Chapter 16.

7.7 Compacting

Compacting corresponds to investing some resources with the purpose of computing a drawing with reduced size, increasing in this way the effectiveness of the visualization at the expense of the efficiency of the drawing method. It corresponds to a global stressing, since the property of being drawn in the minimal area is a global property and not a local one.

In Chapter 9 the problem of compacting an orthogonal representation with respect to the area is proved to be NP-complete. In Chapter 10 the two analogous problems of compacting an orthogonal representation with respect to the maximum edge length, or total edge length are proven to be NP-complete too. Also, in the same chapter, it is proved that finding an approximate solution to these problems may be as hard as actually finding the optimum solution, or, in more formal words, that the three problems do not admit a polynomial time approximation scheme.

7.8 Exploiting the Three-Dimensional Environment

One popular technique to represent large graphs is to use a three-dimensional instead of a two-dimensional space. The explicit hope ([67]) is that the extra dimension will offer, literally, more “space” to place the objects, easing the problem of displaying large structures.

It is also possible to use the extra spatial dimension to encode semantic information. For example, Koike [82] produced a 2D layout of the modules in a message passing system, using the third dimension as a time axis to display the message sequencing information.

Technological advances in computer graphics have made three-dimensional information visualization feasible on personal computers. In fact, low-price high-performance 3D graphic workstations are becoming widely available, and the impressive achievements of the research on computer graphics of the last 20 years are now fully transferred to industrial and commercial products.

Three-dimensional graph drawing has both the characteristic of a hiding and of a stressing approach: while the whole graph is presented to the user, a portion of it is directly shown, suitably enlarged by the perspective distortion.
Conversely, the remaining part of the graph is in the background, dwindled proportionally with the distance from the user, and possibly occluded by the foreground objects.

Interaction is natural in three-dimensional layout, both because the part of the drawing that is in the background can be explored only moving the point of view or rotating the object, and because the movement itself helps the user perceiving the three-dimensional structure of the graph. Experiments involving a 3D graph-tracing task showed that test subjects were able to comprehend about three times as much information in an interactive, three-dimensional environment as in a comparable two-dimensional one, when both stereo viewing and motion parallax information were available [136, 135].

The interest of the 3D graph drawing community has been mainly devoted to straight-line drawings and to orthogonal drawings. Straight-line drawings map vertices to points and edges to straight-line segments. Many different approaches to the construction of straight-line drawings can be found in the literature. For example, the method presented in [23] distributes the vertices of the graph along a "momentum curve" so that there are no crossings among the edges. The produced drawings are then "compressed" into a volume (volume of the smallest box enclosing the drawing) of $4n^3$, where $n$ is the number of vertices of the graph to be drawn. The same paper presents another algorithm which constructs drawings without edge crossings of planar graphs with degree at most 4. It "folds" a 2-dimensional orthogonal grid drawing of area $h \times v$ into a straight-line drawing with volume $h \times v$.

Another classical approach of the graph drawing field is the force directed one [30], which has been exploited in 3D graph drawing to devise the algorithms presented in [17, 26, 35, 96, 137, 55, 106].

Further, the research on straight-line drawings stimulated a deep investigation on theoretical bounds. Examples of bounds on the volume of a straight-line drawing can be found in [23, 107]. Namely, in [23] it is shown that a graph can be drawn in an $n \times 2n \times 2n$ volume, which is asymptotically optimal. In [107] it is shown that, for any fixed $r$, any $r$-colorable graph has a drawing with volume $O(n^2)$, and that the order of magnitude of this bound cannot be improved.

Special types of straight-line drawings have been studied in [13, 50, 3, 62] (visibility representations) and in [89] (proximity drawings).

Three-dimensional orthogonal drawing algorithms are presented in [10, 44, 45, 109, 139, 140, 110, 46, 21, 90].

In Chapter 11 is presented a novel approach to three-dimensional orthogonal grid drawing, explicitly aimed at producing readable layouts, rather than meeting some theoretic bound.

In Chapter 13 of the present thesis most of the three-dimensional drawing methods from the literature are considered and experimentally compared. In
order to evaluate their usability for information visualization purposes, we measure the computation time and three important readability parameters: volume, average edge length, and average number of bends along edges.
Chapter 8

Analysis of Existing Systems

Different systems, by exploiting the strategies discussed above, cope with the problem of effectively visualizing large graphs. To create a usable visualization system requires dozens of design decisions, and any one of them, if badly made, may move the system below the threshold of usability. Thus, each visualization system offers a unique collection both of visualization requirements and original solutions. In this chapter we describe the systems corresponding to fully working industrial or commercial products, well above the threshold of usability. Our inquiry points out that, although several experimental and research projects exist, to our knowledge very few of them correspond to fully fledged systems, capable of being used in practice to handle specific visualization problems where large data sets are involved.

8.1 Cone Trees

The Cone Tree ([120]) is one of the best known three-dimensional layout method in information visualization, in which the three-dimensional environment was explicitly used with the purpose of coping with huge amount of data. The creators of Cone Trees claim that approximately one thousand vertices are representable using a 3D structure, considerably more than could be understood in 2D.

The graph to be displayed is assumed to be hierarchical, or, alternatively, a spanning-tree semantic hiding is performed on the graph to reduce it to a tree.

Mathematically, the layout process is quite simple. Different visualization systems made by others reimplemented it ([19, 64, 72]), sometimes refining the original idea. A vertex is placed at the apex of a cone with its children placed evenly along the circle of its base. In the original formulation, each layer has cones of the same height and the cone base diameters for each level
are reduced in progression so that the bottom layer fits into the width of what the authors called the “room”, i.e., the box containing the full cone tree.

Carrière and Kazman [19] propose a variant that calculates an approximation for the diameter for each cone base by traversing the tree bottom-up and by taking the number of descendants into account at each step to make better use of the available space. Jeong and Pang [72] replace the cones with discs in order to reduce occlusion.

Cone Trees represent one of the first systems successfully exploiting a three-dimensional environment for large graph visualization, although the use of a spanning tree reduction of the input graphs limits the application of this drawing method to quasi-hierarchical graphs.

### 8.2 Site Manager

The Site Manager [98] is a product for webmasters from SGI, and incorporates the H3 and H3Viewer libraries [97, 99]. A three-dimensional hyperbolic view is used, in conjunction with a traditional 2D browser view of the directory structure, to explore the hyperlink structure of huge web sites, for developing or debugging purposes.

The graph, supposed to be hierarchical or quasi-hierarchical, is drawn on the hyperbolic space using a tree layout algorithm similar to the Cone Tree one. The cone angle is 180° and the child vertices rather than being disposed on the circumference of the cone base are distributed on the surface of a hemisphere that covers the mouth of the cone. The drawing so obtained is
projected on a sphere of the Euclidean space with a finite projection called projective model (straight lines are preserved but angles are distorted). As a consequence, while the center of the sphere corresponds to the point of interest (and its immediate neighborhood is barely distorted) the surface of the sphere corresponds to points at infinite distance from such point. The user is allowed to explore the graph by changing the point of interest and by rotating the sphere.

Figure 8.2: A snapshot of the Site Manager system.

Site Manager offers an example of how a warped visualization technique can be coupled with a three-dimensional environment. Furthermore on the input graph a spanning-tree semantic hiding is performed, but, optionally, the hidden edges can be shown.

8.3 NicheWorks

NicheWorks [138] is a visualization tool created by the Bell Labs Visualization Group for the investigation of very large graphs (between 20,000 and 1,000,000 vertices\). Originally designed for telephony applications, was used also for the visualization of the relationships in a large software development effort, for web site analysis, and for correlation analysis in large databases.

\[1\] It takes more than one day, though, for NicheWorks to layout a graph with one million vertices
Figure 8.3: Circular layout (a), hexagonal layout (b), and tree layout (c) produced by the NicheWorks system on web site data.

The layout algorithm comprise two successive steps. First an initial placement of the vertices on the plane is found by using one of the three available methods: circular layout, hexagonal layout, or tree layout (see Fig. 8.3 for an example). Secondly, the layout is refined by iteratively launching one of the following:

- a force directed method, without repulsive forces, called Steepest Descent
- a simulated annealing algorithm that swaps the positions of a pair of vertices based on the same potential function used by the Steepest Descent method, and
- a repelling algorithm that computes the nearest neighbors of each vertex and moves the closest ones apart a small distance.

As for the visualization, the NicheWorks system allows vertices to be deleted, that is concealed. The shown vertices are differently stressed depending on whether they are normal (simply shown), highlighted (emphasized), or focused (emphasized and detailed as much as possible).

8.4 NestedVision3D

NestedVision3D (NV3D for short) is a three-dimensional visualization system designed to aid understanding the structure of large computer programs [111]. The information is modeled as a nested (that is, clustered) graph. The system has been tested with graphs containing more than 35,000 vertices and 100,000 relationships, and has been applied to projects in software reverse engineering, particularly for Y2K malfunctioning diagnosis.
8.5 Other Research Projects and Prototypes

From the sections above it is apparent that very few systems are currently available for visualizing large graphs. Munzner states in her recent Ph.D Thesis ([100], page 17) that “Nicheworks is the only graph drawing system to date besides our own H3 system that scales to large datasets.” (Where “large” means well other 100,000 vertices.)

However, some tools developed with the purpose of testing research hy-
hypothesis and validating solutions offer a small collection of choices and alternatives that are worth while considering when coping with specific visualization problems. A limited list of such systems includes:

Part III

The Efficiency Limits
Chapter 9

The Complexity of Orthogonal Compaction

Investing resources with the purpose of reducing the size of a drawing is a possible strategy to increase the effectiveness of the visualization. In this chapter the problem of compacting an orthogonal drawing with respect to the area is proven to be hard. This result confirms a long surviving conjecture of NP-hardness, justifies the research about applying sophisticated, yet possibly time consuming, techniques to obtain optimally compacted orthogonal grid drawings, and discourages the quest for an optimally compacting polynomial-time algorithm.

9.1 Orthogonal Compaction

The name of the last step of the topology-shape-metrics approach, the compaction step, originates from the fact that during this step, while determining the final coordinates of the vertices and bends, an aesthetic measure between area, total edge length, or maximum edge length is hopefully minimized. The compaction problem is precisely the optimization problem consisting of minimizing one of the three mentioned measures, while performing the compaction step: in particular we call Orthogonal Area Compaction (OAC), Orthogonal Total Edge Length Compaction (OTELEC), and Orthogonal Maximum Edge Length Compaction (OMELC) the three problems, respectively. In this chapter we deal with the OAC problem, while the OTELC and OMELC problems are considered in Chapter 10.

Finding the intrinsic computational complexity of the compaction problem has been for a long time an elusive goal. Decades of research in the field of orthogonal graph drawing have not affected our knowledge in this respect: the problem is mentioned as open in recent papers as in foundational ones.
([131, 68, 81]). As far as we know, the only contribution to this subject is the early result contained in [36], where the trivial case of not connected graphs is demonstrated to be NP-complete. Also, the problem of finding a minimum area layout of a graph in a variable embedding setting is a well studied one [84, 53], and was shown to be NP-hard even for planar graphs [53].

The compaction problem has been one of the challenging tasks in the VLSI research field too, where the requirement of minimizing the size of a circuit layout while preserving its shape, led to formulations similar to those arising in the graph drawing area, although, for VLSI purposes, vertices are possibly replaced by squares and additional constraints (e.g., on the length of specific edges) are generally managed. Since several VLSI formulations, related with the compaction problem, are proved to be NP-hard [87], compacting orthogonal representations is widely believed to be an NP-hard problem too, and heuristics producing suboptimal solutions are applied in all practical cases.

A first strain of heuristics descend from the “rectangular refinement” approach proposed in [127], based on the fact that the compaction problem is tractable when all faces of the orthogonal representation are rectangular, and consisting of splitting the non rectangular faces into rectangles and removing the introduced edges after compaction. This approach may yield a linear time compaction step that minimizes the area, or an $O(n^{7/4} \log n)$ compaction step that minimizes the area and (secondarily) the total edge length [30].

Recently, the compaction step has been the subject of a renewed research interest. The problem of optimal compacting with respect to total edge length was approached with an ILP formulation in [79], relying on branch-and-cut or branch-and-bound techniques to find an optimal solution. Lately, a novel compaction method has been devised that optimizes with respect to the area (and, secondarily, total edge length) in polynomial time in the particular, though relatively frequent, case of turn-regular orthogonal representations ([16]). The latter approach gives raise to new heuristics based on a “turn regularization” rather than a “rectangularization” preprocessing step. Finally, in [78] an experimental comparison of orthogonal compaction heuristics was presented and their computational time and results were compared with the values of the optimal compaction yielded by the algorithm in [79].

This chapter is concerned with the complexity of producing an orthogonal grid drawing $\Gamma$ starting from its orthogonal representation $H$ while minimizing the area of the drawing.

In what follows we will consider, without loss of generality, only orthogonal drawings with no bends, since each bend can be replaced by a dummy vertex of degree two.

Following a standard technique (see, e.g., [59, 108]), rather than address directly the optimization problem we will consider its corresponding decision version, according to which the Orthogonal Area Compaction problem con-
9.2. NP-HARDNESS OF THE OAC PROBLEM

sists in taking as input an orthogonal representation $H$ of a graph $G$ and a constant $K$, and deciding whether integer coordinates can be assigned to the vertices of $G$ so that the area of the drawing is less or equal than $K$. More formally they can be defined as follows:

*Problem: Orthogonal Area Compaction (OAC)*

*Instance:* An orthogonal representation $H$ of a graph $G$ and a constant $K$.

*Question:* Can integer coordinates be assigned to the vertices of $G$ so that the area of the drawing is less or equal than the value of the constant $K$?

We will show in the following sections that the OAC problem is NP-hard and is in NP. This is summarized in the following theorem:

**Theorem 6** The OAC problem is NP-complete.

9.2 NP-hardness of the OAC Problem

In this section, by means of a reduction from the SAT problem, we prove that compacting an orthogonal representation of a connected graph, while minimizing its area is an NP-hard problem. To accomplish this, we introduce a class of orthogonal representations, that we call sliding-rectangles gadgets, admitting an exponential number of orthogonal grid drawings with minimum area, in all of which the basic blocks composing the gadget necessarily inherit the property of being themselves drawn with the minimum area. This property is exploited to build a sliding-rectangles gadget corresponding to a formula $\phi$ of the SAT problem. We will prove the NP-hardness of the OAC problem by showing that such orthogonal representation admits exclusively the subset of orthogonal grid drawings with the minimum area corresponding to the truth assignments satisfying the formula $\phi$.

We briefly recall here the SAT problem:

*Problem: Satisfiability (SAT)*

*Instance:* A set of clauses, each containing literals from a set of boolean variables.

*Question:* Can truth values be assigned to the variables so that each clause contains at least one true literal?

Given a formula $\phi$ in conjunctive normal form with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, we will produce an orthogonal representation $H_A(\phi)$ and a constant $K_A(\phi)$ such that an orthogonal grid drawing of area less or equal than $K_A(\phi)$ exists if and only if $\phi$ is satisfiable.
Notice that in the SAT definition all the variables in the same clause can be assumed to be different, i.e., the version of SAT in which each clause contains appearances of distinct variables is also NP-complete (this can be trivially proved by introducing a linear number of dummy variables and further clauses).

In what follows, by using the gadget introduced in subsection 9.2.1, we will show how to build the instance \((H_A, K_A)\) of the OAC problem corresponding to an instance \(\phi\) of the SAT problem (subsection 9.2.2), and prove that a solution to the OAC problem on instance \((H_A, K_A)\) exists if and only if the corresponding instance \(\phi\) of the SAT problem is satisfiable (subsection 9.2.3).

### 9.2.1 Sliding-Rectangles Gadget

The main problem in the construction of an instance \((H_A, K_A)\) is the fact that the property of being drawn with minimum area is a global property, regarding the whole drawing, and does not necessarily reflect on parts of it. Fig. 9.1.a provides an example in which the area covered by the external box is minimum, while the subgraphs contained inside are not themselves drawn as small as they could be. This is a drawback since we would like to devise a chain of causes and effects leading from a minimized drawing to a satisfied formula and vice versa, but our only property seems to have no consequences if not on the boundary.

![Figure 9.1](image)

(a) (b) (c)

Figure 9.1: While the orthogonal grid drawing in (a) occupies the minimum area, its subgraphs are not themselves drawn with minimum area. Conversely, in (b) and (c), the property of being drawn with minimum area is both a global and a local property.

Obviously, we could change the size of the external box so to force the global optimality to imply a local optimality, as in Fig. 9.1.b, or add a suitable number of objects to the drawing to obtain the same effect (Fig. 9.1.c), but by doing this we limit the number of optimal solutions, i.e., we tend to produce orthogonal representations admitting only one orthogonal drawing with the minimum area.
Instead, since we expect the compacted drawing to correspond to a satisfying assignment and vice versa, we are seeking for an orthogonal representation that admits as many compacted drawings as there are assignments that satisfy the formula (possibly an exponential number).

What we need is a systematic way to assure that the area optimality is inherited by the internal parts of the grid drawing while preserving a suitable degree of “freedom” (i.e., number of alternatives) for the orthogonal grid drawing of the whole graph.

To this purpose we produce the sliding-rectangles gadget of Fig. 9.2. We hypothesize that the inside subgraphs can be modeled by $n$ contiguous $(3, h)$-compactable rectangles. Each rectangle can slide vertically with respect to the following and preceding ones. The box around the graph has top and bottom side $4n + 4$ vertices long and right side $h + 7$ vertices long. Between the rectangles and the external boundary we place a belt consisting of a path of $4 + 8n$ vertices. The first rectangle, the belt, and the external box are linked together as shown in Fig. 9.2. Instead of giving the $\alpha$ values for each vertex of the belt, we will describe its angles specifying the turn sequence $\sigma = (r^{tl^4})^4r^4$, where an $r$ represents a right turn, and an $l$ represents a left turn and $r^n$ ($l^n$) represents a repetition of $n$ right turns (left turns, respectively). The turn sequence $\sigma$ succinctly describes the angles met when traversing the path clockwise starting from the vertex shared with the rest of the graph, and will be used in the following for the sake of brevity.

Note that, since four right turns are both at the beginning and at the end of the belt, the turn sequence $\sigma = (r^{tl^4})^4r^4$, may be equivalently written as $r(r^{3l^4}r^{3l^4})^nr^4$ or $r^3(r^{tl^4}r^3)^nr$. It follows that removing a $r^{3l^4}r$ subsequence has the same effect of removing a $r^{tl^4}r^3$ one. What’s more, removing a $r^{3l^4}r$ subsequence and inserting a $r^{tl^4}r^3$ one in the correct place (before the trailing $r$, for example) leaves the whole turn sequence unchanged.
Figure 9.3: (b) An orthogonal drawing with the minimum area of a sliding-rectangle gadget. When a rectangle slides from the “down” position to the “up” position, the darkened area in (a) is covered and the darkened area in (d) is available; a $r^3l^4r$ subsequence (black vertices in (a)) can be removed, and a $rl^4r^3$ subsequence (black vertices in (c)), can be inserted in the turn sequence.

The reason why we are interested in removing one subsequence and inserting the other is apparent from Fig. 9.3: Suppose to slide a rectangle upwards of three horizontal grid lines, a $r^3l^4r$ subsequence on the upper side may be removed, since it is partially covered by the rectangle. At the same time the new room made on the bottom side of the rectangle is exactly what is needed to host a $rl^4r^3$ subsequence.

**Lemma 1** In each orthogonal grid drawing of the sliding-rectangles gadget with the minimum area $(4n+3) \times (h+6)$:

1. each rectangle is drawn in the minimum area, and
2. each rectangle assumes necessarily one of the two positions “up” and “down” depicted in Figure 9.3.

**Proof:** Consider the areas available to the belt above and below each rectangle in an orthogonal grid drawing with the minimum area of the sliding-rectangles gadget. If the rectangle is drawn itself in the minimum area and is in the down (up) position, 8 vertices of the belt can be placed in the space above (below), as shown in Fig. 9.3.b.

In order to show that an orthogonal grid drawing in the minimum area in which a rectangle is not drawn itself in the minimum area or has an intermediate position does not exist, we need to show that the belt needs all the areas above or below the rectangles, and that if a rectangle has an intermediate position, the two areas above and below the rectangle can not host the same number of vertices of the belt.
Observe that the edges of the belt are alternating horizontal and vertical segments, and the vertical edges alternate between the four types \( t_1, t_2, t_3, \) and \( t_4 \), depicted in Fig. 9.4.a, 9.4.b, 9.4.c, and 9.4.d, respectively.

Suppose that the rectangle is in the upper or lower position. Four horizontal grid lines are available to the belt. We prove that no more than eight vertices of the belt can be hosted in the four horizontal grid lines. In fact, from Fig. 9.4 it is apparent that a type \( t_1 \) or a type \( t_3 \) vertical edge needs at least three grid points of the vertical grid line it belongs to, and thus a vertical grid line hosting a type \( t_1 \) or a type \( t_3 \) vertical edge cannot host another vertical edge. The same holds for a vertical grid line hosting a type \( t_2 \) or \( t_4 \), since this would force the belt to traverse the vertical grid line two times in opposite directions, and therefore being trapped in the same side of the grid line, or to traverse it two times in the same direction, which is impossible without a third traversal in the opposite direction. Since each vertical segment takes one vertical grid line, all the space above or below the rectangles is used by the belt.

If a rectangle has an intermediate position it leaves less than four horizontal grid lines to the belt in the space above and below the rectangle, and fails to host as many vertices (observe, for example, that a type \( t_1 \) or \( t_3 \) vertical edge cannot be hosted in such a space, since the belt would be trapped in the same side of the vertical grid line hosting the edge).

It follows that an orthogonal grid drawing in the minimum area of the sliding-rectangle gadget in which a rectangle is not drawn itself in the minimum area or has an intermediate position does not exist.

![Figure 9.4: The vertical segments of the belt are of the four type \( t_1, t_2, t_3, \) and \( t_4 \), represented in (a), (b), (c), and (d), respectively.](image)

From the above lemma, it follows that the sliding-rectangles gadget admits an exponential number of orthogonal grid drawings with the minimum area.

The following properties introduce some variants to the sliding-rectangles gadget, for which an accordingly modified version of Lemma 1 holds. The proofs are omitted for brevity.

**Property 1** In the sliding-rectangles gadget, a \((w', h + 3)\)-compactable rectangle, with \( w' \) arbitrary, can be inserted at any position between the sliding-
rectangles, provided that \( w' + 1 \) vertices are added to the top and bottom sides of the external box.

**Property 2** In the sliding-rectangles gadget, a \((3, h)\)-compactable rectangle can be replaced by a \((3+4c, h)\)-compactable one, where \( c \) is an arbitrary positive integer, provided that \( 4c \) vertices are added to the top and bottom sides of the external box, and a \((r^{4^{k}})^{c}\) sub-sequence is inserted at the beginning of the turn sequence of the belt.

Fig. 9.5 shows a sliding-rectangles gadget featuring both variants.

Also, it is easy to verify that if the rectangles of a sliding-rectangles gadget are allowed to assume only a subset of all the possible (otherwise exponential) configurations, an accordingly modified version of Lemma 1 holds, stating that only the corresponding subset of orthogonal grid drawings with minimum area are admitted.

### 9.2.2 Instance \((H_A, K_A)\) Construction Rules

In this subsection we describe how to construct an instance \((H_A(\phi), K_A(\phi))\) of the OAC problem corresponding to an instance \( \phi \) of the SAT problem, in such a way as to incorporate a sliding-rectangles gadget.

The construction of the orthogonal representation \( H_A(\phi) \) requires three steps:

i) build a clause-gadget for each clause \( C_i \),

ii) combine clause-gadgets together, and

iii) add external boundary and belt.

These three steps are described in the following three paragraphs. The fourth paragraph is concerned with producing a value for \( K_A(\phi) \).
9.2. NP-HARDNESS OF THE OAC PROBLEM

i) Clause-gadget Construction

In the following we assume that the formula $\phi$ of the SAT problem has $n$ boolean variables, $x_1, \ldots, x_n$, and $m$ clauses $C_1, \ldots, C_m$.

The clause-gadget is composed by $n$ chambers, one for each variable, whether the variable actually occurs in the clause $C_i$ or not. We call $(i, j)$-chamber the chamber of clause $C_i$ corresponding to the variable $x_j$. The $(i, j)$-chamber, with $1 < j < n$ is shown in Fig. 9.6a, while the $(i, 1)$-chamber and the $(i, n)$-chamber are shown in Fig. 9.6b and 9.6.c, respectively.

Figure 9.6: First line: the chambers corresponding to: (a) a variable $x_j$, with $1 < j < n$; (b) the variable $x_1$; and (c) the variable $x_n$. Second line: true-compliant orthogonal grid drawings corresponding to the orthogonal representations of figure (a), (b), and (c), respectively. Third line: false-compliant orthogonal grid drawings corresponding to the same orthogonal representations.

Observe that the edge lengths of Fig. 9.6a, 9.6.b, and 9.6.c are not meaningful, since such figures are only meant to describe orthogonal representations. However some particular orthogonal grid drawings of the chambers will be so recurrent in what follows as to deserve a definition: we define true-compliant an orthogonal grid drawing of a chamber such that the vertex distances are
Chapter 9. The Complexity of Orthogonal Compaction

Figure 9.7: (a) The chambers corresponding to a clause of a formula with six variables. The black vertices are weldings. The obstacles inserted in a \((i,j)\)-chamber, when variable \(x_j\) does not occur in clause \(C_i\) (b); when variable \(x_j\) occurs in the clause \(C_i\) with a positive literal (c); or a negative literal (d). The figure shows only the case of an \((i,j)\)-chamber with \(1 < j < n\); other cases are similar.

Exactly those represented in Fig. 9.6.d, 9.6.e, or 9.6.f, and false-compliant an orthogonal grid drawing of a chamber such that the vertex distances are exactly those represented in Fig. 9.6.g, 9.6.h, or 9.6.i.

For the sake of brevity we call compliant a true-compliant or false-compliant orthogonal grid drawing of a chamber, and we say that, in a given orthogonal grid drawing \(\Gamma\) of \(H_n\), a chamber is true-compliant (false-compliant, compliant, respectively) whenever the orthogonal grid drawing of the chamber induced by \(\Gamma\) is true-compliant (false-compliant, compliant, respectively).

All the \(n\) chambers corresponding to clause \(C_i\) are attached together in a row, in such a way that the \((i,j)\)-chamber shares two vertices with the \((i,j+1)\)-chamber. We call such vertices weldings. Fig. 9.7 shows all the chambers of a clause-gadget for a formula with six variables.

To be completed the clause-gadget is added with two types of further subgraphs: obstacles, and pathways. An \((i,j)\)-chamber corresponding to a variable \(x_j\) not occurring in the clause \(C_i\) receives an obstacle as shown in Fig. 9.7.b. Any other \((i,j)\)-chamber receives two obstacles as shown in Fig. 9.7.c, if the variable \(x_j\) occurs with a positive literal, or as shown in Fig. 9.7.d, otherwise. Fig. 9.8.a shows an example of a clause-gadget with its obstacles.

Finally, the clause-gadget is augmented with a pathway composed by a
succession of $2n - 1$ A-shaped structures linked together as shown in Fig. 9.8.b. The pathway originates from the $(i, 1)$-chamber and terminates in the $(i, n)$-chamber, as shown in the same figure.

ii) Combining Clause-gadgets Together

All the clause-gadgets corresponding to formula $\phi$ are placed one upon the other, so that each $(i, j)$-chamber shares its bottom 8 vertices with the $(i+1, j)$-chamber, for $i = 1, \ldots, n - 1$. Furthermore, hinges are introduced. Hinges are vertical paths, originating from the weldings.

A hinge 8 vertices long links the welding between the $(i, j)$-chamber and the $(i, j+1)$-chamber with the welding between the $(i+1, j)$-chamber and the $(i+1, j+1)$-chamber, for $i = 1, \ldots, m - i$, and $j = 1, \ldots, n - 1$.

A hinge 6 vertices long attaches to the welding between the $(i, j)$-chamber and the $(i, j + 1)$-chamber, with $i = 1$ or $i = n$, and $j = 1, \ldots, n - 1$. The clause-gadgets and hinges for a formula with five variables and four clauses are shown in Fig. 9.9.a.

iii) Adding External Boundary and Belt

To obtain the final orthogonal representation $H_A(\phi)$ an external boundary and a belt are added to the construction. The external boundary has a top and bottom sides of $9n + 3$ vertices and a right side of $9m + 8$ vertices. The
Figure 9.9: (a) Clause-gadgets and hinges for a formula with five variables and four clauses. The black vertices are weldings. The inside of the clause-gadgets is not represented (darkened areas). (b) Adding external boundary and belt to the construction of (a) (darkened area) to obtain the final orthogonal representation \( H_A(\phi) \).

*belt* is a path inserted between the boundary and the core of the construction and composed by \( 2 + 24(n - 1) \) vertices, so that its turn pattern is \((r^4 l^4)^{2n} r^4\). The external boundary, the belt, and the core of the construction are attached together as shown in Fig. 9.9.b.

iv) Computing Constant \( K_A(\phi) \)

The instance \((H_A, K_A)\) of the OAC problem is completely defined as the value of \( K_A(\phi) = (9n + 4) \times (9m + 7) \) is assigned. Fig. 9.10 shows an example of \( H_A(\phi) \) for a formula \( \phi \) with four boolean variables and four clauses with \( K_A(\phi) = 40 \times 43 = 1,720 \).

9.2.3 Correctness

Here we prove that an orthogonal grid drawing \( \Gamma \) of area at most \( K_A(\phi) \) can be found for the orthogonal representation \( H_A(\phi) \) if and only if the corresponding instance \( \phi \) of the SAT problem admits a solution.

The following properties hold:

**Property 3** An orthogonal grid drawing of the orthogonal representation \( H_A(\phi) \) has length at least \( 9n + 4 \), and height at least \( 9m + 7 \).
9.2. NP-HARDNESS OF THE OAC PROBLEM

Proof: It suffices considering the number of vertices on the top side an right side of the external boundary of the orthogonal representation $H_A(\phi)$ (see Fig. 9.9.b).

Property 4 The horizontal (vertical) distance between a welding $v$ and a vertex on one of the vertical (horizontal) sides of the external boundary of $H_A(\phi)$ is the same in every orthogonal grid drawing $\Gamma$ of $H_A(\phi)$ with area $K_A(\phi)$.

Proof: For the horizontal distance it suffices considering that a path $p$ leading from a vertex on the left vertical side, to a vertex on the right vertical side of a clause gadget can be found in $H_A(\phi)$, such that (i) $v$ belongs to $p$, (ii) the coordinates of the vertices of $p$ are non decreasing with respect to the $x$-axis, and (iii) the number of horizontal edges of $p$ is $9n + 2$. Since the distance between the right vertical side of a clause gadget and the right vertical side of the whole drawing is at least two, and the horizontal edges have length at least one, the horizontal position of $v$ in any drawing of length $9n + 4$ is fixed.

Regarding the vertical distance between a welding and the external boundary, notice that, since the area of the orthogonal grid drawing $\Gamma$ is minimum, all the welding between $(i, j)$-chamber and $(i, j + 1)$-chamber, with $i = 1, \ldots, n$ have the same $x$ coordinate, i.e., the hinges that attach to them lie necessarily on the same vertical grid line. Considering the length of the hinges, and that at least a horizontal grid line must be left between two hinges, to host the pathway, and between the external boundary and the hinges, to host the belt, the statement follows.

Property 5 An orthogonal grid drawing of an $(i, j)$-chamber is $(9, 9)$-compactable.

Proof: For the minimum width it suffices considering the number of vertices on the top side of the subgraphs represented in Fig. 9.6.a, 9.6.b, and 9.6.c.

To prove that when the chambers are drawn in the minimum width their minimum height is 9, for $j = 1$ or $j = n$ it suffice consider the number of vertices on the left and right side of Fig. 9.6.b, and 9.6.c, respectively. For other values of $j$, observe that the chamber is composed by two non connected subgraphs, and that at least a line must lay between them to host the pathway.

Property 6 A compliant $(i, j)$-chamber corresponding to a variable $x_j$ not occurring in clause $C_i$ contains two $A$-shaped structures of the pathway.

Proof: The statement follows from the observation that the pathway must necessarily overlap with the dotted lines shown in Fig. 9.11. From the same figure is apparent that the only way to accomplish this is by inserting two $A$-shaped structures of the pathway inside an $(i, j)$-chamber.
CHAPTER 9. THE COMPLEXITY OF ORTHOGONAL COMPACTION

Property 7 A true-compliant (false-compliant) \((i, j)\)-chamber contains two A-shaped structures of the pathway if the corresponding literal is negative (positive), and may contain only one if the literal is positive (negative).

Proof: The proof is obvious considering Fig. 9.12 and Fig. 9.13 where the cases are represented.

Let \( \Gamma \) be an orthogonal grid drawing of \( H_A \). We say that a clause-gadget \( C_i \) is compliant in \( \Gamma \) if each chamber of \( C_i \) is compliant in \( \Gamma \). Also, we define truth configuration of \( C_i \) in \( \Gamma \) as the succession of boolean values \( b_j \), \( j = 1, \ldots, n \), such that \( b_j \) is true (false) if the corresponding \((i, j)\)-chamber is true-compliant (false-compliant).

Lemma 2 A clause-gadget admits a truth configuration \( T \) if and only if assigning the sequence of boolean values of \( T \) to the variables \( x_1, \ldots, x_n \) produces at least one true literal in the corresponding clause \( C_i \).

Proof: By contradiction: suppose that the clause-gadget admits a truth configuration \( T \), and that the sequence of boolean values of \( T \) assigned to the variables \( x_1, \ldots, x_n \) yield a false value for all the literals of the corresponding clause \( C_i \). Each chambers of the clause-gadget corresponds to a variable not occurring in \( C_i \), or occurring with an opposite truth value. Thus, each chamber must contain two A-shaped structures of the pathway (Properties 6 and 7, respectively). It follows that the number of A-shaped structures of the pathway of the clause-gadget should be \( 2n \), while is \( 2n - 1 \).

Conversely, suppose \( T \) is a truth configuration that, when assigned to the variables \( x_1, \ldots, x_n \), produces at least a true literal in the clause \( C_i \). Let \( x_i \) be a variable yielding a true literal. For Property 7 the \((i, t)\)-chamber admits a compliant orthogonal grid drawing containing only one A-shaped structure of the pathway. Since \( 2n - 2 \) A-shaped structures and \( n - 1 \) chambers are left, their drawing may be compliant with the corresponding truth value of the truth configuration \( T \), since two A-shaped structures of the pathway are contained in each of them (Properties 6, and 7).

Lemma 3 The orthogonal representation \( H_A(\phi) \) admits an orthogonal grid drawing of area at most \( K_A(\phi) \) if and only if formula \( \phi \) is satisfiable.

Proof: Suppose formula \( \phi \) is satisfiable and let \( T \) a truth configuration corresponding to an assignment satisfying \( \phi \). Lemma 2 states that each clause-gadget admits an orthogonal grid drawing compliant with \( T \). If all the clause-gadgets are drawn compliant with the truth assignment \( T \), each chamber is drawn in the minimum area, and the vertical column of chambers corresponding to the same boolean variable covers a rectangular area of \( 7 \times 9m \), so is
assimilable to a \((3 + 4c, 9m)\)-compactable rectangle with \(c = 1\) (Property 2). Between each pair of contiguous sliding rectangles a \((0, 9m + 4)\)-rectangle is inserted, as allowed by Property 1. Furthermore, according to the above two variants, the top and bottom sides of the external boundary are \(9n + 3\) vertices long, the right side of the boundary is \(9m + 8\) vertices long, and the belt has a bend pattern \(\sigma = (t^4 t^1)^2 n t^3\). It follows that the orthogonal representation \(H_A(\phi)\) is a sliding-rectangles gadget, and Lemma 1 assures that an orthogonal grid drawing with area \(K_A(\phi)\) exists for each truth configuration corresponding to a truth assignment satisfying \(\phi\).

Conversely, suppose formula \(\phi\) is not satisfiable. Lemma 2 implies that there isn’t a truth configuration that can be assumed by all clause-gadgets. Since each chamber is attached to the chamber below with its bottom-side vertices, it follows that in any orthogonal grid drawing of \(H_A(\phi)\), at least one chamber is not compliant. As a consequence one of the following holds:

1. a chamber has height greater than 9,
2. a clause-gadget has length greater than \(9n\), or
3. all clause-gadget have length equal to \(9n\), and a column of hinges has height grater than \(9m + 4\).

Each of the above three statements implies that the orthogonal grid drawing \(\Gamma\) of \(H_A(\phi)\) has an area greater than \(K_A(\phi)\). In fact:

- case 1 implies that at least one column of chambers has height greater than \(9m\), and Lemma 1 rules out the existence of an orthogonal grid drawing of the sliding-rectangles gadget with area \(K_A(\phi)\) in which a rectangle has an area greater than \(7 \times 9m\).
- case 2 implies that the width of the whole orthogonal grid drawing is greater than \(9n + 4\), and from Property 3 and the definition of \(K_A(\phi)\), the statement follows.
- Finally, case 3 implies analogously that the height of the orthogonal drawing is greater that \(9m + 7\).

**Lemma 4** The OAC problem is NP-hard.

**Proof:** The statement follows from Lemma 3 and from the fact that the orthogonal representation \(H_A(\phi)\) has \(O(n \times m)\) vertices, and its construction (and the computation of \(K_A(\phi)\)) can be done in polynomial time.
9.3 The OAC Problem is in NP

To prove that the Orthogonal Area Compaction problem is in NP we produce a nondeterministic Turing machines that decide it in polynomial time.

The nondeterministic Turing machines that we describe in the following take as input the instance \((H, K)\), and generate the set \(S\) of orthogonal grid drawings of \(H\) with coordinates in the range \([0, v - 1]\), where \(v\) is the number of vertices of \(H\). Then they check each orthogonal grid drawing in \(S\) is to verify in polynomial time if its area, total edge length, or maximum edge length, respectively, is less or equal than the constant \(K\).

It’s easy to show that, if an orthogonal grid drawing \(\Gamma \notin S\) of the orthogonal representation \(H\) exists with area (total edge length, maximum edge length, respectively) less or equal than a constant \(K\), then an orthogonal grid drawing \(\Gamma'^{'} \in S\) exists with equal or less area (total edge length, maximum edge length, respectively). In fact, since our orthogonal representations have no bends, if \(\Gamma\) is an orthogonal grid drawing of \(H\) such that the horizontal (vertical) distance between two of its vertices is bigger than \(v - 1\), then \(\Gamma\) necessarily contains a vertical (horizontal) grid line non intersecting any vertex of \(H\) that can be removed, decreasing the distance of such two vertices, the area, the total edge length and, possibly, the maximum edge length. Furthermore, if \(\Gamma \notin S\) is an orthogonal grid drawing of \(H\) such that the distance of any two vertices is less or equal to \(v - 1\), then an orthogonal grid drawing \(\Gamma'^{'} \in S\) exists with the same area (total edge length, maximum edge length, respectively).

Observe that, since the coordinates of the vertices in the range \([0, v - 1]\), it’s easy to check in polynomial time whether an orthogonal grid drawing \(\Gamma \in S\) is a feasible solution (vertices do not overlap, edges are orthogonal and do not intersect, angles around vertices are coherent with \(H\) labeling), and whether the area, total edge length, or maximum edge length of \(\Gamma\) is less or equal than the constant \(K\).

The nondeterministic Turing machine for the OAC problem works as follows: it takes as input the instance \((H, K)\), and, if \(v\) is the number of vertices of \(H\), writes an arbitrary sequence of \(v\) coordinate pairs in the range \([0, v - 1]\). When this writing stops, the machine goes back and checks to see whether the string written is an orthogonal grid drawing, and, if so, whether its area is less or equal than \(K\).

The following lemma is, therefore, proved:

**Lemma 5** The OAC problem is in NP.
9.3. THE OAC PROBLEM IS IN NP

Figure 9.10: The orthogonal representation $H_A(\phi)$ corresponding to the formula $\phi = (x_2 \lor x_4) \land (x_1 \lor x_2 \lor \overline{x}_3 \lor x_4) \land (\overline{x}_3) \land (x_1 \lor x_2 \lor x_3)$. The particular orthogonal grid drawing shown in the figure has minimum area and corresponds to the truth assignment: $x_1 = false, x_2 = true, x_3 = false, x_4 = true$. 
CHAPTER 9. THE COMPLEXITY OF ORTHOGONAL COMPACTION

Figure 9.11: The first line shows the possible configurations of a compliant \((i, j)\)-chamber corresponding to a variable \(x_j\) not occurring in the clause \(C_i\). The second line shows that the chambers necessarily contain two A-shaped structures of the pathway when are drawn in the minimum area.

Figure 9.12: When variable \(x_j\) occurs in the clause \(C_i\) with a positive literal, a true-compliant \((i, j)\)-chamber (a and c), may contain only one A-shaped structure of the pathway (e and g, respectively), while a false-compliant \((i, j)\)-chamber (b and d) contains necessarily two A-shaped structures (f and g, respectively).
Figure 9.13: When variable $x_j$ occurs in the clause $C_i$ with a negative literal, a \emph{false-compliant} $(i,j)$-chamber (a and c), may contain only one A-shaped structure of the pathway (e and g, respectively), while a \emph{true-compliant} $(i,j)$-chamber (b and d) contains necessarily two A-shaped structures (f and g, respectively).
Chapter 10

Related Problems

This chapter discusses the problem of compaction with respect to the maximum and total edge length, which are proven to be NP-complete as well.

10.1 NP-Completeness of the OTELC and OMELC Problems

The previous chapter addresses the complexity of producing an orthogonal grid drawing starting from its orthogonal representation while minimizing the area of the drawing. The total edge length, and the maximum edge length are two alternative measures for the size of the drawing that could be used in the compaction step, and give rise to other two different optimization problems. The three criteria of minimizing area, maximum edge length, and total edge length, are considered to have roughly the same aesthetic effect: that of reducing the size of the drawing (or of part of it) and so improving its readability. However, conflicts between the three requirements (see Fig. 10.1) imply that they constitute three different, although closely related, optimization problems.

In an analogous way as for the area minimization problem, we recast the two problems of minimizing the total edge length and maximum edge length as decision problems:

Problem: Orthogonal Total Edge Length Compaction (OTELC)
Instance: An orthogonal representation $H$ of a graph $G$ and a constant $K$.
Question: Can integer coordinates be assigned to the vertices of $G$ so that the total edge length of the drawing is less or equal than the value of the constant $K$?

Problem: Orthogonal Maximum Edge Length Compaction (OMELC)
Figure 10.1: The orthogonal drawings (a) and (b) correspond to the same orthogonal representation, and show how the two requirements of minimizing the area and minimizing the total (or maximum) edge length may not be met by a single drawing (the graph is biconnected and its orthogonal representation is “turn-regular” as defined in [16]). The drawings (c) and (d) too correspond to a single orthogonal representation: (c) minimizes the maximum edge length and (d) the total edge length (the graph is biconnected, its orthogonal representation is “turn-regular” and rectangular).

**Instance:** An orthogonal representation $H$ of a graph $G$ and a constant $K$.

**Question:** Can integer coordinates be assigned to the vertices of $G$ so that the **maximum edge length** of the drawing is less or equal than the value of the constant $K$?

To prove that the Orthogonal Total Edge Length Compaction problem is NP-hard we reduce the SAT problem to it by slightly modifying the construction described in Section 9.2.

Observe that, in any orthogonal grid drawing of $H_A(\phi)$ with area $K_A(\phi)$, the total edge length can not be greater than $\ell_0 = (l + 1) \times (h + 1)$, where $l$ and $h$ are the minimum values of the length and height of an orthogonal drawing of $H_A(\phi)$, respectively. To obtain the instance $(H_{TEL}(\phi), K_{TEL}(\phi))$ of the OTELC problem we add to $H_A(\phi)$ a number of $\ell_0$ edges along the top and right sides of $H_A(\phi)$ and connect them to $H_A(\phi)$ as shown in Fig. 10.2.a. We assign to $K_{TEL}(\phi)$ the value $\ell_0(l + 2) + \ell_0(h + 2) + \ell_0 = \ell_0(l + h + 5)$.

If $\phi$ is satisfiable, then for Lemma 3 $H_A(\phi)$ admits an orthogonal grid drawing with area $K_A(\phi)$, and $H_{TEL}(\phi)$ admits an orthogonal grid drawing with total edge length less or equal than $K_{TEL}(\phi)$. Converse, if $H_{TEL}(\phi)$ admits an orthogonal grid drawing of total edge length less or equal than $K_{TEL}(\phi)$, then $H_A(\phi)$ admits an orthogonal grid drawing with area $K_A(\phi)$. In fact, it’s easy to see that every orthogonal grid drawing $\Gamma$ of $H_{TEL}(\phi)$ such that $H_A(\phi)$ covers an area bigger than $K_A(\phi)$ has a total edge length greater than $K_{TEL}(\phi)$. Then, from Lemma 3 follows that the corresponding formula $\phi$ is satisfiable.

Similarly, the SAT problem can be reduced to the Orthogonal Maximum
10.2. Final Remarks

We have shown that compacting an orthogonal representation while minimizing an aesthetic measure between area, maximum edge length, and total edge

Figure 10.2: (a) The orthogonal representation $H_{TEL}(\phi)$, and (b and c) the two possible cases for the orthogonal representation $H_{MEL}(\phi)$. The darkened areas represent the $H_A(\phi)$ orthogonal representation of Fig. 9.9.b.

Edge Length Compaction problem. Namely, to obtain the instance $(H_{MEL}(\phi), K_{MEL}(\phi))$ of the OMELC problem we modify the orthogonal representation $H_A(\phi)$, adding a rectangular box to it, in such a way that the number of vertices along the top side of the obtained orthogonal representation is equal to the number of the vertices along the right side of it (see Fig. 10.2.b and 10.2.c). Finally, we add a pair of edges running along the top and right side of the construction as shown in the same figures.

Since the last added two edges are the longest in any orthogonal grid drawing of $H_{MEL}(\phi)$, an orthogonal grid drawing of $H_{MEL}(\phi)$ that minimizes the maximum length, also minimizes the perimeter of $H_A(\phi)$. In particular, when the maximum edge length of $H_{MEL}(\phi)$ is $K_{MEL}(\phi) = \max(9n + 4, 9m + 7)$, the perimeter of $H_A(\phi)$ is exactly $(9m + 4) \times (9n + 7)$, and the area of $H_A(\phi)$ is exactly $K_A(\phi)$, so that, an orthogonal grid drawing of $H_{MEL}(\phi)$ with maximum edge length equal to $K_{MEL}(\phi)$ exists if and only if the corresponding formula $\phi$ is satisfiable.

The following lemma is then proved:

**Lemma 6** The OTELC, and OMELC problems are NP-hard.

Since similar nondeterministic Turing machines can be easily devised to show that the the OTELC and OMELC problems are in NP, the following theorem is proved:

**Theorem 7** The OTELC, and OMELC problems are NP-Complete.
length is an NP-complete problem and that it doesn’t allow a polynomial-time approximation scheme.

An interesting topic is whether the three problems retain their complexity when focusing on particular classes of graphs. One may ask, for example, what is the influence of the connectivity properties of the graphs. For biconnected graphs, in spite of the fact that the proposed constructions are not biconnected (due to hinges and belt attachment), it’s easy to modify these parts (thickening them as shown in Figure 10.3) so to produce an orthogonal representation \( H_A(\phi) \) of a biconnected graph.

![Diagram](a) ![Diagram](b)

Figure 10.3: Hinges and belt attachment can be thickened as shown in (a) and (b), respectively, to make the orthogonal representation \( H_A(\phi) \) biconnected.

Other interesting problems are the following:

- does an orthogonal representation, whose underlying graph is a simple cycle, retain the complexity of the three general problems?

- does “turn-regularity” (defined in [16]) characterize the orthogonal representations for which the compaction problem is polynomial?
Part IV

Stressing and Hiding in
Three-Dimensional Graph
Drawing
Chapter 11

A Split&Push Approach to Three-Dimensional Orthogonal Drawing

We present a method for constructing orthogonal drawings of graphs of maximum degree six in three dimensions without intersections between edges devised with the explicit purpose of generating readable drawings rather that meeting asymptotic bounds. It can be considered more as a general strategy rather than as a specific algorithm. The approach is based on generating the final drawing through a sequence of steps, starting from a “degenerate” drawing. At each step the drawing “splits” into two pieces and finds a structure more similar to its final version. The new method aims at constructing drawings without any “privileged” direction and with a routing strategy that is not decided in advance, but depends on the specific needs of the drawing. This chapter derives from the material presented at the 6th International Symposium on Graph Drawing, GD'98, Montreal, Canada, August 1998 [32]. A journal version can be found in [33].

11.1 A Strategy for Constructing 3D Orthogonal Drawings

In this section we describe a novel approach to three-dimensional orthogonal graph drawing, that can be considered more as a general strategy rather than as a specific algorithm. In fact, an algorithm actually adopting this approach is presented in Chapter 12. In the framework described hereunder a three-dimensional orthogonal drawing algorithm is generated through four algorithmic steps, corresponding to four properties successively gained by the drawing.
The recognition of such intermediate solutions, offers a promising perspective in a research field that, although rich of interesting and deep results, suffers, in our opinion, from a lack of general methodologies.

In this chapter we will refer exclusively to three-dimensional drawings, and will omit the "three-dimensional" adjective from the definitions. Further, to simplify the notation, when this does not cause ambiguities, we shall consider an orthogonal drawing as a graph with coordinate values for its vertices and bends.

We also make use of more unusual definitions that describe intermediate products of our design process. Such definitions allow us to describe "degenerate" drawings where vertices can overlap, edges can intersect, and/or can have length 0.

A 01-drawing is an orthogonal grid drawing such that each edge has either length 0 or length 1 and vertices may overlap. Observe that a 01-drawing does not have bends along the edges.

A 0-drawing is a (very!) trivial 01-drawing such that each edge has length 0 and all vertices have the same coordinates. A 1-drawing is a 01-drawing such that all edges have length 1 and vertices have distinct coordinates. (See Fig. 11.1.) Observe that while all graphs have a 01-drawing, only some admit a 1-drawing. For example the triangle graph does not admit a 1-drawing.

![Figure 11.1: A 1-drawing of a graph with ten vertices.](image)

Let $G$ be a graph. A subdivision $G_1$ of $G$ is a graph obtained from $G$ by replacing some edges of $G$ with simple paths. We partition the vertices of $G_1$ into vertices that belong also to $G$ (we call them original vertices) and vertices that belong only to $G_1$ (we call them dummy vertices). Observe that a subdivision $G_2$ of $G_1$ is a subdivision of $G$. If $G$ does not have vertices with degree greater than 6, because of the drawing algorithms mentioned in the introduction, there always exists a subdivision of $G$ that admits a 1-drawing.
11.1. A STRATEGY FOR CONSTRUCTING 3D ORTHOGONAL DRAWINGS

From now on, unless otherwise specified, we deal only with graphs whose maximum degree is at most 6.

A dummy path of $G_1$ is a path consisting only of dummy vertices except, possibly, at the endpoints (that can be original vertices). A planar path of an orthogonal drawing is a maximal path whose vertices are on the same plane. A planar dummy path is self-intersecting if it has two distinct vertices with the same coordinates. We consider only paths with at least one edge.

Our method constructs orthogonal grid drawings with all vertices at distinct coordinates and without intersections between edges (except at the common endpoints). The drawing process consists of a sequence of steps. Each step maps a 01-drawing of a graph $G$ into a 01-drawing of a subdivision of $G$. We start with a 0-drawing of $G$ and, at the last step, we get a 1-drawing of a subdivision $G_1$ of $G$. Hence, an orthogonal grid drawing of $G$ is obtained by replacing each path of $G_1$, corresponding to an edge $\{u, v\}$ of $G$, with an orthogonal polygonal line connecting $u$ and $v$.

The general strategy is as follows. Let $G$ be a graph. We consider several subsequent subdivisions of $G$. We construct an orthogonal grid drawing $\Gamma$ of $G$ in four phases.

**Vertex Scattering:** Construct a scattered representation $\Gamma_1$ of $G$, i.e. a 01-drawing such that:

- $\Gamma_1$ is a subdivision of $G$,
- all the original vertices have different coordinates, and
- all the planar dummy paths are not self-intersecting.

After this phase dummy vertices may still overlap both with dummy and with original vertices.

**Direction Distribution:** Construct a direction-consistent representation $\Gamma_2$ of $G$, i.e. a 01-drawing such that:

- $\Gamma_2$ is a scattered representation of $G$,
- for each vertex $v$ of $\Gamma_2$, $v$ and all its adjacent vertices have different coordinates.

We call this phase Direction Distribution because after this phase the edges incident on $v$ "leave" $v$ with different directions. Observe that this is true both in the case $v$ is original and in the case $v$ is dummy.

**Vertex-Edge Overlap Removal:** Construct a vertex-edge-consistent representation $\Gamma_3$ of $G$, i.e. a 01-drawing such that:
• $\Gamma_3$ is a direction-consistent representation of $G$,
• for each original vertex $v$, no dummy vertex has the same coordinates of $v$.

After this step the original vertices do not “collide” with dummy vertices. Observe that groups of dummy vertices sharing the same coordinates may still exist.

**Crossing Removal:** Construct a 1-drawing $\Gamma_4$ that is a vertex-edge-consistent representation of $G$.

Observe that, since $\Gamma_4$ is a 1-drawing, all its vertices, both original and dummy, have different coordinates. Also, observe that an orthogonal grid drawing $\Gamma$ of $G$ is easily obtained from $\Gamma_4$.

Each of the above phases is performed by repeatedly applying the same simple primitive operation called *split*. Informally, this operation “cuts” the entire graph with a plane perpendicular to one of the axes. The vertices lying in the “cutting” plane are partitioned into two subsets that are “pushed” into two adjacent planes. A more formal definition follows.

In what follows, the term *direction* always refers to a direction that is isothetic with respect to one of the axes and the term *plane* always refers to a plane perpendicular to one of the axes. Given a direction $d$ we denote by $-d$ its opposite direction.

Let $\Gamma$ be a 01-drawing. A *split operation* has 4 parameters $d, P, \phi, \rho$, where:

• $d$ is a direction.
• $P$ is a plane perpendicular to $d$.
• Function $\phi$ maps each vertex of $\Gamma$ laying in $P$ to a boolean.
• Function $\rho$ maps each edge $\{u, v\}$ of $\Gamma$ laying in $P$ such that $\phi(u) \neq \phi(v)$ and such that $u$ and $v$ have different coordinates to a boolean.

Operation *split $(d, P, \phi, \rho)$*, applied to $\Gamma$, performs as follows (see Fig. 11.2).

1. Move one unit in the $d$ direction all vertices in the open half-space determined by $P$ and $d$. Such vertices are “pushed” towards $d$.
2. Move one unit in the $d$ direction each vertex $u$ on $P$ with $\phi(u) = true$.
3. For each edge $\{u, v\}$ that after the above steps has length greater than one, replace $\{u, v\}$ with the new edges $\{u, w\}$ and $\{w, v\}$, where $w$ is a new dummy vertex. Vertex $w$ is positioned as follows.
(a) If the function $\rho(u,v)$ is not defined, then vertex $w$ is simply put in the middle point of the segment $u,v$.

(b) Else, (the function $\rho(u,v)$ is defined) suppose, without loss of generality, that $\phi(u) = true$ and $\phi(v) = false$. Two cases are possible. If $\rho(u,v) = true$, then $w$ is put at distance 1 in the $d$ direction from $v$. If $\rho(u,v) = false$, then $w$ is put at distance 1 in the $-d$ direction from $u$. Roughly speaking, the function $\rho$ is used to specify which is the orientation of the “elbow” connecting $u$ and $v$.

![Diagram](image)

Figure 11.2: An example of split: (a) before the split and (b) after the split. Vertices with $\phi = true$ ($\phi = false$) are black (light grey). Edges with $\rho = true$ ($\rho = false$) are labelled t (f). The little cubes are dummy vertices inserted by the *split*.

Observe that a *split* operation applied to a 01-drawing of a graph $G$ constructs a 01-drawing of a subdivision of $G$. Also, although *split* is a simple primitive, it has several degrees of freedom, expressed by the split parameters, whose usage can lead to very different drawing algorithms. Further, *split* has to be “handled with care”, since by applying a “random” sequence of *split* operations there is no guarantee that the process terminates with a 1-drawing.

### 11.2 Feasibility of the Approach

In this section we show how the *split* operation can be used to perform the four drawing phases described in Section 11.1.
Since the definition of a scattered representation requires that all the planar dummy paths are not self-intersecting and since an edge of zero length implies the existence of a self-intersecting path, we have:

**Property 8** All the edges of a scattered representation have length 1.

We now prove that a scattered representation can always be constructed.

**Theorem 8** Let $\Gamma_0$ be a 0-drawing of a graph $G$. There exists a finite sequence of split operations that, starting from $\Gamma_0$, constructs a scattered representation of $G$.

**Proof:** Consider a sequence of split operations all performed with planes perpendicular to the same axis, say the $x$-axis, and such that each split separates one original vertex from the others.

Namely, suppose that the $n$ vertices of $\Gamma_0$ are labelled $v_1, \ldots, v_n$ and that all of them are positioned at the origin. For each $i$ such that $1 \leq i \leq n - 1$ we perform $\text{split}(d, P, \phi_i, \rho)$ where:

- $d$ is the direction of the $x$-axis.
- $P$ is the plane $x = 0$.
- Function $\phi_i$ maps vertex $v_i$ to true and all the other vertices on $P$ to false.
- Function $\rho$ is not defined on any edge.

At the end of the process all vertices lie on the same line and original vertices have different coordinates. Furthermore, all the obtained dummy paths consist only of straight line segments with length 1 and with the same direction. Hence, all dummy paths are not self-intersecting.

Let $u$ be a vertex. We call the six directions around $u$ access directions of $u$. Consider the access direction of $u$ determined by traversing edge $\{u, v\}$ from $u$ to $v$; this is the access direction of $u$ used by $\{u, v\}$. An access direction of $u$ that is not used by any of its incident edges is a free direction of $u$.

Given a direction $d$ and a vertex $v$, we denote by $P_{d,v}$ the plane through $v$ and perpendicular to $d$. The following theorem shows that, starting from a scattered representation, we can always construct a direction-consistent representation.

**Theorem 9** Let $\Gamma_1$ be a scattered representation of a graph $G$. There exists a finite sequence of split operations that, starting from $\Gamma_1$, constructs a direction-consistent representation of $G$. 
11.2. FEASIBILITY OF THE APPROACH

**Proof:** We consider one by one each vertex $u$ of $\Gamma_1$ with edges $\{u, v\}$ and $\{u, w\}$ that use the same access direction $d$ of $u$. Since $\Gamma_1$ is a scattered representation of $G$ we have that:

- $u$ is an original vertex, and
- at least one of $v$ and $w$ (say $v$) is dummy.

Also, by Property 8 we have that no edge incident to $u$ has length 0, and hence use a direction of $u$.

Two cases are possible. Case 1: at least one free direction $d'$ of $u$ is orthogonal to $d$; see Fig. 11.3.a. Case 2: direction $-d$ is the only free direction of $u$; see Fig. 11.4.a.

![Figure 11.3](image1.png)

**Figure 11.3:** Case 1 in the proof of Theorem 9, before (a) and after (b) the split operation.

![Figure 11.4](image2.png)

**Figure 11.4:** Case 2 in the proof of Theorem 9, before (a) and after (b) the split operation.
Case 1: We perform $\text{split}(d', P_{\phi, \rho})$ as follows. We set $\phi(v) = \text{true}$, all the other vertices of $P_{\phi, \rho}$ have $\phi = \text{false}$. Also, $\rho(u, v) = \text{true}$, all the other edges in the domain of $\rho$ have $\rho = \text{false}$.

After performing the $\text{split}$ (see Figure 11.3.b), the first edge resulting from the subdivision of $\{u, v\}$ uses the direction $d'$ of $u$. The usage of the other access directions of $u$ is unchanged. Also, all the other vertices still use the same access directions as before the split with the exception, possibly, of $v$ (that is dummy).

Case 2: Let $d''$ be a non-free direction of $u$ different from $d$. We perform the same split operation as the one of Case 1, using direction $d''$ instead of $d$. After the $\text{split}$, (see Figure 11.4.b), the first edge resulting from the subdivision of $\{u, v\}$ uses the direction $d''$ of $u$. At this point, since at least one direction of the free directions of $u$ is orthogonal to $d''$ (direction $-d$), we can apply the same strategy of Case 1.

Finally, it is easy to observe that the above split operations preserve the properties of the scattered representations.

In the following we show that, starting from a direction-consistent representation, a vertex-edge-consistent representation can be obtained.

We define a simpler version of $\text{split}(d, P, \phi, \rho)$, called $\text{trivialsplit}(d, P)$, where $\phi$ is $\text{false}$ for all vertices of the cutting plane, and, as a consequence, the domain of $\rho$ is empty. Roughly speaking, $\text{trivialsplit}$ has the effect of inserting a new plane in the drawing that contains only the dummy vertices that are caused by the edge “stretches”. We use $\text{trivialsplit}$ for tackling the cases where a dummy vertex has the same coordinates of another vertex. The following property follows from the definition.

Property 9: Operation $\text{trivialsplit}$ does not affect the usage of the access directions of the vertices.

Theorem 10: Let $\Gamma_2$ be a direction-consistent representation of a graph $G$. There exists a finite sequence of split operations that, starting from $\Gamma_2$, constructs a vertex-edge-consistent representation of $G$.

Proof: Consider one by one each original vertex $u$ of $\Gamma_2$ such that there exists a dummy vertex $v$ with the same coordinates of $u$. Let $\{v', v\}$ and $\{v, v''\}$ be the incident edges of $v$. By Property 8 and by the fact that $\Gamma_2$ is a scattered representation of $G$, it follows that $v, v'$ and $v''$ have different coordinates.

Let $d'$ and $d''$ be the directions of $v$ used by $\{v', v\}$ and by $\{v, v''\}$, respectively (see Fig. 11.5.a and Fig. 11.6.a). We perform $\text{trivialsplit}(d', P_{d', v})$ and $\text{trivialsplit}(d'', P_{d'', v})$. After performing such operations vertex $v$ is guaranteed to be adjacent to dummy vertices $u'$ and $u''$ created by the performed splits.
11.2. FEASIBILITY OF THE APPROACH

Two cases are possible (see Fig. 11.5.b and Fig. 11.6.b): either \( d' = -d'' \) or not. In the first case we define \( d''' \) as any direction orthogonal to \( d' \); in the second case we define \( d''' \) as any direction among \( d', d'' \), and the two directions orthogonal to \( d' \) and \( d'' \). At this point we perform a third split. Namely, we apply \( \text{split}(d''', P_{\phi,v}, \phi, \rho) \) as follows (see Fig. 11.5.c and Fig. 11.6.c).

- \( \phi(v) = \text{true} \), all the other vertices of \( P_{\phi,v} \) have \( \phi = \text{false} \).
- All the edges in the domain of \( \rho \) have \( \rho = \text{true} \).

Note that at this point \( u \) and \( v \) have different coordinates. Further, by Property 9 and because of the structure we have chosen for \( \text{split} \) we have that the entire sequence of operations preserves the properties of the direction-consistent representations, and does not generate new vertex-edge overlaps.

In the remaining part of this section, we study how to perform the last phase of the general strategy presented in Section 11.1. Namely, we are going to show that, given a vertex-edge-consistent representation of a graph \( G \), it is possible to construct a new vertex-edge-consistent representation of \( G \) that
is a 1-drawing. Before introducing the corresponding theorem we need some intermediate terminology and results.

Let $\Gamma$ be a 01-drawing of $G$. We say that two distinct vertex-disjoint planar paths $p'$ and $p''$ of $\Gamma$ on the same plane intersect if there exist two vertices one of $p'$ and the other of $p''$ with the same coordinates. We denote by $\chi(\Gamma)$ the number of the pairs of intersecting planar dummy paths of $\Gamma$. Observe that $\chi(\Gamma)$ can be greater than one even if there are just two vertices with the same coordinates (see Fig. 11.7).

Figure 11.7: Two dummy vertices with the same coordinates originating 3 pairs of intersecting planar dummy paths.

Suppose we need to perform an operation trivialsplit($d, P$) on $\Gamma$ and let $\Gamma'$ be the obtained 01-drawing. Let $P'$ be the plane of $\Gamma$ parallel to $P$ and at distance 1 from $P$ in the $d$ direction.

**Property 10** Plane $P'$ does not contain any edge of $\Gamma'$.

Of course this implies that $P'$ does not contain any planar path. Also, because of Property 10, we have:

**Property 11** The planar dummy paths of $\Gamma'$ are in one-to-one correspondence with the planar dummy paths of $\Gamma$.

**Property 12** $\chi(\Gamma) = \chi(\Gamma')$.

**Proof:** This follows from Property 11 and from the fact that two planar dummy paths of $\Gamma$ intersect if and only if the corresponding planar dummy paths of $\Gamma'$ intersect.
11.2. FEASIBILITY OF THE APPROACH

**Theorem 11** Let $\Gamma_3$ be a vertex-edge-consistent representation of a graph $G$. There exists a finite sequence of split operations that, starting from $\Gamma_3$, constructs a 1-drawing of a subdivision of $G$.

**Proof:** Since $\Gamma_3$ is vertex-edge-consistent, all original vertices have distinct coordinates, but some dummy vertices may still overlap.

If $\chi(\Gamma_3) = 0$, then $\Gamma_3$ is already a 1-drawing of $G$. Otherwise, we repeatedly select a pair of intersecting planar dummy paths $p'$ and $p''$ (see Fig. 11.8.a) and “remove” their intersection, decreasing the value of $\chi$. Such removal is performed as follows.

Let $u$ and $v$ be the end-vertices of $p'$. We have three cases:

![Diagram](image)

Figure 11.8: Intersecting planar dummy paths in the proof of Theorem 11. The black vertex is original. All the other vertices are dummy. The little cubes are dummy vertices inserted by the split operations described in the proof. Slanted edges indicate the crossing.

1. exactly one of $u$ and $v$ (say $v$) is an original vertex,

2. both $u$ and $v$ are original vertices, or
3. both $u$ and $v$ are dummy vertices.

In Case 1 (see Figs. 11.8.a and 11.8.b) we perform $trivialsplit(d, P_{d,v})$ where $d$ is the direction $p'$ leaves $v$. In Case 2 we perform $trivialsplit(d', P_{d',u})$ and $trivialsplit(d', P_{d',u})$ where $d'$ (or $d''$) is the direction $p'$ leaves $u$ ($v$). In Case 3 we do not perform any $trivialsplit$.

After the above splits, by Property 12, the value of $\chi$ stays unchanged. Also, observe that the drawing is still a vertex-edge-consistent representation of $G$. At this point we concentrate on Case 1, the other cases are similar and are omitted for brevity. We denote by $s$ the dummy vertex introduced along $p'$ by $trivialsplit (d, P_{d,v})$.

Let $d''$ be a direction orthogonal to the plane $P$ where $p'$ and $p''$ intersect. We perform $split (d'', P, \phi, \rho)$, by setting (see Fig. 11.8.c):

- $\phi(x) = true$ for each vertex $x \in p'$ and $x \neq v, s$ (false otherwise) and
- $\rho = true$ for all the edges in the domain of $\rho$.

We have that $\chi$ decreases after the split by at least one unit. It is easy to see that such a split preserves the properties of the vertex-edge-consistent representations.

In this section we have shown that $split$ is a powerful tool in performing the phases of the strategy presented in Section 11.1. Namely, each of Vertex Scattering, Direction Distribution, Vertex-edge Overlap Removal, and Crossing Removal can be performed by a finite sequence of splits.
Chapter 12

The Reduce-Forks Algorithm

We present an algorithm called Reduce-Forks and using the Split&Push framework for three-dimensional orthogonal graph drawing described in the previous chapter. The phases of the strategy are refined into several heuristics that are illustrated in the corresponding sections. In Chapter 13 the Reduce-Forks algorithm will be compared with most of the drawing methods taken from the literature. The work described in this chapter appeared in [32, 33].

12.1 Vertex Scattering

An edge \( \{u, v\} \) is cut by \( \text{split}(d, P, \phi, \rho) \) if \( u \) and \( v \) have different values of \( \phi \). Informally, they were in plane \( P \) before the split and are in different planes after the split. A pair of adjacent edges that are cut by a split is a fork.

Roughly speaking, the heuristic of Reduce-Forks for Vertex Scattering works as follows. We select an arbitrary pair of original vertices \( u \) and \( v \) of \( G \) with the same coordinates. Let \( P', P'', \text{ and } P''' \) be the three planes containing \( u \) and \( v \). We consider the set of split operations with planes \( P', P'', \text{ and } P''' \) and that separate \( u \) from \( v \) and perform one with “a few” forks. We choose to keep small the number of forks because each fork will require the insertion of at least one dummy vertex in the subsequent Direction Distribution phase. Such dummy vertices will become bends in the final drawing. We repeatedly apply the above strategy until a scattered representation is obtained.

More formally, observe that finding a split with no forks is equivalent to finding a matching cut. A matching cut in a graph is a subset of edges that are pairwise vertex disjoint (matching) and such that their removal makes the graph disconnected (cut). Unfortunately, the problem of finding a matching cut in a graph is NP-complete (see [113]). The proof in [113] is based on a reduction from the NAE3SAT problem [59] and holds for graphs of arbitrary degree.
However, a simple heuristic for finding a cut with a few forks is described below.

Consider vertices $u$ and $v$. We color black and red the vertices in the two sides of the split. Each step of the heuristic colors one vertex. At a certain step a vertex can be black, red or free (uncolored). At the beginning $u$ is black, $v$ is red, and all the other vertices are free.

Colored vertices adjacent to free vertices are active vertices. Black (Red) vertices adjacent to red (black) vertices are boundary vertices. See Fig. 12.1. Each step works as follows.

1. If a boundary active red vertex, say $x$, exists, then color red one free vertex $y$ adjacent to $x$. The rationale for this choice is the following: since vertex $x$ is adjacent to at least one black vertex $w$ (from the definition of boundary vertex), by coloring $y$ red we prevent a fork between $\{x,y\}$ and $\{x,w\}$. Analogously, if a boundary active black vertex exists, then color black one of its adjacent free vertices.

2. Else, if an active red vertex, say $x$, exists, then choose a free vertex $y$ adjacent to $x$ and color $y$ red. This is done to avoid cutting edge $\{x,y\}$. Analogously, if an active black vertex exists, then color black one of its adjacent free vertices.

Figure 12.1: Red, black, and free vertices in the Vertex Scattering heuristic of Algorithm Reduce-Forks.
3. Else, randomly color black or red a random free vertex.

We perform the above heuristic to each of the subgraphs induced by the vertices on $P'$, $P''$, and $P'''$. Then we select a split with the plane among $P'$, $P''$, and $P'''$ where the cut with the smallest number of forks has been found.

The heuristic can be implemented to run time and space linear in the size of the current 01-drawing (a graph of maximum degree six has a linear number of edges).

Observe that, since each split gives different coordinates to at least two original vertices formerly having the same coordinates, in the worst case the heuristic is used a number times that is linear in the number of original vertices.

Fig. 12.4 shows the sequence of splits performed by the heuristic on the $K_6$ graph.

12.2 Direction Distribution

Now, for each original vertex $u$ of $G$ with edges $\{u, v\}$ and $\{u, w\}$ such that $v$ and $w$ have the same coordinates (at least one of $v$ and $w$ is dummy), we have to find a split that separates $v$ from $w$ (see Fig. 12.2a). Of course there are many degrees of freedom for choosing the split. In Reduce-Forks a heuristic is adopted that follows the approach of the proof of Theorem 9. However, in performing the splits we try to move an entire planar dummy path rather than moving a single dummy vertex. This has the effect of both decreasing the number of bends (dummy vertices with orthogonal incident edges) introduced by the split, and of occasionally solving an analogous problem on the other extreme of the planar dummy path.

More formally, we apply the following algorithm.

1. Compute the (two) planar dummy paths $p_v$ and $q_v$ containing $\{u, v\}$ (see Figs. 12.2b–12.2c) and the (two) planar dummy paths $p_w$ and $q_w$ containing $\{u, w\}$ (see Figs. 12.2d–12.2e).

2. For each path of $p_v$, $q_v$, $p_w$, and $q_w$ determine the split operations that separate the path (except for the possible original vertices) from all the other vertices that lie on its plane. For each path we have exactly two possible splits. Fig. 12.3 shows the effect of two possible splits on the configuration of Fig. 12.2a.

3. Weight the eight split operations obtained in the previous step according to the number $n_d$ of vertices that become direction-consistent after the split and, secondarily, to the number of bends they introduce. Observe that $1 \leq n_d \leq 2$. In the example of Fig. 12.3 the split described by Fig. 12.3.b is preferred to the split described in Fig. 12.3a.
Figure 12.2: An example illustrating the heuristic adopted by the Reduce-Forks algorithm for the Direction Distribution phase. Black vertices are original. All other vertices are dummy. In (a) two vertices (v, and w) adjacent to the original vertex u, share the same coordinates. Paths $p_v$, $q_v$, $p_w$, and $q_w$ are shown with dark grey in (b), (c), (d), and (e), respectively.
4. Select and apply the split operation with minimum weight.

Observe that, since each original vertex requires at most six splits, this phase is performed with a number of splits that is, in the worst case, linear in the number of original vertices.

\section{12.3 Vertex-Edge Overlap Removal and Crossing Removal}

To perform the Vertex-Edge Overlap Removal and the Crossing Removal phases a technique is used similar to the one applied for the Direction Distribution phase. Namely, we identify a set of splits that can “do the job”. We weight such splits and then apply the ones with minimum weights.

For each original vertex \( u \) of \( G \) such that \( v \) has the same coordinates as \( u \):

1. Compute the (at most three) planar dummy paths containing \( v \).

2. For each path computed in the previous step, determine the split operations that separate the path (except for the possible original vertices) from all the vertices that lie on its plane. For each path we have exactly two possible splits.
3. Weight the split operations obtained in the previous step according to the number of bends and/or crossings they introduce.

4. Select and apply the split operation with minimum weight.

For each pair of dummy vertices $u$ and $v$ having the same coordinates:

1. Compute all the planar dummy paths containing $u$ or $v$.
2. Determine all the split operations that separate such paths and $u$ from $v$.
3. Weight such splits according to the number of bends they introduce.
4. Select and apply the split with minimum weight.

The presented techniques are easily extensible to obtain drawings of graphs of arbitrary degree with the following strategy:

- The Vertex Scattering step remains unchanged.
- In the Direction Distribution step for vertices of degree greater than six, we first “saturate” the six available directions and then we evenly distribute the remaining edges.
- The Vertex-edge Overlap Removal step remains unchanged.
- In the Crossing Removal step we distinguish between crossings that are “needed” because of the overlay between edges that is unavoidable because of the high degree and the crossings that can be removed. For the latter type of crossings we apply the techniques presented in Section 11.2, while the first type of crossings are handled in a post-processing phase, where vertices are suitably expanded.

Figs. 12.4 and 12.5 show how Reduce-Forks computes a drawing of a $K_6$ graph. Spheres represent original vertices while cubes represent dummy vertices. Vertices with the same coordinates are drawn inside the same box.
Figure 12.4: The Vertex Scattering phase of algorithm Reduce-Forks applied on a $K_6$ graph. (a) is a 0-drawing and (f) is a scattered representation.
Figure 12.5: (a–c) the Direction Distribution phase of algorithm Reduce Forks applied on the scattered representation of the $K_6$ graph of Fig. 12.4.f (duplicated in (a) for the convenience of the reader). (d) final drawing. Observe that in this example the Vertex-edge Overlap Removal and the Crossing Removal phases are not necessary since (c) is already a 1-drawing.
Chapter 13

Experimental Comparison of Three-Dimensional Orthogonal Drawing Methods

In this chapter we describe the results of an experiment concerning most of the three-dimensional orthogonal drawing methods found in literature. Also, we validate the Split&Push approach introduced in Chapter 11 by comparing the performance of algorithm Reduce Forks described in Chapter 12 with the other algorithms. The comparison shows that no algorithm tested can meet both the requirements of minimum area and reduced number of bends, confirming the existence of a tradeoff between the two measures, a concept that was suggested by several researchers, but still waits to be rigorously proven. Unfortunately, the drawings produced by the Split&Push algorithm, extremely readable when the graph is small, seem to loose their good qualities when the size of the input graph increases. Thus, current orthogonal three-dimensional drawing methods have to be considered still unusable for representing large graphs. They don’t reach the threshold of usability. Part of the work described in this chapter appeared in [32, 33].

13.1 Algorithms Description

While several experimental comparisons and extensive empirical analysis can be found in the literature for two-dimensional drawing methods [14, 31, 132], three-dimensional drawing algorithms have not been the object of the same attention. This is particularly unfortunate since both for its theoretical appeal and for the high number of potential applications, research in 3D graph drawing is flourishing. Low-price high-performance 3D graphic workstations are becoming widely available and the demand of visualization of large graphs
increases with the popularity of the graph drawing methods and tools, and 3D graph drawing seems to offer interesting perspectives to such a demand.

In what follows we will focus on three-dimensional orthogonal drawing algorithms.

Biedl [10] shows a linear time algorithm (in what follows we call it Slices) that draws a graph in $O(n^2)$ volume with at most 14 bends per edge. The drawing is obtained by placing all the vertices on a certain horizontal plane and by assigning a further horizontal plane to every edge, “one slice per edge”.

Eades, Stirk, and Whitesides [44] propose a $O(n^{3/2})$-time algorithm, based on augmenting the graph to an Eulerian graph and on applying a variation of an algorithm by Kolmogorov and Barzdn [83]. The algorithm produces drawings that have $O(n^{3/2})$ volume and at most 16 bends per edge. We call this algorithm Kolmogorov.

The algorithm proposed by Eades, Symvonis, and Whitesides in [45] (we call it Compact) requires $O(n^{3/2})$ time and volume and introduces at most 7 bends per edge.

In the same paper [45] Eades, Symvonis, and Whitesides presented a second algorithm (we call it Three-Bends) whose complexity is linear, while its volume is $27n^3$, and at most 3 bends per edge are introduced. Algorithm Three-Bends is based on augmenting the graph to a 6-regular graph and on a coloring technique. The implementation used in the experimental comparison follows the description of the algorithm given in the journal version [46], in which, by using the result in [124] for the coloring phase, the time complexity of algorithm Three-Bends is $O(n)^1$.

Papakostas and Tollis [109] present a linear time algorithm (we call it Interactive) that requires at most 4.66$m^3$ volume and at most 3 bends per edge. It is incremental and can be extended to draw graphs with vertices of arbitrary degree. The construction starts from a first pair of adjacent vertices, and then it adds one vertex at a time with its incident edges.

Wood [140] presents an algorithm for maximum degree 5 graphs that requires $O(n^3)$ volume and at most 2 bends per edge. Recently [139], the result has been extended to maximum degree 6 graphs using no more than 4 bends per edge. The volume is at most $2.37n^3$, the total number of bends is always less than $7m/3$, where $m$ is the number of edges.

Closson, Gartshore, Johansen, and Wismath present an algorithm that uses $O(\sqrt{n}) \times O(\sqrt{n}) \times O(n)$ space, and introduces a maximum of 6 bends per edge, working in linear time [21]. We call such algorithm Dynamic for its particular characteristic of allowing both insertions and deletions of vertices and edges in constant time.

Finally, Eades, Symvonis, and Whitesides in the journal version [46] of [45]

---

1In the paper [45] the coloring phase is assumed to run in $O(n^{3/2})$ time.
presented several algorithms in an effort to explore the trade-offs between volume occupation and number of bends. In particular, the authors present three linear algorithms that draw degree six graph in \( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) \), \( O(\sqrt{n}) \times O(n) \times O(n) \), and \( O(n) \times O(n) \times O(n) \) volume, introducing a maximum of 6, 5, and 4 bends per edge, respectively. We call them Six-Bends, Five-Bends, and Four-Bends.

The publication of several of the three-dimensional orthogonal algorithms mentioned above, constituted such a breakthrough in the search for algorithms able to meet asymptotic bounds, that they were published without any experiment (and sometimes without any implementation at all). All the more interesting is the assessment that some of them have a practical behavior far better than what guaranteed by the authors. However, few of them seem to be candidate algorithms for visualization purposes. In Fig. 13.1 the theoretical bounds for the algorithms tested in the experimental comparison are shown.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ref.</th>
<th>Time compl.</th>
<th>Max bends</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact</td>
<td>[45, 46]</td>
<td>( O(n^{3/2}) )</td>
<td>7</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Dynamic</td>
<td>[21]</td>
<td>( O(n) )</td>
<td>6</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Interactive</td>
<td>[110]</td>
<td>( O(n) )</td>
<td>3</td>
<td>( O(n) \times O(n) \times O(n) )</td>
</tr>
<tr>
<td>Five-Bends</td>
<td>[46]</td>
<td>( O(n) )</td>
<td>5</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Kolmogorov</td>
<td>[44]</td>
<td>( O(n^{3/2}) )</td>
<td>16</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Reduce-Forks</td>
<td>[33]</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Six-Bends</td>
<td>[46]</td>
<td>( O(n) )</td>
<td>6</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Slices</td>
<td>[10]</td>
<td>( O(n) )</td>
<td>14</td>
<td>( O(\sqrt{n}) \times O(\sqrt{n}) \times O(n) )</td>
</tr>
<tr>
<td>Three-Bends</td>
<td>[45, 46]</td>
<td>( O(n) )</td>
<td>3</td>
<td>( O(n) \times O(n) \times O(n) )</td>
</tr>
</tbody>
</table>

Figure 13.1: Theoretical bounds for the algorithms tested in the experimental comparison. No theoretical bounds could be found for algorithm Reduce-Forks.

We included between the tested algorithms the algorithm Slices and Kolmogorov for historic reasons. In fact, the Slices algorithm by Biedl [10] was the first, to our knowledge, to propose a three-dimensional layout in which all the vertices lay on the same plane, while the edges use the space above and below the such plane for routing purposes. This strategy was adopted by most of the following algorithms, and between them the algorithm Kolmogorov, was the first to meet the optimal bound on the volume.

To explore the volume-bends trade-off we implemented and tested the Compact Six-Bends, Five Bends, and Three-Bends (the Four-Bends algorithm has an asymptotic behavior that is worse than Three-Bends one described in the same paper).

Algorithm Interactive has the same asymptotic volume occupation as Three-Bends, but smaller constants, while the maximum number of bends
introduced on an edge is 3 for both algorithms. Algorithm Dynamic has the
same asymptotic volume occupation and maximum number of bends per edge
as Six-Bends.

In conclusion, we have implemented and performed the experimental com-
parison of algorithms Compact [45, 46], Dynamic [21], Interactive [110],
Five-Bends [46], Kolmogorov [44], Reduce-Forks [33], Six-Bends [46], Sli-
ces [10], and Three-Bends [45, 46] with a large test suite of graphs. The ex-
periments have been performed on a Linux platform by using the 3DCube [114]
system. All the algorithms have been implemented in C++.  

Unfortunately, we didn’t had the time to implement the algorithm in [139]
and to include it in the experiment.

13.2 Graph Base Generation

The first problem arising in a graph drawing experiment is the adoption of
a suitable graph base. Since the 3D graph drawing field, for its novelty, still
lacks well established real-life benchmark suites, we were forced to test the
algorithms on randomized graphs instead of real-life examples, and since no
graph base was available for three-dimensional orthogonal graph drawing we
underwent the process of producing one.

The test suite described hereunder is available at http://www.dia.uni-
roma3.it/~patrigna/3dcube/test_suite.html. It was first used for the exper-
iment presented in the paper [33], and used by the authors of [110, 90] to
discuss the practical behavior of their algorithms.

The test suite is composed of 1900 randomly generated graphs having from
6 to 100 vertices, 20 graphs for each value of vertex cardinality. All graphs
are connected, with maximum vertex degree 6, without multi-edges and self-
loops. The density is chosen to be in the middle of the allowed interval: the
number of edges is twice the number of vertices. Note that in a connected
graph of maximum degree 6, the density can range from 1 to 3. Also, as put
in evidence in [31], in the practical applications of graph drawing it is unusual
to have graphs with density greater than 2.

The randomization procedure was very simple. For each graph the num-
ber of vertices and edges was set before the randomization. Edge insertions
were performed on distinct randomized vertices, provided their degree was
less than 6 and an edge between the two vertices did not already exist. Af-
ter all the edges had been inserted, the graph was tested for connectedness.
Non connected graphs had all their edges removed and reinserted from scratch
with the same randomized process described above, till a connected graph was
produced.
13.3 Experimental Comparison

We performed the experimental comparison by running each of the tested algorithms on each graph of the graph base. After the drawing was produced we launched a simple post-processing algorithm with the purpose to eliminate empty planes (planes not containing vertices nor bends).

We considered two families of quality measures. For the efficiency we relied on the time performance (CPU seconds); for the readability we measured the average number of bends along the edges, the average edge length, and the average volume of the minimum enclosing box with sides isothetic to the axes. Figs. 13.2, 13.3 13.4 and 13.5 illustrate the results of the comparison.

The comparison shows that no algorithm tested can be considered “the best”. Namely, some algorithms are more effective in the average number of bends (Interactive, Reduce-Forks, and Three-Bends) while other algorithms perform better with respect to the average volume (Compact and Slices) or to the edge length (Compact, Interactive, and Reduce-Forks). More precisely:

- The average number of bends (see Fig. 13.3) is comparable for Interactive, Reduce-Forks, and Three-Bends, since it remains for all of them under the value of 3 bends per edge, while it is more and more high for Five-Bends, Six-Bends, Compact, and Dynamic. The average number of bends is a little higher for Slices, and it is definitely much too high for Kolmogorov. Furthermore, Reduce-Forks performs better than the other algorithms for graphs with number of vertices in the range 5–30. Interactive performs better in the range 30–100. Another issue concerns the results of the experiments vs. the theoretical analysis. About Kolmogorov the literature shows an upper bound of 16 bends per edge [45] while our experiments obtain about 19 on average. This inconsistency might show a little “flaw” in the theoretical analysis. Further, about Compact the experiments show that the average case is much better than the worst case [45].

- Concerning the average edge length (see Fig. 13.4), Reduce-Forks seems to be one of the best performing. In fact is the one that produces shortest edges for graphs up to 50 vertices, whereas Compact is better from 50 to 100. Other algorithms show worse performance.

- The values of volume occupation (see Fig. 13.5) show that Compact, Six-Bends, and Slices have the best performance for graphs bigger than 30 vertices, while Reduce-Forks performs better for smaller graphs.

Some examples of the drawings constructed by the experimented algorithms are shown in Fig. 13.6.
CHAPTER 13. EXPERIMENTAL COMPARISON OF
THREE-DIMENSIONAL ORTHOGONAL DRAWING METHODS

For these considerations, we can say that Reduce-Forks is the most effective algorithm for graphs in the range 5-30. Also, among the algorithms that have the lowest number of bends along the edges (Interactive, Reduce-Forks, and Three-Bends), Reduce-Forks is the one that has the best behavior in terms of edge length and volume. This is obtained at the expense of an efficiency that is much worse than the other algorithms. However, the CPU time does not seem to be a critical issue for the size of graphs in this interval. In fact, even for Reduce-Forks, the CPU time never exceeds 100 seconds, which is still a reasonable time for most applications.

13.4 Practical Usability of Three-Dimensional Orthogonal Drawing Algorithms

The tests we performed confirm the widespread conviction in the three-dimensional orthogonal research area that a tradeoff between aesthetic criteria imposes that a drawing method that guarantees fewer bends per edge uses more space. Both the asymptotic bounds of the algorithms and their experimental behavior seem to confirm this conjecture.

Unfortunately, the fact that a drawing with a few bends and contained in a restricted area cannot be automatically produced discourages the adoption of three-dimensional orthogonal drawings for practical purposes. The user may experience a disorienting feeling of confusion even when presented with drawings of relatively small graphs within optimal bounds.

By and large, orthogonal three-dimensional drawings seem unable to exploit the potential of the three-dimensional environment. As for the Reduce- Fork algorithm presented in Chapter 11, although it yields surprisingly clear drawings of small graphs, it loses its effectiveness when the size of the input graph increases, and its poor efficiency contributes to discouraging its adoption for larger graphs.

As for Reduce-Fork algorithm, it produces extremely readable drawings when the size of the input graph is small (see Figures 12.3 and 13.6 for some examples), but, unfortunately, the drawings seem to lose their good qualities when the size of the input graph increases. Although graphs with vertices in the range 10-100 are crucial in several applications [31], the drawing algorithm fails to successfully exploit the three-dimensional environment to represent large graphs.

Thus, although 3D graph drawing seems to offer interesting perspectives for the visualization of large graphs, the experiments presented in this chapter show that the aesthetic quality of the drawings produced with the existing algorithms is still not sufficient to deal with large graphs (see, for example, Fig. 13.6). Also in this respect it would be important to improve the readabil-
13.4. PRACTICAL USABILITY OF THREE-DIMENSIONAL
ORTHOGONAL DRAWING ALGORITHMS

ity of the produced drawings, even at the expenses of a higher computation
time.

Such improvement could be reached exploring several alternatives:

- New, more sophisticated algorithms and heuristics (alternative to Reduce-
Forks) could be devised within the Split&Push paradigm described in
Chapter 11. Such heuristics might be based on modified versions of the
3D graph drawing algorithms listed in Section 13.1.

- The impact of bend-stretching (or possibly other post-processing tech-
niques) on the performance of the different algorithms may be consid-
ered. A preliminary study in this direction is the paper [90], in which
several refinement heuristics are described and tested. The main result
of [90] seems to be that the Reduce-Fork algorithm is a good candidate
for this kind of post-processing refinement, since the quality of its draw-
ing was significantly improved by the tested techniques. Unfortunately,
the computational time of the refinement techniques was not considered,
and so the practical usability of such techniques for very large data sets
remains open.
Figure 13.2: Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slices, Three-Bends, Five-Bends, Six-Bends, and Dynamic with respect to time performance (on a logarithmic scale).
Figure 13.3: Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slices, Three-Bends, Five-Bends, Six-Bends, and Dynamic with respect to average number of bends.
Figure 13.4: Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slices, Three-Bends, Five-Bends, Six-Bends, and Dynamic with respect to average edge length.
Figure 13.5: Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slices, Three-Bends, Five-Bends, Six-Bends, and Dynamic with respect to volume occupation.
Figure 13.6: Three-dimensional orthogonal drawings of a $K_7$ as yielded by Compact (a), Interactive (b), Kolmogorov (c), Reduce-Forks (d), Slices (e), and Three-Bends (f).
Part V

Information Hiding in Specific Application Contexts
Chapter 14

The Visualization of Network Partitions

Partitioning is often used to support better graph drawing. The approach described in this chapter is one occasion in which this relation is reversed and graph drawing is used to support better partitioning. In the effort to represent a very large partitioned graph the drawing method is strictly tuned on the domain-specific problem, relying on the additional information about the blocks the vertices belong to. Also, according to the hiding principle, most of the vertices are concealed, and, since the overlapping between vertices is used to occlude them, a very smooth and controlled hiding is allowed. A short description of the visualization aspects involved by the system can be found in [88].

14.1 The Network Partitioning Problem

A network or hypergraph \(G\) is a pair \((V,H)\), in which \(V\) is a set of vertices and \(H\) is a set of non-empty subsets of \(V\) called hyperedges.

Constrained \(k\)-way partitioning is the problem of partitioning the vertices of a network into \(k\) disjoint subsets (called blocks), in such a way to minimize the number of hyperedges spanning two or more blocks. The sizes of the blocks are allowed to vary around the value \(\frac{n}{k}\), where \(n = |V|\); a typical value for the allowed variance is 10%. In current state-of-the-art benchmark problems, \(|V|\) and \(|H|\) range from 10,000 to 200,000, and \(k\) is less than 10 [1].

Although the canonical partitioning application is VLSI design packaging, both at the logic and at the physical level, the same formulation may be found in different fields, as in parallel processing, where it is required to assign a large number of computations to a fixed number of processors.

The problem of constrained \(k\)-way network partitioning is NP-hard [59],

115
and a wide range of heuristics have been devised for it. The most renowned
heuristics are the so-called move-based algorithms, descending from the Kernighan-
Lin bisectioning algorithm introduced in [76] for graphs and extended to net-
works in [125]. Such algorithm improves an initial random partition by applying
an iterative procedure based on vertex-pair swaps.

Fiduccia-Mattheyses [52] and Sanchis [121] algorithms descend from the
Kernighan-Lin iterative improvement approach.

A collection of heuristics has been applied on this problem. A limited list
includes simulated annealing, tabu search, and genetic algorithms (see [2] for
a recent survey).

14.2 Interactive Partitioning

We used an innovative human-computer interaction paradigm called Human-
Guided Simple Search, or HuGSS for short, which was already successfully
applied to the problem of Capacitated Vehicle Routing with Time Windows [4].

Problems that are candidate to be approached with this paradigm are NP-
hard operation-research problems. Scheduling, routing, and layout tasks are
eamples with a broad application in industry. Typical heuristics for such
difficult combinatorial optimization problems combine some form of gradient
descent to find a local minimum with some strategy for escaping nonoptimal
local minima and exploring the search space. The HuGSS framework for
interactive search divides these two subtasks cleanly between human and com-
puter: the computer is responsible only for finding local minima using a simple
search method, while the human is responsible for steering the exploration of
the search space, escaping nonoptimal local minima and pursuing the good
opportunities offered by the current solution. HuGSS tries to combine the
fast processing speed of computers with human’s superior ability for strategic
assessment, by assigning each to its natural role.

With respect to previous approaches using human-computer interaction
the HuGSS paradigm advocates a stricter division of the roles between hu-
man and machine, and embodies a stronger notion of human guidance. Also,
HuGSS introduces stronger visualization requirements, since it is important
to convey to the user not only what is the current solution, but also if some
good opportunity is present and the direction in which the exploration of the
solution space has to be carried on.

The visualization challenge is even more difficult for the problem of network
partitioning, since the networks that must be handled are generally huge. The
ISPD98 benchmark [1] we used for our experimentation features hypergraphs
obtained from IBM circuits ranging from 10,000 to 200,000 vertices and as
many hyperedges.
14.3. VISUALIZING NETWORK PARTITIONS

The solutions we propose to such visualization problems are based on a combination of different patterns: we don’t tackle the general problem of visualizing a huge network, but we assume that additional information about the block of the partition each vertex belongs to is available. The two force directed visualizations proposed feature a large amount of overlappings between the vertices, producing several degrees of hiding that contributes to reducing the amount of information shown to the user.

14.3 Visualizing Network Partitions

Because it is difficult to unambiguously draw a hypergraph (the only works the author is aware of are [91] and [8], the latter admittedly ambiguous for hypergraphs with more than a few dozen hyperedges), we convert the hypergraph into a graph with weighted edges, by applying the so-called clique transformation, consisting of replacing each hyperedge of degree $n$ with a clique of $\frac{n(n-1)}{2}$ edges whose weights are $\frac{2}{n(n-1)}$.

The transformation yields high-density graphs: for the benchmark in [1], for example, even if the average degree of the hyperedges is less than four, the number of edges of the produced graph is one order of magnitude bigger than the number of the original hyperedges, and the average degree of vertices is as high as 20 or 40. Note that, due to the density of the graphs obtained, the techniques based on a spanning-tree reduction can not be applied. In fact, 90% of the edges would be omitted.

We designed our visualization to help the user make decisions about how to focus the search and to select search heuristics.

14.3.1 Choosing the Drawing Convention

Due to the large number of vertices, using exclusively the vertex color to indicate which block each vertex belongs to is not sufficient to yield a clear representation. Placing vertices of the same block near to each other, instead, has the double effect of stressing the belonging of the vertices to a common block through their position, and using the additional information offered by the partition to ease the layout process.

Since the average degree of the vertices is as high as 20 or 40, the kind of exploration that the user needs to perform on the graph is a high level view of blocks and groups of strongly connected vertices, rather than a detailed focused analysis on the neighborhood of a single vertex. It follows that, according to the remarks closing Chapter 4, the straight-line drawing convention (without bends, but prone to overlappings) offers a clearer representation than the orthogonal convention (no overlappings, but a number of bends introduced).
Since no strong graph theoretic property is guaranteed (the graphs are not planar, and nothing is known about their connectivity) a force directed approach seems a natural candidate for representing them. In fact, in such a framework two requirements can be easily accommodated:

- the weights of the edges can be accounted for by the elastic constant of the edge, and
- vertices of the same color can be kept near one to the other by connecting them together or to a common hub.

The main inconvenience with a force directed approach is that, if repulsive forces are considered, the visualization process will be slowed down by the introduced quadratic component. On the other hand, if repulsive forces are neglected, the vertices will easily overlap. The two visualization modes presented in this chapter find a narrow escape in this dilemma by neglecting repulsive forces while using vertex-overlaps as a way of hiding unnecessary information.

### 14.3.2 Induced Graph Visualization

The force-directed method we used to determine the position of each vertex in the first visualization mode is realized as follows. Each edge in the induced graph is replaced by a spring that opposes stretching or compressing with a force proportional to its weight. Furthermore, a spring is attached between each vertex and the hub of its block. The hubs themselves are fixed and positioned uniformly around a circle. For reasons of efficiency, no other force is considered, including repulsive forces between vertices.

A limited number of iterations (usually ten) are performed, starting from an initial placement in which the vertices are assigned the coordinates of their hubs.

In addition to vertex position, other visual methods are used to convey information about the partition. Membership in a block is indicated by vertex color. Edge weights are mapped onto intensity, so that edges of greater weight appear brighter. And at each hub an icon comprising a star-shaped polygon and circle indicates whether the block is near its minimum or maximum allowed size (the circle reaches the internal or external vertices of the polygon, respectively). The circle turns red if the size of the corresponding block is outside the boundaries of a feasible solution.

A drawing produced with this algorithm is shown in Figure 14.1. Since each vertex is attracted towards the hub of its block, (the strength of such force may be finely tuned by the user), vertices in the same block tend to overlap near their hub, and only the ones that are strongly linked to other blocks
stretch out and are easily distinguishable. These are typically the vertices that interest us the most, since they are those most likely to move usefully between blocks.

14.3.3 Pairwise Representation

The visualization in Figure 14.1 is useful for comprehending the general structure of a partition and the relative sizes of the blocks. Figure 14.2 shows the result of a second force-directed system we designed specifically for visualizing pairs of blocks in isolation. We install a spring for each edge between vertices of the block pair and add a rightwards-pulling force to each vertex in proportion to the weight of its edges to the other blocks. Thus, each vertex is pulled towards the top or bottom of the screen, depending on its affinity for the top or bottom block and pulled towards the right depending on its affinity for the other blocks. The vertices in the left center of the screen are those that are strongly connected to both blocks and weakly connected to the other blocks (see Figure 14.2a); a focused search involving these vertices\footnote{A more useful search might focus on these vertices and their close neighbors in the hypergraph structure. Our interface allows the user to expand the current set of focused vertices to include all their neighbors.}
Figure 14.2: Visualizing pairs of blocks.

might provide the opportunity for reducing the cut set between the blocks by exchanging some of these vertices. A better scenario is shown in Figure 14.2d: it is clear that many vertices in the bottom block would prefer to be in the top block, but it is full—the circle surrounds the block icon in the top-left corner—and cannot accommodate them at the moment. However, if vertices can be moved out of the top block without incurring additional cut-set costs (we provide an operator that allows the user to request such an adjustment), the bottom-block vertices might then be included with some likely savings in cut-set cost. Finally, Figures 14.2b and 14.2c show less promising cases: there are few vertices loosely connected to the other blocks that are also strongly connected to both blocks in the pair; moving vertices between these blocks is less likely to reduce the cut-set cost. In terms of efficiency, we found that we could obtain good results with a few seconds of computation per invocation of the spring embedder on a 500 MHz PC.
Chapter 15

A Visual Network Partitioning System

In this chapter we will describe an interactive system for network partitioning devised and implemented at Merl – Mitsubishi Electric Research Laboratories, Cambridge, MA. A description of the visualization aspects involved by the system was presented in [88].

15.1 Interface and Interactive Exploration

We now describe how the user can interactively explore and modify the visualizations described above. The user can move the position of the hubs of each block, and then re-invoke the spring embedder. Moving two blocks farther from each other generally better reveals the relationships between them.

The drawing algorithms are only invoked when the user requests them. This turned out to be very useful for understanding the effect of deploying a refinement algorithm: the nodes that move from one block to another can be easily spotted because they appear in their previous location but with their new block’s color.

15.1.1 Visualization Tuning

Several parameters allow the user to fine tune the drawing method in both the visualization modes described in the previous chapter. The elastic constant and the natural length of each kind of spring in the force directed methods is modifiable by the user.

Although all the edges are considered by the drawing process, they may be selectively visualized based on their properties. Edges can be filtered based on different criteria. The user may ask to visualize only:
• the edges whose weight is greater than a specified value.

• the edges corresponding to hyperedges shared by two (three, four, five, ... ) blocks of the partition.

• the edges incident to the currently selected vertices.

• the edges corresponding to hyperedges that are containend in the same block of the partition but for one vertex (two vertices, three vertices, ...).

This is useful both to improve the comprehensibility of a drawing and to decrease the drawing time for the larger nets.

15.2 User Operations

The Human-Guided Simple Search approach translates to the following search and focus operators available to the user in our network-partitioning system:

1. Manually edit the current partition by moving nodes between blocks

2. Launch a refinement heuristic on the whole network, or on a focused subset of the blocks or nodes.

3. Navigate a history list of previous solutions and revert to an earlier one.

The first two operators are described in the following subsections.

15.2.1 Manual Editing

In the various visualization modes, the user is allowed to select vertices by selecting an area of the screen with the mouse, or by pointing directly at a single vertex. The vertices currently selected feature a white dot in them. An operator allows the user to extend the current selection to the visible neighborhood (taking into account only the visualized edges, that is, those that are not hidden), which can provide additional insight into the relationships among nodes. We have found it especially useful, when switching between visualization modes, to select a set of nodes first in order to understand the relationships between the different visualizations. A selection of vertices can be saved and loaded to a file to be retrieved in other sessions of the system.

The selected vertices may be dragged and dropped on the icon associated with each block, in order to move them to the corresponding block. After each movement the netcut cost is recomputed. The new score and the difference with respect to the previous one is shown to the user.
15.2. USER OPERATIONS

15.2.2 Refinement Heuristics

Several heuristics are available to the user to modify the current partition. The user may launch one of the following:

- A simple algorithm to produce a random partition.
- An algorithm that randomly distributes the currently selected vertices between the currently selected blocks of the partition.
- The hMetis Recursive k-way partitioning algorithm (from the publicly distributed hMetis library) generating a new partition. This algorithm recursively applies a bisectioning method to obtain the desired number of blocks.
- The hMetis k-way partitioning algorithm (always from the hMetis library) to create a new partition.
- The hMetis algorithm, both in its recursive and non recursive versions, on the current partition to redistribute the vertices of the selected blocks exclusively.
- The Fiduccia-Mattheyses refinement algorithm to improve (if possible) the current partition. The refinement could be launched on the whole partition, on the selected blocks only, or on the selected vertices only.
- A simplified version of the Fiduccia-Mattheyses algorithm called Early Exit FM in which the tentative movement of vertices between blocks is interrupted when the current netcut cost is lower than a specified threshold.
- The Sanchis refinement algorithm (on the whole partition, on the selected blocks only, or on the selected vertices only).
- A version of the Sanchis algorithm featuring the early exit strategy.
- An heuristic that forces the movement of a specified number (1, 10, 20, 100, 200, all free moves) of the selected vertices to the selected blocks. The criterion for choosing the vertices to be moved is analogous to that used in the Fiduccia-Mattheyses algorithm (the same data structure is used).
- A Branch and Bound algorithm that considers the selected vertices in the pairwise visualization mode and explores all the possible assignments of such vertices to the pair of selected blocks. The search space is limited with considerations about the constraints on the sizes of the blocks.
The search space can be further limited by specifying the maximum number of vertices that can change their block with respect to the current partition (2-ply, 3-ply, 4-ply, ...).

- A novel algorithm we called Fix-All-Fixable that moves sets of vertices from one block to the other trying to fix whole hyperedges.

Also, the partition can be saved and load in a file with a simple textual format.
Chapter 16

Drawing Infinite Trees

We study the problem of designing layout facilities for the navigation of an “infinite” graph, i.e. a graph that is so large that its visualization is unfeasible, even by gluing together all the screen snapshots that a user can take during the navigation. We propose a framework for designing layout facilities that support the navigation of an infinite tree. The framework allows us to exploit the knowledge of future moves of the user in order to reduce the changes in her mental map during the navigation. Variants of the classical Reingold-Tilford algorithm are presented and their performance is studied both experimentally and analytically. The content of this chapter is published in [29].

16.1 Introduction

Designing layout facilities for the navigation of “infinite” graphs is one of the challenges of the Graph Drawing field. By “infinite” we mean that it would be unfeasible to visualize the graph entirely, even by gluing together all the screen snapshots that a user can take during the navigation. Examples of graphs that can be considered infinite in the above sense come from disparate application areas including World Wide Web, Knowledge Bases, Communication Networks, and Semantic Networks. Several papers on the visualization of very large graphs appear in the literature. A limited list includes [65, 93, 74, 138, 99, 98, 37]. However, the classical assumption is that the input graph, even if huge, is completely known in advance.

We adopt a different point of view making the assumption, similar to the one in [39], that either the graph is too large to be entirely known or it is practically unfeasible to draw it all. Hence, during the navigation, the user is allowed to see only the content of a limited size window that she can move for exploring the graph. It is clear that such a window cannot be purely geometric: If the graph is so large to be impossible to visualize or even to know, then
the idea of moving a window on a pre-computed drawing does not make sense. Therefore, our idea of window is that of a topological window, defined in terms of the structure of the graph. For example, at each time of the navigation the topological window may display the subgraph induced by the vertices that have a constant topological distance from a given vertex, considered as the center of the window.

Hence, the graph is displayed as a sequence of drawings determined by the navigation path followed by the user. Since consecutive drawings in the sequence can have a large overlap, it is essential to devise visualization strategies that preserve the mental map [43, 94, 77, 15] of the user. Algorithms devised to preserve the mental map are usually evaluated in terms of both static and dynamic quality measures [95, 22]. Static measures evaluate standard Graph Drawing aesthetics like number of crossings or area used by the drawing. Dynamic measures evaluate how much a drawing is similar to the one(s) preceding it in the sequence.

Our work starts from two main observations: (1) While the display area is bounded by the size of the screen, the size of the portion of graph that the system can know at each instant of time is only bounded by the resources of the computer. (2) Very often, even if the graph is infinite, the visit of vertices follows an almost predictable pattern. For example, it can be experimentally verified that most users access a Web site following a limited set of favorite patterns. Thus, the following question naturally rises. Is it possible to devise a drawing strategy that takes advantage of the knowledge of part of the future navigation moves of the user in order to achieve better performance in terms of dynamic quality measures? In other words, suppose that the user is observing the drawing of a certain subgraph $G_1$ and suppose to know in advance that the next navigation step will require the visualization of a subgraph $G_2$; given this knowledge, is it possible to draw $G_1$ so to reduce the changes in the overlapping portions of the drawings of $G_1$ and $G_2$? The idea is to exploit the knowledge of future moves in the current visualization. It is interesting to note that the importance of knowing in advance the sequence of graphs to display within an off-line visualization framework has been discussed in [105].

The main components of our framework are: A visualization window defining, at each step of the navigation, what is the graph (a portion of the infinite graph) to visualize. A knowledge window defining, at each step of the navigation, a supergraph of the visualized one. The size of the knowledge window affects the drawing of the visualization window. A favorite neighbor for each vertex of the infinite graph. The favorite neighbor defines what is the next navigation step that the user is going to perform.

As an application of the above framework, we study the problem of navigating in an infinite tree. In spite of the structural simplicity of trees, designing drawing strategies for navigating in infinite trees offers several experimental
and theoretical challenges. Also, the Web is often visualized as a tree-like structure (see e.g. [5, 71]). We devise variants of the classical Reingold-Tilford algorithm [119]. This choice is motivated by the fact that the Reingold-Tilford algorithm is among the most robust, simple, effective, and well-known algorithms of Graph Drawing. The main results of this chapter can be listed as follows. We formally define a framework for designing Graph Drawing facilities that support the navigation of an infinite graph (Section 16.2). The framework is especially targeted to take into account the knowledge of the future moves of the user. We design a set of simple strategies based on the Reingold-Tilford algorithm for visualizing infinite trees (Section 16.2). We perform an experimental analysis of the above strategies (Section 16.3). The analysis shows that exploiting the knowledge of the future moves of the user is not a trivial task, and that some simple strategies can have more drawbacks than benefits from the knowledge of the future. Motivated by the experimental results, we perform a theoretical analysis of the drawing algorithms (Section 16.4). The analysis both explains our experimental results and allows us to compare the performance of the drawing algorithms as the size of the knowledge window goes to infinity.

Through the paper we make use of standard Graph Drawing terminology [30].

16.2 The Framework and the Drawing Strategies

All our trees are finite subtrees of an infinite tree $T$ and each edge is oriented from the parent to the child. The time of the framework is discrete. Hence, it assumes integer values starting from 0. At instant $t$ the user can look at a visualization window of a given positive integer size $h$. The visualization window is a subtree $T_h(t)$ of $T$ rooted at a certain vertex $r(t)$ and induced by the vertices of $T$ at (oriented) distance at most $h$ from $r(t)$. Vertex $r(t)$ is the point of view of the user at instant $t$. If the point of view at instant $t$ is $r(t)$, then the user can move it at instant $t + 1$ to any child of $r(t)$. A navigation of $T$ in any interval $[t_1, t_2]$ of time consists of the path (visualization path) $r(t_1), \ldots, r(t_2)$. The overlap between two visualization windows at consecutive instants $t$ and $t + 1$ is $T_h(t) \cap T_h(t + 1) = T_{h-1}(t + 1)$. An overlap between consecutive visualization windows is illustrated in Figure 16.1.

From the point of view of the readability of the visualization, we consider static and dynamic quality measures. The static quality measures evaluate the drawings of $T_h(t)$ and of $T_h(t + 1)$ separately. Among the possible static quality measures we focus on the width of the drawing, since it is one of the few degrees of freedom when drawing a rooted tree. The dynamic quality measures evaluate the variations in the overlap of the drawings of $T_h(t)$ and $T_h(t + 1)$.
Figure 16.1: Two successive snapshots produced by the system Leonardo [25]
showing the effect of a single navigation step in a randomly generated tree;
the light boxes show $T_h(t)$ and $T_h(t + 1)$, the shaded boxes show the portion
of the tree visible both before and after the navigation step, that is $T_{h-1}(t + 1)$.
The value of $h$ is 3.

Namely, they measure the differences between the drawing of $T_{h-1}(t + 1)$
inside the drawing of $T_h(t)$ and the drawing of $T_{h-1}(t + 1)$ inside the drawing
of $T_h(t + 1)$. In order to optimize the drawing with respect to the dynamic
quality measure we exploit the following two observations: (1) Even if the
visualization window is constrained to have size $h$, the drawing algorithm that
computes the drawing of the visualization window at instant $t$ can know a
subtree of $T$ larger than $T_h(t)$. This subtree is denoted as $T_{h+k}(t)$. It has root
$r(t)$ and is induced by the vertices of $T$ at (oriented) distance at most $h+k$ from
$r(t)$. Constant $k \geq 0$ is an integer that describes how much the algorithm is
allowed to know about $T$ more than what belongs to the visualization window.
Tree $T_{h+k}(t)$ is called knowledge window; $h+k$ is the size of the knowledge
window. (2) Very often, even if the graph is infinite, the visit of vertices
follows an almost predictable pattern. One possibility for modeling this issue
is to weight the children of each vertex with their probability to be visited
after their parent. We make here the simpler assumption that for each vertex
one favorite child is defined. Under this assumption the visualization path in
any interval $[t_1, t_2]$ is $r(t_1), \ldots, r(t_2)$, where $r(t + 1)$ ($t_1 \leq t < t_2$) is always
the favorite child of $r(t)$.

The drawing algorithms that we study are variants of the classical Reingold-
Tilford algorithm [119]. This choice is motivated by the fact that the Reingold-
16.2. THE FRAMEWORK AND THE DRAWING STRATEGIES

Tilford algorithm is among the most robust, simple, effective, and well known algorithms of Graph Drawing.

Suppose to have a visualization window of size \( h \) and to have a knowledge window of size \( h + k \). At instant \( t \) our algorithms compute a drawing of a tree larger than \( T_h(t) \). Namely, they receive as input \( T_{h+k}(t) \) and perform as follows. They prune \( T_{h+k}(t) \) of the vertices that will never be part of any visualization window during the navigation. The resulting pruned tree is called \( T_{h,k}(t) \) and is defined as \( T_{h,k}(t) = \bigcup_{i=0}^{k} T_h(t+i) \). Then, they compute a drawing of \( T_{h,k}(t) \) and display the portion \( T_h(t) \) of height \( h \).

Note that the definition of tree \( T_{h,k}(t) \) relies on the knowledge of the vertices \( r(t), r(t+1), \ldots, r(t+k) \) of the visualization path. Also, observe that \( T_{h,k}(t) \) has height \( h + k \) and that \( T_{h,0}(t) = T_h(t) \). Figure 16.2 shows a portion of an infinite tree with a visualization window of height \( h = 3 \) and a knowledge window of height \( h + k = 9 \). In the figure, the black vertices belong to the visualization path and together with the grey vertices induce \( T_{h,k}(t) \).

![Figure 16.2: A portion of an infinite tree, a visualization window of height \( h = 3 \), and a knowledge window of height \( h + k = 9 \). A drawing algorithm computes a drawing of \( T_{h,k}(t) \) (grey and black vertices). Only the portion of the drawing inside the visualization window is displayed on the screen. The portion of the tree induced by the white vertices is not taken into account by the drawing algorithm.](image)

Our variants of the Reingold-Tilford Algorithms are characterized by a different choice for the embedding of \( T_{h,k}(t) \). Leftmost-Embedding Algorithm: The embedding of \( T_{h,k}(t) \) is chosen so that for each vertex \( v \) along the visualization path, the favorite child of \( v \) is the leftmost child of \( v \). As a result, the leftmost vertex at any given level of \( T_{h,k}(t) \) is a vertex of the visualization path. Central-Embedding Algorithm: The embedding of \( T_{h,k}(t) \) is chosen so that for each vertex \( v \) along the visualization path, the favorite child of \( v \) is the middle child of \( v \). A vertex \( u \) is the middle child of \( v \) if the number of vertices preceding \( u \) in the adjacency list of \( v \) differs by at most one from the number of
vertices following \( u \) in the adjacency list of \( v \). Random-Embedding Algorithm: 
The embedding of \( T_{h,k}(t) \) is chosen so that the favorite child of each vertex \( v \) along the visualization path can be in any position of the adjacency list of \( v \).

We are interested in evaluating the performance of the above algorithms. 
The \textit{total-shift} dynamic quality measures the change of the \( x \)-value of a non-root vertex \( v \) of \( T_{h-1}(t+1) \) with respect to its parent \( p(v) \). Since our algorithms perform a pruning step before computing the drawing, we evaluate the total-shift dynamic quality measure with respect to pruned subtrees. Namely, we denote the total-shift dynamic quality measure as \( m_D(h, k, t) \) and define it as follows: 
\[
m_D(h, k, t) = \sum_{v \in T_{h-1}(t+1)} |x(v(t), t) - x(p(v(t)), t) - (x(v(t+1), t + 1) - x(p(v(t)), t + 1))|,
\]
where \( x(v(t), t) \) is the \( x \)-coordinate of vertex \( v \) at instant \( t \) computed by an algorithm that considers \( T_{h,k}(t) \). The static quality measure that we use evaluates the width of the drawing of the visualization window. More precisely, let \( B_{h,k}(t) \) be the smallest isothetic box that contains the first \( h \) levels of a drawing of \( T_{h,k}(t) \). We denote as \( m_S(h, k, t) \) the width of box \( B_{h,k}(t) \).

### 16.3 Experimental Results

This section contains the results of an experimental comparison of Random-Embedding Algorithm, Central-Embedding Algorithm, and Leftmost-Embedding Algorithm with respect to the dynamic and static quality measures \( m_D \) and \( m_S \) introduced in Section 16.2. In each experiment we consider a random path on a randomly generated tree. Since we actually perform a finite number \( l \) of navigation steps on each tree, once the visualization window height \( h \) and the knowledge \( k \) have been fixed, it suffices to produce a tree with height \( h + k + l \) to simulate the behavior of the algorithms on an infinite tree. Furthermore, in order to simulate a random infinite path on an infinite tree, we need to avoid choosing those random paths ending with a leaf before the expected length is reached.

The generation of a random tree and of the corresponding random path is performed as follows. In order to generate a tree of given height and with vertex outdegree ranging in a fixed interval, we randomly choose in such interval the number of children of each vertex, according to the uniform distribution, starting from the root and considering one level at a time, until we reach the desired tree height. Then we randomly select a leaf on the last level of the tree, implicitly defining a random path from the root to the chosen leaf. In each experiment we fix the parameters \( h, k, \) and \( l \), we randomly generate a tree of height \( h + k + l \) and a path over it, and we perform \( l \) navigation steps, computing at each step the value of \( m_D(h, k, t) \) and \( m_S(h, k, t) \). For each tree, we average the values of the dynamic and static quality measures over the
Figure 16.3: Experimental results for $h = 5$, $0 \leq k \leq 6$, minimum vertex outdegree 0, and maximum vertex outdegree between 2 and 5. On the left (right) the improvement or worsening of the dynamic (static) quality measure, for Random, Central, and Leftmost Embedding Algorithm, respectively, is shown.
whole navigation and then we average these values over the entire set of randomly generated trees. The resulting measures are denoted as $m_D(h, k)$ for the total-shift dynamic quality measure and $m_S(h, k)$ for width static quality measure. Instead of plotting the raw values of $m_D(h, k)$ and $m_S(h, k)$, we display their relative improvements, denoted as $i_D(h, k) = \frac{m_D(h, 0) - m_D(h, k)}{m_D(h, 0)}$ and $i_S(h, k) = \frac{m_S(h, 0)}{m_S(h, k)}$, respectively.

Figure 16.3 shows the results of the experiments performed with $h = 5$, $0 \leq k \leq 6$, minimum vertex outdegree 0, and maximum vertex outdegree between 2 and 6. In particular, Figure 16.3.a, Figure 16.3.c, and Figure 16.3.e show the values of $i_D(h, k)$ for the Random, Central, and Leftmost-Embedding Algorithm, respectively, while Figure 16.3.b, Figure 16.3.d, and Figure 16.3.f show the corresponding values of $i_S(h, k)$. Each point on the diagrams is obtained computing the average over 1000 trees. The axes scales are chosen with the purpose of producing diagrams in the range $[0, 1] \times [0, 1]$. Concerning the static quality measure, the behavior of $i_S(h, k)$ shows the expected worsening of the width of the drawing $k$ increases. For what concerns the dynamic quality measure, the diagrams show that the best values are obtained by Leftmost-Embedding Algorithm and the worst values by Central-Embedding Algorithm. In all charts, as the average outdegree of tree $T$ increases, the dynamic improvement for any fixed value of $k$ decreases. Also notice that $i_D(h, k)$ measured for Random and Central-Embedding Algorithm can be negative when the outdegree of the vertices increases (see Figure 16.3.a and Figure 16.3.b). This shows that not all strategies are able to take advantage of the knowledge of the future. On the other hand, $i_D(h, k)$ is always positive for Leftmost-Embedding Algorithm, which performs better than the other two in all experiments that we have run.

The charts relative to Leftmost-Embedding Algorithm illustrate tradeoffs between the values of the dynamic and static quality measures. In particular, as $k$ increases, there is an improvement of the performances with respect to $m_D(h, k)$ and a worsening with respect to $m_S(h, k)$. We can summarize the outcome of our experimental study as follows. Each level of knowledge identifies a tradeoff between the aesthetics of the drawing and the preservation of the mental map. An effective approach to the design of a drawing algorithm that achieves a pleasing compromise between aesthetic requirements and preservation of the mental map is based on using only a limited knowledge of the future.

### 16.4 Analysis

In this section we analytically compare the performances of Leftmost-Embedding Algorithm and of Central-Embedding Algorithm. This comparison
16.4. ANALYSIS

considers the behavior of the total-shift quality measures for both algorithms when \( k \) goes to infinity. The analytical behavior of the curves of \( m_D(h, k, t) \) for large values of \( k \) is consistent with the behavior observed experimentally for small values of \( k \). We start by introducing some definitions and basic tools. Let \( v \) be a vertex of \( T_{h,k}(t) \) and consider a drawing \( \Gamma \) of \( T_{h,k}(t) \). The difference between the \( x \)-coordinate of \( v \) and the \( x \)-coordinate of \( p(v) \) in \( \Gamma \) is denoted as \( \delta_{h,k}(v, t) \), i.e., \( \delta_{h,k}(v, t) = x(p(v), t) - x(v, t) \). The following lemma shows the interplay between time and knowledge by relating the values of \( \delta() \) at consecutive instants of time with the values of \( \delta() \) for consecutive values of \( k \).

**Lemma 7** Let \( v \) be a non-root vertex in \( T_{h,k}(t) \cap T_{h,k}(t+1) = T_{h,k-1}(t+1) \). For any drawing produced by **Leftmost-Embedding Algorithm** (**Central-Embedding Algorithm**) we have \( \delta_{h,k}(v, t) = \delta_{h,k-1}(v, t+1) \).

**Proof**: \( \delta_{h,k}(v, t) \) is measured on the drawing of \( T_{h,k}(t) \), while \( \delta_{h,k-1}(v, t+1) \) is measured on the drawing of \( T_{h,k-1}(t+1) \). Because \( T_{h,k-1}(t+1) \) is the subtree of \( T_{h,k}(t) \) rooted at \( r(t+1) \), and since **Leftmost-Embedding Algorithm** (**Central-Embedding Algorithm**) draws a subtree with a bottom-up strategy independent of the size of the enclosing supertree, we have that \( \delta_{h,k}(v, t) = \delta_{h,k-1}(v, t+1) \).

From the previous lemma we have that the variation of \( \delta \) from instant \( t \) to instant \( t + 1 \) is equal to the variation of \( \delta \) when the knowledge at instant \( t + 1 \) varies from \( k - 1 \) to \( k \).

**Corollary 1** \( \delta_{h,k}(v, t + 1) - \delta_{h,k}(v, t) = \delta_{h,k}(v, t + 1) - \delta_{h,k-1}(v, t + 1) \).

Corollary 1 allows us to study the variation of \( \delta \) by considering \( T_{h,k-1}(t+1) \) and \( T_{h,k}(t+1) \), instead of looking at \( T_{h,k}(t) \) and \( T_{h,k}(t+1) \). Since \( T_{h,k-1}(t+1) \) and \( T_{h,k}(t+1) \) share the root, we can unambiguously define a **level** for their vertices; we assume that the root has level 0.

### 16.4.1 Analysis of **Leftmost-Embedding Algorithm**

Consider a drawing \( \Gamma \) of \( T_{h,k}(t+1) \) produced by **Leftmost-Embedding Algorithm**. All vertices that have the same level in \( T_{h,k}(t+1) \) also have the same \( y \)-coordinate in \( \Gamma \). Thus, we also talk about levels of \( \Gamma \). Let \( 0 \leq i \leq k \) be a level of \( \Gamma \), let \( r(t + 1 + i) \) be the vertex of the visualization path at level \( i \), and let \( u \) be its rightmost child. We denote with \( \omega(i,k) \) the quantity \( \omega(i,k) = x(u, t + 1) - x(r(t + 1 + i), t + 1) = \lfloor \delta_{h,k}(u, t + 1) \rfloor \). Also, we denote with \( \mu(k) \) the quantity \( \mu(k) = \omega(k,k) \). Let \( 1 \leq i \leq k \) be a level of \( \Gamma \), let \( r(t + i) \) be the vertex of the visualization path at level \( i - 1 \), and let \( r(t + 1 + i) \)
its leftmost child. Also, let \( v \) be the rightmost child of \( r(t+i) \) and let \( u \) be the rightmost child of \( r(t+1+i) \). We denote with \( \lambda(i,k) \) the quantity \( \lambda(i,k) = x(v) - x(u) \). Examples of the quantities \( \lambda(i,k) \), \( \omega(i,k) \), and \( \mu(k) \) are depicted in Figure 16.4.

Figure 16.4: Quantities \( \lambda(i,k) \), \( \omega(i,k) \), and \( \mu(k) \) for the analysis of the drawings produced by \textbf{Leftmost-Embedding Algorithm}.

**Lemma 8** For any positive integer \( k \) and for any integer \( 0 \leq i \leq k - 1 \), we have that \( \lambda(i,k) = \lambda(i,k-1) \).

**Lemma 9** Let \( t \) be an instant of time, and let \( h \) and \( k \) be the integers that define the size of the visualization window and of the knowledge window. For a drawing of \( T_{h,k}(t) \) produced by \textbf{Leftmost-Embedding Algorithm} we have that:

\[
\omega(i,k) = \frac{\mu(k)}{2^{k-i}} + \sum_{j=1}^{k-i} \frac{\lambda(i+j,k)}{2^j} \quad i \in [0,k]
\]

**Proof:** The proof follows from the solution of the following recurrence equation (see also Figure 16.4):

\[
\omega(i,k) = \begin{cases} 
\frac{1}{2}[\omega(i+1,k) + \lambda(i+j,k)] & 0 \leq i < k \\
\mu(k) & i = k
\end{cases}
\]

We are now ready to study the variation of \( \omega \) as \( k \) goes to infinity. In order to ensure a finite limit, we need to bound the values of \( \mu(k) \), \( \mu(k-1) \), and \( \lambda(k,k) \). Lemma 10 shows that this is always possible under the assumption that the maximum vertex degree of \( T \) is finite. The limit for the variation of \( \omega \) as \( k \) goes to infinity is computed in Lemma 11.
Let $T$ be an infinite tree whose maximum vertex outdegree is bounded by a constant. Then there exists a constant $\xi$ such that $\forall k \, \lambda(k,k) - \mu(k) + 2\mu(k-1) \leq \xi$.

**Lemma 11** Let $T$ be an infinite tree whose maximum vertex outdegree is bounded by a constant. Then,

$$\forall i \in [0, k] \quad \lim_{k \to \infty} (\omega(i, k) - \omega(i, k - 1)) = 0$$

**Proof:** In view of Lemma 10, a constant $\xi$ exists such that $\forall k \, \lambda(k,k) - \mu(k) + 2\mu(k-1) \leq \xi$. The claim immediately follows from Lemma 8 and Lemma 9:

$$\lim_{k \to \infty} (\omega(i, k) - \omega(i, k - 1)) = \lim_{k \to \infty} \frac{\lambda(k,k) - \mu(k) + 2\mu(k-1)}{2^{k-1}} \leq \lim_{k \to \infty} \frac{\xi}{2^{k-1}} = 0$$

The following lemma provides a concise formula for the sum of the contributions to the dynamic quality measure $m_D(h, k, t)$ due to the vertices laying on the same level of $T_{h,k}(t)$.

**Lemma 12** Let $t$ be an instant of time, and let $h$ and $k$ be the integers that define the size of the visualization window and of the knowledge window. For a drawing of $T_{h,k}(t)$ produced by the Leftmost-Embedding Algorithm, we have that $\forall i \in [0, h - 2]$:

$$\sum_{v \in T_{h-1}(t+1) \text{ at level } i + 1} |\delta_{h,k-1}(v, t + 1) - \delta_{h,k}(v, t + 1)| = \text{outdeg}(v) |(\omega(i, k) - \omega(i, k - 1)|$$

We are now able to express the dynamic quality measure as a function of $\omega(i, k)$ and to compute the limit of the improvement $i_D(h, k, t) = \frac{m_D(h, 0, t) - m_D(h, k, t)}{m_D(h, 0, t)}$ as $k$ goes to infinity.

**Theorem 12** Let $t$ be an instant of time, and let $h$ and $k$ be two integers that define the size of the visualization window and of the knowledge window. For a drawing of $T_{h,k}(t)$ produced by the Leftmost-Embedding Algorithm, let $m_D(h, k, t)$ be the total-shift dynamic quality measure. Then

$$\lim_{k \to \infty} \frac{m_D(h, 0, t) - m_D(h, k, t)}{m_D(h, 0, t)} = 1$$

**Proof:** We recall from Section 16.2 the definition of the dynamic quality measure:

$$m_D(h, k, t) = \sum_{v \in T_{h-1}(t+1) \setminus \{r(t+1)\}} |x(p(v), t) - x(v, t) - (x(p(v), t + 1) - x(v, t + 1))|$$
According to the definition of δ we have that \( x(p(v), t) - x(v, t) = \delta_{h,k}(v, t) \) and \( x(p(v), t + 1) - x(v, t + 1) = \delta_{h,k}(v, t + 1) \) and that, in view of Lemma 7, \( \delta_{h,k}(v, t) = \delta_{h,k-1}(v, t + 1) \). Hence

\[
m_D(h, k, t) = \sum_{v \in T_{h-1}(t+1) - \{r(t+1)\}} |\delta_{h,k-1}(v, t + 1) - \delta_{h,k}(v, t + 1)|
\]

Partitioning vertices in \( T_{h-1}(t + 1) \) according to their level we have

\[
m_D(h, k, t) = \sum_{i=0}^{k-2} \sum_{v \in T_{h-1}(t + 1)} |\delta_{h,k-1}(v, t + 1) - \delta_{h,k}(v, t + 1)|
\]

at level \( i + 1 \)

Then, in view of Lemma 12, and supposing \( k \geq h \) in order to use \( \omega \) in its domain:

\[
m_D(h, k, t) = \sum_{i=0}^{k-2} \text{outdeg}(r(t + 1 + i))|\omega(i, k) - \omega(i, k - 1)|
\]

Therefore, the truth of the statement follows from Lemma 11.

16.4.2 Analysis of Central-Embedding Algorithm

In this section we compare the behavior of Central-Embedding Algorithm to that of Leftmost-Embedding Algorithm with respect to the total-shift dynamic quality measure. Consider a drawing of \( T_{h,k}(t + 1) \) produced by Central-Embedding Algorithm. The quantities \( \omega(i, k) \) and \( \mu(k) \) can be defined in the same way as in Subsection 16.4.1. We need two new definitions. Refer also to Figure 16.5. Let \( r(t + 1 + i) \) be a vertex at level \( 1 \leq i \leq k \) in the visualization path and let \( p(r(t + 1 + i)) \) be its parent. Let \( u \) and \( v \) be the rightmost and the leftmost children of \( r(t + 1 + i) \), respectively. Let \( w \) and \( z \) be the rightmost and the leftmost children of \( p(r(t + 1 + i)) \), respectively. We denote with \( \lambda(i, k) \) the quantity \( \lambda(i, k) = x(v) - x(z) \) and with \( \lambda'(i, k) \) the quantity \( \lambda'(i, k) = x(w) - x(u) \).

With reasoning similar to that of Lemma 9 and Lemma 12, the following lemmas can be proved.

**Lemma 13** Let \( t \) be an instant of time, and let \( h \) and \( k \) be the integers that define the size of the visualization window and of the knowledge window. For a drawing of \( T_{h,k}(t) \) produced by Central-Embedding Algorithm we have that:

\[
\omega(i, k) = \mu(k) + \sum_{j=1}^{k-i} \frac{\lambda(i + j, k) + \lambda'(i + j, k)}{2} \quad i \in [0, k]
\]
16.4. ANALYSIS

Figure 16.5: Pruned tree and quantities $\lambda(i, k)$, $\lambda^r(i, k)$, $\omega(i, k)$, and $\mu(k)$ for the analysis of the drawings produced by Central-Embedding Algorithm.

**Lemma 14** Let $t$ be an instant of time, and let $h$ and $k$ be the integers that define the size of the visualization window and of the knowledge window. For a drawing of $T_{h,k}(t)$ produced by Central-Embedding Algorithm we have that for all $i \in [0, h - 2]$:

$$\sum_{v \in T_{h-1}(t+1)} |\delta_{h-1}(v, t+1) - \delta_{h,k}(v, t+1)| = \lceil \text{outdeg}(r_i) - 1 \rceil [\omega(i, k) - \omega(i, k-1)]$$

By means of Lemma 12 and Lemma 14, we can prove the following theorem, which states that for any sufficiently large value of $k$ the performances of Leftmost-Embedding Algorithm are always better than the performances of Central-Embedding Algorithm.

**Theorem 13** Let $t$ be an instant of time, and let $h$ and $k$ be the integers that define the size of the visualization window and the size of the knowledge window. Let $m_D^l(h, k, t)$ and $m_D^r(h, k, t)$ be the total-shift dynamic quality measure of Leftmost-Embedding Algorithm and Central-Embedding Algorithm, respectively. Then $\exists k_0 \geq 0 : \forall k \geq k_0, m_D^r(h, k, t) \geq m_D^l(h, k, t)$.
Chapter 17

Conclusions

Our conclusions regarding the impact of large data sets on graph drawing are that general and canonical approaches, such as the Topology-Shape-Metric one, seem unsuitable for the purpose, and the adoption of the orthogonal drawing convention itself, otherwise so effective when small graphs are involved, is currently to be discouraged both in the two and in the three dimensions. On the other hand, solutions to domain specific problems exist and can be found, moving along the lines of general principles, using suitable techniques, and resolving tradeoffs.

The problem of drawing large graphs is not susceptible to solutions in a simple and unique way, but it seems to have many different solutions as many application domains are considered, and each solution is an original recipe implying several choices regarding the use of general methodologies and techniques, and challenging the designer creativity and imagination.
Bibliography


BIBLIOGRAPHY


[89] Liotta, G., and Di Battista, G. Computing proximity drawings of trees in the 3-dimensional space. In *Proc. 4th Workshop Algorithms Data*


BIBLIOGRAPHY


Università La Sapienza
Dottorato di Ricerca in Ingegneria Informatica
Collana delle tesi
Collection of Theses


June 1995.


VIII-96-1  Luca Cabibbo. *Querying and Updating Complex-Object Databases.* May 1996.


XII-00-1 Marco Daniele. *Advances in Planning as Model Checking*. April 2000.


