Dipartimento di Informatica e
Scienze dell'Informazione

Optimizing and Visualizing Planar Graphs via Rectangular Dualization

by

Gianluca Quercini

Theses Series **DISI-TH-2009-03**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Optimizing and Visualizing Planar Graphs via Rectangular Dualization

by

Gianluca Quercini

June, 2009

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by GIANLUCA QUERCINI
DISI, Univ. di Genova
QUERCINI@DISI.UNIGE.IT

Date of submission: February 2009

Title: OPTIMIZING AND VISUALIZING PLANAR GRAPHS VIA RECTANGULAR DUALIZATION

Advisor: MASSIMO ANCONA
DISI, Univ. di Genova
ANCONA@DISI.UNIGE.IT

Ext. Reviewers: DAVID MOUNT
Department of Computer Science, University of Maryland
MOUNT@CS.UMD.EDU

MICHEL SYSKA
Université Nice Sophia-Antipolis
SYSKA@I3S.UNICE.FR

Abstract

Graphs are among the most studied and used mathematical tools for representing data and their relationships. The increasing number of applications exploiting their descriptive power makes graph visualization a key issue in modern graph theory and computer science. This is witnessed by the widespread interest Graph Drawing has been able to excite in the research community over the past decade. Ever more often software offers a valuable support to key fields of science (such as chemistry, biology, physics and mathematics) in elaborating and interpreting huge quantities of data. Usually, however, a drawing explains the nature of a phenomenon better than hundreds of tables and sterile figures; that is why visualization is so important in modern Computer Science. Graph Drawing is an important part of Information Visualization addressing specific visualization problems on graphs. Most of the research work in Graph Drawing aims at creating representations complying with aesthetic criteria, relying on the principle that a drawing nice and pleasant to see is also effective. Difficulties arise when the size of graphs blows up; in this case appropriate optimization techniques must be used before visualization. One widely used optimization is graph clustering, which lets the representation of a reduced version, called clustered graph, of a large graph. Visualizing a clustered graph is far from being a straightforward task and raises further issues that need to be carefully faced. This thesis investigates a new approach to graph visualization and optimization based on rectangular dualization. A special care is devoted to the description of an efficient algorithm that creates a rectangular dual representation of any planar graph. This is particularly important, as the lack of such an algorithm has been the main reason why rectangular dualization was not so successful as it should have been. It proved, in fact, to have a remarkable impact on a number of applications, ranging from VLSI engineering and Graph Drawing to 3D virtual worlds visualization. These benefits are deeply discussed, mostly focusing on the relevant results obtained in graph visualization. In particular, a generalization of rectangular dualization (called c-rectangular dualization) is introduced as a powerful tool for visualizing and navigating large clustered graphs.

To my parents and my beautiful girlfriend Donatella

Gaudeamus igitur iuvenes dum sumus
Post iucundam iuventutem
post molestam senectutem
nos habebit humus!

Vita nostra brevis est, brevi finietur,
venit mors velociter,
rapit nos atrociter,
nemini parcetur.

Vivat academia, vivant professores!
Vivat membrum quod libet,
vivant membra quae libet,
semper sint in flore.

Vivat et respublica et qui illam regit!
Vivat nostra civitas,
maecenatum charitas,
quae nos hic protegit.

Pereat tristitia, pereant osores!
Pereat diabolus,
quivis antiburschius,
atque irrisores. (Kindleben)

Acknowledgements

After completing the final draft of the thesis, it is time to acknowledge all people who supported me throughout these years. Much of my regret, my memory is awful (actually, I may swear that my grandma's memory is even better than mine); thus, I beg in advance the pardon of whom I may have forgotten.

First, I would like to thank my love **Donatella**, who is the best gift of my PhD here in Genoa. She was by my side in every moment and she never quit to support and spur me. I thank her because she listened to me (or, at least, pretended to listen to me very well) when I taught her basic graph theory! If she did not leave me then, she's unlikely to leave me in future! I thank her for all the love she gave and is giving me. I thank her for making for me delicious cookies and cakes every weekend. Thanks for walking this difficult road by my side, sweetheart!

My parents, **Giovanna** and **Vittorio**, deserve the next mention. They taught me to believe in myself and to follow my dreams with no fears. I thank them for giving me the chance to get such a high level of education and for letting me freely choose the path to my future. I will be grateful to them through all my life!

I give a special mention to all my family: my "big" brother **Stefano** and his wife **Cristina**, who gave me two smart nephews, **Marco** and **Luca**; my brother **Andrea**, his wife **Carolina** and my sweet niece **Selene**; my twin brother **Gianpaolo**, with whom I grew up and shared many experiences and exciting moments; my second twin **Luca**, who is a brother *de facto*; my grandparents, **Angela**, **Arturo**, **Maria**, **Vittorio**, as without them I would never enjoyed this marvellous moment.

At the academic level, I would like to thank my advisor, Prof. **Massimo Ancona**, who backed me in my Master's Thesis and accepted to be my advisor for this PhD thesis. If I reached this important milestone, I owe him, who always believed in me and gave me precious hints for improving my work. I thank Prof. **Bruce Reed**, who was my advisor at Inria, Sophia Antipolis, where I had a 6 months internship, which turned out to be very important. I thank the external reviewers, Prof. **David Mount** and Prof. **Michel Syska** for taking their time to read my thesis and contributing to improve it. I thank Prof. **Peter Eades**, who encouraged me in my work when I held my seminar at the University of Technology in Sydney.

Now it is time to recall all my friends at Disi. **Dave**, with whom I had the privilege to work side by side and write some papers. **Betty**, who was always there when I needed help

in bad moments. **Rocco**, who introduced me to the magic world of Canotto and Baiocco. **Paul**, a person who became a myth in our department and was always there, every single day. **Pastorelli**, or Pasto or Paolino: thanks for destroying my OcORD! Without that bad action, I would not have written a good part of this thesis. **Nicola**, who kept me company when I had to spend my time late at night at the department to complete my thesis. **Laura**, who shared her pain while writing the thesis. And then **Daniela**, **Simona**, **Gillo**, **Dani**, **Sara**, **Massi**, **Adriana**, **Crive**, **Chiara**, **Alessio**, **Daniele**, **Alice**, **Marco**... I also thank the "four little bastards", **Ilaria**, **Giulia**, **Fabio** and **Ulde** (or Ico) for the marvellous photo they took of themselves.

I thank all people who worked on OcORD: **Alessandro Vassalli**, **Paolo Mazzarello**, **Marco Costa** (Cussio), **Paolo Pastorelli**. I drew upon their excellent work. I thanks **Federica Pian** for drawing a marvellous logo for OcORD (which has an awful name) and the family of my girlfriend, **Thea** and **Otta**, above anyone.

We come to very special mentions. **Davide** and **Pamela** definitely deserve this, as they are precious friends of mine, who always understood the bad moments I came across and let me going through them. Finally, I conclude with my adopted sister, **Giada**, who always spent nice words for me and has been supporting me since time; and never disdained to drink a bottle of wine with me: Try ....the wine!!

Table of Contents

Chapter 1

Background

1.1 Introduction

A graph is an abstract representation of some kind of *relationship* in a given set of *entities*. More formally, a graph G is composed of a set of *nodes* or *vertices*, corresponding to the entities, and a set of *edges*, which are connections among the nodes. As such, this definition says everything and nothing. At least, this is the impression I got each time I found myself explaining what my job was about to people who are not keen on the subject. In particular, the word "graph" seemed to mislead my interlocutors, who always thought of pie charts or bar graphs. The only way to get my interlocutors to understand was to show them a drawing of a graph (Fig. 1.1). This is the power (and the aim) of visualization! New (and possibly difficult) concepts turn out to be immediately clear, when they are properly visualized. Talking of graphs to a computer is totally different. Computers better understand a graph as a data structure and do not care of visualization. Consequently, to make humans and computers understand each other, a drawing needs to be translated into a data structure and, vice versa, a data structure needs to be translated into a drawing. The first task takes a lot of coding, but is relatively easy; the latter is the subject of an important branch of graph theory known as *Graph Drawing*.

The application domain of graphs covers different disciplines, ranging from chemistry and biology to cartography and software engineering. But they are also useful to solve problems which are more closely related to our every day life. Suppose that Bob wants to fly from Rome to Sydney; like most of us, he connects to an Internet site (such as Expedia or EDreams), fills in a form and submits his request. In few seconds (hopefully), the web browser provides Bob with a list of travel options. The answer to Bob's request is given by a computer program, running on a web server, which looks for the "best path" to fly from one city to another. The best path is chosen based on Bob's preferences; suppose, for

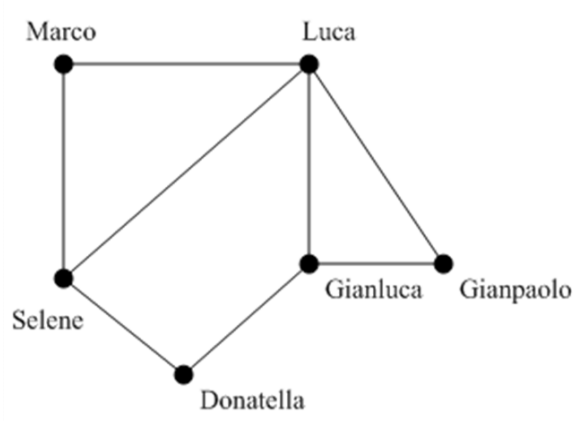


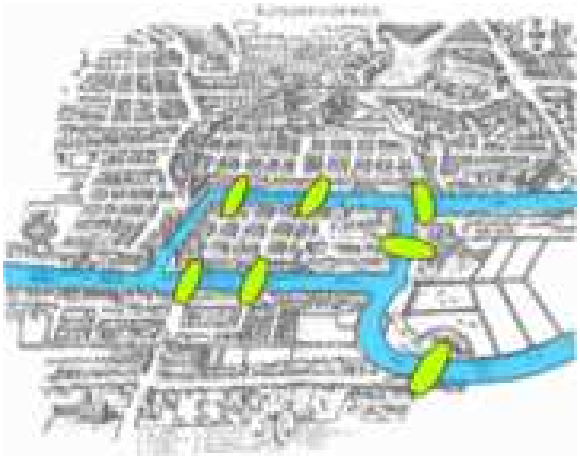
Figure 1.1: Example of a graph describing the relationship "friend of".

example, that Bob wants to minimize the number of flight changes. In order to efficiently find the best path, the server program uses a graph, which represents the relationship "directly connected" defined on the set of the airports. Two airports are directly connected if and only if at least one flight serves them directly. The problem of finding the best path from Rome to Sydney reduces to the famous *shortest path problem* on a graph, for which efficient algorithms exist.

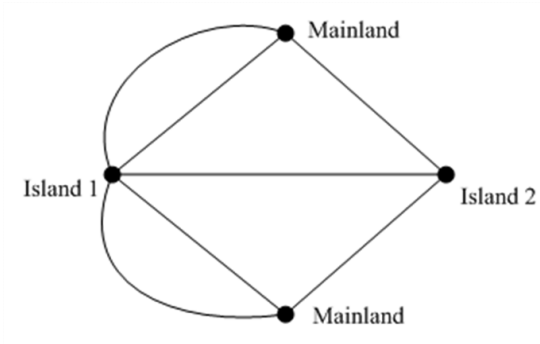
The history of graphs began in 1736 with the famous paper written by Leonhard Euler on the problem of the seven bridges of Königsberg [Eul41]. Königsberg (now Kaliningrad) is a town in Prussia set on both sides of the Pregel River and has two islands connected to each other and the mainland by seven bridges (Fig. 1.2 (a)). The problem consists in finding a walk through the town, such that each bridge is crossed only once.

Euler noticed that only the sequence of crossed bridges was important to solve the problem; thus, walks within an island or the mainland were not to be taken into account. Based on this, he reformulated the problem in abstract terms, modelling it with the graph shown in Fig. 1.2 (b). Euler observed that, except for the two landmasses taken as start and end points of the walk, whenever one enters a landmass by a bridge, one has to leave it by another bridge. Thus, if every bridge is traversed only once, at least two landmasses (vertices) must have an even number of bridges touching them. This is a contradiction, as all the landmasses have an odd number of bridges incident with them. Therefore, the problem has no solution. Restating this in graph-theoretical language, there exist a *walk* in a graph which traverses each edge only once if and only if the graph is connected and exactly zero or two nodes have an odd number of incident edges. Such a walk is now called *Eulerian path* in his honour.

The key actor of this thesis is a special graph called *rectangular dual*. *Rectangular dualiza-*



(a)



(b)

Figure 1.2: (a) The seven bridges of Königsberg (b) The graph used by Euler to solve the problem. The two non-adjacent nodes represent opposite sides of the mainland.

tion was introduced in the early seventies to solve a space planning problem in architecture. It was then successfully used to create floorplans of VLSI circuits. However, its potential, in our opinion, has not been fully investigated, as it would have deserved to. This is greatly due to the lack of efficient algorithms, which this thesis tries to answer. Our main results are related (but not limited) to graph visualization. For this reason, this chapter gives a comprehensive introduction to graph drawing and information visualization.

1.2 Graph Drawing

Apparently, there is no need to visualize a graph when it is treated as a data structure. Indeed, drawing the path from Rome to Sydney in the example of the previous section seems to be helpless. Usually, a textual description such as: "Take the flight X from Rome to Hong Kong and then the flight Y from Hong Kong to Sydney" is just enough. However, suppose now that Bob's homework (Bob is a computer science student besides a travel lover!) consists in implementing an algorithm which, given a graph, finds a maximal set of edges, such that no two of them is incident with the same vertex. Typically, each vertex and edge is identified with a natural number. If the algorithm outputs the required set as a list of integer numbers, how can Bob check whether his implementation works fine? He might manually draw the graph and keep the correspondence between edges and natural numbers, but he can afford this solution only when graphs are small.

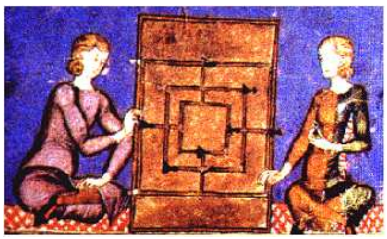
Anyway, besides being a data structure, a graph is also a visualization tool. Drawing has been one of the first communication medium. It is, indeed, an international language,

which gathers all human beings in a sort of a pre-Babel world. Road signs are a clear example of this. Furthermore, often does a drawing easily explicate difficult concepts. Even in everyday language, to reproach ironically somebody who seems not to understand an apparently easy concept, we say: "Do you need a drawing?". In the case of graphs, describing in words relationships in a set of objects is difficult and does not help the reader to understand.

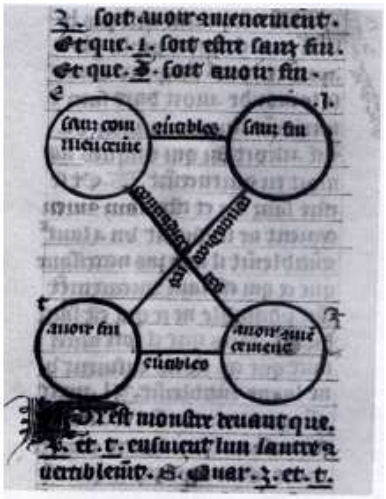
1.2.1 History

Tradition wants Euler to be the father of graph theory and modern graph drawing too. To solve the seven Königsberg problem, in fact, he drew a simple sketch of a graph. This statement, however, is not correct at all, as pointed out in [KMBW02]. In [Eul41], in fact, Euler refers to Leibniz's desire for a new kind of geometry without measures.

Mill games are the oldest examples of graph drawing. Although other (maybe more ancient) games based on the concept of graph, Mill games explicitly use a drawing of a graph (Fig. 1.3 (a)). Another wonderful example is the family tree which used to decorate the atria of patrician Roman villas. Unfortunately, we know about their existence only from Seneca and Pliny the Elder accounts. The first genealogical trees, whose we have a direct witness, date back to the Middle Ages. Graphs were also used as a teaching method to represent and visualize abstract information. The *squares of opposition* (Fig. 1.3 (b)), for example, were used in the Middle Ages to make students understand concepts of logic such as syllogisms.



(a)



(b)

Figure 1.3: (a) A depiction of a Morris game board [KMBW02]. (b) A square of opposition. The nodes are logical propositions and the edges are relations between the propositions [KMBW02].

As far as modern graph theory is concerned, J.B. Listing wrote in 1847 a famous treatise on topology, explaining in particular how to draw paths in graphs. He also included a famous example of a graph, which can be drawn in a single stroke. Sir William Rowan Hamilton (1805-1865) created a game called "Icosian Calculus", where the game board actually is a drawing of a graph. René Just Hüy (1743 - 1822) laid the very first foundations of crystallography using graphs to draw abstract representations of crystals. Many other interesting examples could be enumerated, but their description is out of the scope of this thesis. The interested reader is referred to [KMBW02], for a more detailed history of graph drawing.

1.2.2 Applications

Graph drawing is not just about solving purely mathematical problems (which, anyway, would not be blameworthy at all) but is strictly related to important applications.

Drawing graphs is challenging even for humans, especially when they have many nodes and edges. The main issue is the size of the solution space; indeed, there are potentially infinite ways to draw a graph. What is the best one? The answer depends on the requirements of the application. For example, genealogical trees and underground maps can be represented as trees, but they should be rendered differently. In a genealogical tree, in fact, nodes must be disposed according to a hierarchical structure, whereas in an underground map stations should be placed so as to reflect their geographical position as much as possible. This does not necessarily implies that each application needs an ad-hoc graph drawing algorithm; luckily, there are algorithms which are general enough to adapt to different contexts. In general, the more are the constraints a drawing has to respond, the easier is for an algorithm to create it, as the solution space narrows down.

Workflows are good examples of graph drawings. A workflow displays a sequence of actions and tasks which have to be performed within an organization. Each task is a node and edges represent a relationship between two tasks. Although at the beginning they were received coldly, because accused of dehumanizing the work, workflows became a central tool in industry, especially from the early 1990's. *Software engineering*, for instance, extensively uses workflows to describe the sequence of steps needed to design a software. Visualizing a workflow with the help of automatic tools is not an easy task. First of all, a workflow may be a graph of any kind. Next, nodes have to be arranged so that the flow of the tasks to accomplish is easy to understand by everyone. From the drawing it should be immediately understandable whether two nodes represent two similar tasks or two tasks that have to be accomplished in parallel. Moreover, each node is provided with a textual description which explains what the task is about. Labelling nodes is another challenging task in graph drawing, as the drawing may result confused.

Biology and chemistry are two domains where relationships matter. A *molecule*, for instance, is defined as an electrically stable group of at least two atoms linked by chemical bonds. The geometric and topological study of the molecular structure is necessary to learn about the molecule behaviour. A common technique, used when designing drugs, is evaluating a large set of molecular structures by comparing the graphs representing them [BCF01]. Similar molecules will show similar patterns and will have similar subgraphs. The problem is how a chemist can recognize these similarities by simply looking at the drawings. In [BCF01] graph visualization algorithms are given to represent molecular structures in a standardized way.

In the very last years, the expression *social network* is on everyone's lips, because of the impressive popularity gained by social network internet sites such as LinkedIn or Facebook. A social network is a group of people which are related by some kind of social relationship (friendship, co-workers, collaborators and so on). The study of social networks is highly relevant to understand the dynamics of modern society: how people relate to each other when they work together, why some countries seem to develop more than others and so on. The need of visualizing a social network has already come up in the 30's [Fre00]. Moreno first proposed graphs as a natural way to represent a social network [Mor32]. He also described several techniques to make drawings more understandable and clear. In particular, he claimed that non-planar graphs should be avoided as much as possible. It is not easy to find patterns in a social network from a drawing of the corresponding graph. Furthermore, no common rule states what a good drawing looks like. This, in fact, strongly depends on the kind of patterns which are looked for.

1.2.3 Aim of Graph Drawing

The aim of graph drawing is finding efficient algorithms to create "nice" drawings. Clearly, it is an ambitious goal. Coding aesthetic rules and making an algorithm understand them is something that goes beyond the traditional computer applications. Aesthetic tastes, in fact, are supposed to be a prerogative of human beings. However, the duty is somewhat simplified in that a graph has a structure, which severely constraints the way to represent it.

The basic requirement is *planarity*. A planar graph can be represented in the plane without edge crossings, which makes the graph more readable. Unfortunately, very often are graphs non-planar; social networks are such an example. In this case, the objective is to minimize the number of edge crossings as much as possible. Some work deals with representing non-planar graphs [BK97, PT00, CKW04]; however, only planar graphs will be taken into account in this thesis.

It is quite difficult to judge whether a drawing is good or not. On one hand, in fact, it has

to meet some constraints which are application-dependent; in other words, the drawing depends on the *semantic* of the data being represented. As said before, drawing a tree describing the London underground with an algorithm which is thought for displaying genealogical trees is not a good choice. On the other hand, a set of aesthetic criteria have been devised to evaluate the effectiveness of a drawing:

- **Edge crossing minimization.** If a graph is not planar, some adjacencies are broken so that to represent the least edge crossings possible.
- **Bend minimization.** In certain drawings not every edge can be rendered as a straight line. Consequently, either it is represented by a curve, or by a chain of segments approximating the curve. A *bend* is a point at which the edge changes direction; the more are the bends per edge, the more is difficult to understand the drawing.
- **Area minimization.** In order to save space, the vertices of the graph should not be spread out. Moreover, if the area of a drawing is more than that strictly needed, the information is more confused.
- **Angular resolution maximization.** The *angular resolution* is the smallest angle between two edges incident with the same vertex. Maximizing the angular resolution is difficult for *high-degree vertices*, that is vertices with many incident edges. In this case, in fact, edges incident with the same node may need to be drawn much close to one another.
- **Symmetry maximization.** The drawing should enhance the symmetries in the graph, if any.
- **Aspect ratio.** It is the ratio between the lengths of the two sides of the smallest rectangle enclosing the drawing. This is a crucial aspect, especially when the drawing must be rendered on a screen.
- **Length minimization.** The length of the longest edge should be minimized, to minimize the area.
- **Face shape.** If faces have a regular shape (e.g. they are all rectangles or convex polygons), the whole drawing looks nicer.

There is no agreement on which constraint should be prioritized. Purchase states that it is more important minimizing edge crossings, while bend reduction and symmetry maximization have little impact [Pur97]. For social networks, the aesthetic constraints listed above are too generic. In [BMK96] an experiment is described, aiming at evaluating the impact of different criteria. The position of a node with respect to the centre of the drawing, for

example, reflects the importance of that node in the social network. It can also be objected that many of the described rules have little to do with aesthetic and were motivated by practical reasons. Minimizing the area of a circuit (or the length of the wires), for instance, makes the circuit faster and more efficient.

1.2.4 Drawing Styles

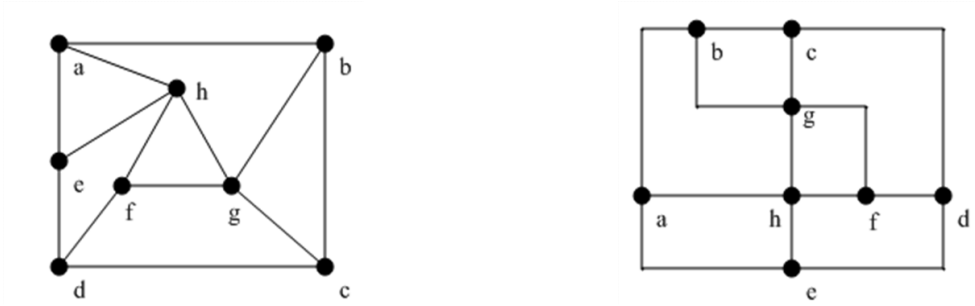
When we are introduced to graph theory at the University, we get used to represent nodes as points in the plane and edges as straight lines. This is called a *straight-line drawing* (Fig. 1.4 (a)). Fary proved that every planar graph admits a straight-line drawing [Far48]. Straight line edges definitely improve the readability of the graph. A key issue is to find a good compromise between area and angular resolution. While minimizing the first, the latter decreases and maximizing the latter increases the first. Every plane graph admits a straight-line embedding with $O(n^2)$ area [DFPP90, Sch90]; however, this bound does not take into account angular resolution, which can be poor, especially for high-degree vertices.

(a) (b)

Figure 1.4: (a) A straight-line drawing. (b) A polyline drawing.

In a *polyline drawing* an edge is rendered as a polygonal chain (Fig. 1.4 (b)). With respect to straight-line drawing, a polyline drawing is more flexible, as curved edges are allowed with an arbitrary number of bends. The implementation is easy because each curve can be mathematically modelled as a Bezier curve. The main disadvantage, however, is a poor readability: edges with a high number of bends may result to be difficult to follow.

Figure 1.5 (a) shows an example of *convex drawing*. All edges are rendered as straight lines and faces as convex polygons. Not all graphs admit a convex drawing. In 1960 Tutte proved that all planar 3-connected graphs admit a convex drawing [Tut60a]. Thomassen characterized the class of graphs admitting a convex drawing in [Tho80]. Based on this result, Chiba et al. described an algorithm which creates convex drawings in linear time [CON85].



(a) (b)

Figure 1.5: (a) A convex drawing. (b) An orthogonal drawing.

A popular drawing style is *Orthogonal Drawing* (Fig. 1.5 (b)). Each edge is drawn as a chain of horizontal and vertical segments. Usually, vertices and bends are points of a grid with integer coordinates. Orthogonal drawing is particularly suitable for drawing circuits and entity-relationship diagrams. Primary goals are maximizing the angular resolution and minimizing the area of the drawing. Any tree admits a planar orthogonal grid drawing with $O(n)$ area and any planar graph of degree at most 4 admits a planar orthogonal grid drawing with $O(n^2)$ area [Shi76, Val81]. Minimizing the number of bends is a key issue. Tamassia and Garg proved that testing whether a graph admits a planar orthogonal drawing with no bends is an NP-complete problem [GT02]. Consequently, finding a drawing with a minimum number of bends is NP-hard. An efficient algorithm which minimizes the bends in a fixed embedding setting is described in [Tam87]. Three heuristic algorithms creating drawings with $O(n)$ bends are presented in [Sto84].

The major limitation of orthogonal drawing is that it requires a graph to have maximum degree four. This can be worked around in three ways (Fig. 1.6). The first approach is the *box orthogonal drawing*, which consists of drawing vertices as rectangular boxes [TDBB88]; the incident edges are connected to the four sides of the box in a prescribed order. As pointed out in [EFK00], the size of each rectangular box may blow up independently of the vertex degree. A better approach, known as *Kandinski*, represents vertices as squares of equal size, disposed on a grid [FK96]. The price to pay for keeping the vertices small is an high number of bends. Moreover, if the grid is too small, the angular resolution is poor. A third approach is the *quasi-orthogonal drawing* [KM98]. The vertices are still represented as points on a grid and edges run mainly on grid paths. The only exception to this rule is allowed for some edges incident with vertices with degree higher than 4. The main disadvantage of this solution is a poor angular resolution.

There are also some variants of the orthogonal drawing, obtained by introducing some constraints. Fig. 1.7 (a) shows a *rectangular drawing*, which is an orthogonal drawing with all faces being rectangular-shaped. A rectangular drawing is not possible for every graph;

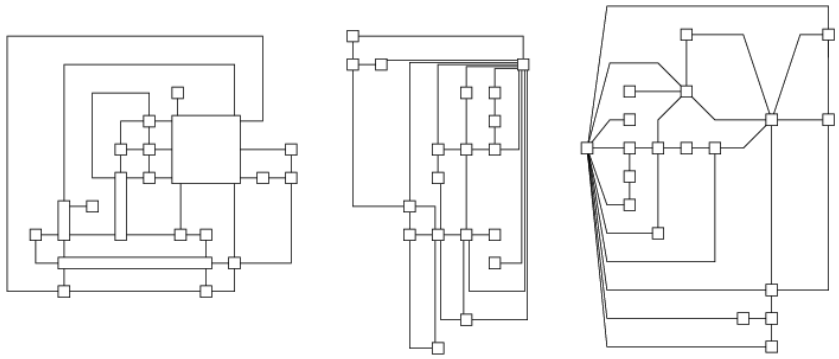


Figure 1.6: From left to right: a box orthogonal drawing, the Kandinski model and a quasi-orthogonal drawing. Figure taken from [KM98].

Thomassen [Tho92] and Rahman [RNG04] independently gave necessary and sufficient conditions. In particular, a rectangular drawing of a subcubic graph (a graph with maximum degree 3) is a rectangular dual. A *box-rectangular drawing* is a rectangular drawing where vertices are represented as rectangular boxes; it is useful to draw graphs with unbounded maximum degree.

A *visibility representation* (Fig. 1.7 (b)) maps vertices to horizontal segments and edges to vertical segments drawn between visible vertex segments. Two parallel segments are visible if they can be linked by a segment orthogonal to both which does not cross any other segment. A visibility representation has many advantages: no bends arise, the drawing is highly readable and textual labels can be easily added to the vertices as they have a large size. The main drawback is that the size of a vertex is not predictable. A straightforward generalization of a visibility representation is called *2-visibility representation*, which draws vertices as rectangular boxes.

Finally, *3D Graph Drawing* received great attention in the last decade. One of the key issues in Graph Drawing is visualizing large graphs. As the number of vertices and edges increases, it becomes more difficult to enforce the aesthetic constraints listed in Section 1.2.3. Usually, techniques such as *clustering* and *fisheye distortion* (see Section 1.3) are used to decrease the complexity of a graph. 3D Graph Drawing is an attempt to create more "space", adding a new dimension. A straightforward way to create a 3D drawing of a graph is to generalize the drawing styles described in this section. For example, in [BEF+98] a 3D visibility representation is described. Some representations have been specifically thought for 3D, such as the *cone tree* [RMC91]. Displaying graphs in 3D introduces new problems. First, a node in 3D can hide other nodes and this happens in all the possible 3D layouts of a graph. Occlusions can be worked around only with techniques which are independent

(a) (b)

Figure 1.7: (a) A rectangular drawing. (b) A visibility representation.

of the drawing: transparencies, rotations and so on. In a certain sense, a drawing in three dimensions is forced to be interactive, with all which this implies. Gaining more space is not the only motivation of 3D Graph Drawing. Since we live in a 3D environment, 3D Graph Drawing creates a real-world metaphor which helps to better perceive complex structures.

1.2.5 Algorithms for Drawing Graphs

Fleischer classified the graph drawing algorithms into four classes [FH01]: force-directed methods, Sugiyama-like methods, flow methods and interactive drawings.

Force-directed methods consider the graph as a system of objects ruled by forces acting between them. The vertices are the objects of the system interacting with each other based on a given force; the edges are objects which do not interact and add new forces to the vertices. A classical example is the analogy with springs connecting electrically charged particles: the vertices are the particles which repel each other and the edges are springs that connect the particles. When springs have a shorter length than their natural length, they exert a repulsion force; otherwise, they exert an attraction force. The drawing of the graph is obtained when the system reaches an equilibrium state, that is when the sum of the forces acting on each vertex is zero. This model is known as the *spring embedder* and was presented by Eades in 1984 [Ead84]. Two forces are used: a *spring force*, which ensures that the length of each edge is approximately equal to the natural length of the spring, and an *electrical force*, which prevents nodes from being too close to each other. To describe the force exerted by a spring, Eades used a logarithmic function instead of Hook law, because otherwise the forces would have been too strong. Spring Embedder is an efficient algorithm; it is easy to understand, due to the real-world metaphor with a spring system, easy to implement and can be extended to 3D. The running time is quite slow, if compared with other methods, but the results are good. Historically, the *barycentre method*, proposed

by Tutte in 1960 [Tut60b], is considered as the first force-directed method. Tutte algorithm creates a convex drawing without edge crossings of a given three-connected planar graph. The idea of the algorithm is simple; the vertices of a given face (say the exterior one) are pinned down and the position of the remaining vertices is given by a convex combination of the positions of its neighbours. Therefore, drawing a graph reduces to solving a system of linear equations. In 1989 the algorithm of Kamada and Kawai introduced the so-called *graph theoretic distance* approach [KK89]. While the distance between two nodes is an Euclidean distance in the drawing , the distance in the graph is the length of the shortest path between the two nodes. The aim of the approach is to minimize the difference between the two distances. The resulting algorithm turned out to be expensive, but contributed to a simple and intuitive definition of "nice drawing". These techniques are particularly suitable for drawing small graphs, containing up to 50 vertices. More recently, advanced approaches have been proposed to produce good layouts for graphs with over thousands vertices [HH01, HK00, Wal01, QE01].

The Sugiyama-like algorithms produce *layered drawings*, which are the standard way in which directed graphs are drawn. They owe their names to Sugiyama, who in 1981 described the most famous algorithm for this kind of drawing [STT81]. In a layered drawing, most of the edges are drawn in one specific direction (usually from top to bottom), with no crossings (if possible) and as straight as possible. Given a directed graph G, the Sugiyama algorithm goes through four steps. In the first step, G is made acyclic by reversing appropriate edges; this way, the edges can be drawn in just one direction in the next step. In the second step, vertices are assigned to horizontal layers, such that all edges can be directed downward. Moreover, the endpoints of each edge must belong to two adjacent layers. The third step computes an ordering of the vertices so as to minimize edge crossings. In the fourth step, the x-coordinate is computed for each vertex, ensuring that no vertex lays on the straight line between two adjacent vertices. Finally, the edges reversed in the first step are reversed again. Almost all steps of this algorithm are NP-complete problems; for this reason the method has been extensively studied and several heuristics have been proposed.

A clear example of a *flow-based method* is the algorithm of Tamassia, which creates a drawing with the smallest possible number of bends for a planar graph $G = (V, E)$ with degree at most four and with a fixed embedding [Tam87]. In any orthogonal drawing of G, the angles $\phi(e_1, e_2)$ between two adjacent edges e_1 and e_2 are multiples of $\frac{\pi}{2}$. Using the quantity $\psi(e_1, e_2) = 2 \cdot \phi(e_1, e_2)$, the values of each angle belong to the interval $[0, 4]$. A network can be created in the following way (Fig. 1.8):

1. A node n_v is created for each node $v \in V$.
2. A node n_f is created for each face f of G.
3. An edge $a_{v,f}$ is added in such a way that $a_{v,f}$ is directed from node n_v to node v_f if vertex v is incident with face f.

4. For any two adjacent faces f_1 and f_2 two edges are added: a_{f_1,f_2}, directed from n_{f_1} to n_{f_2}, and a_{f_2,f_1} directed from n_{f_2} to n_{f_1}.

5. There is a source node s connected to all nodes n_v and a sink t connected to all nodes n_f.

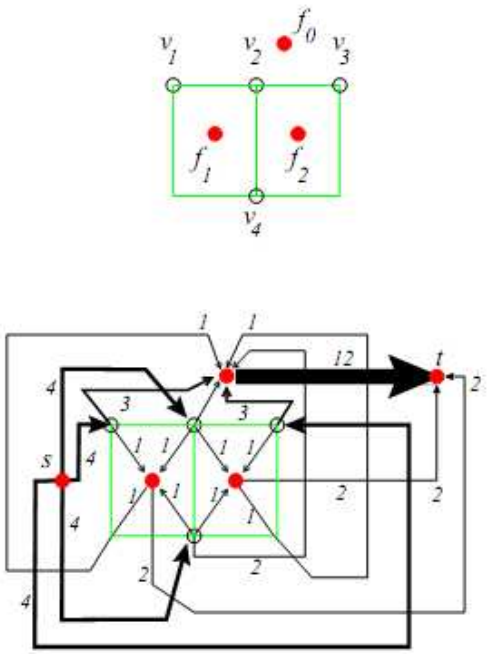


Figure 1.8: From top to bottom: an orthogonal drawing of the input graph; the flow in the corresponding network. The edges of the network are coloured in black. Some edges are omitted as the flow on them is 0. Figure taken from [EFK01].

The problem of finding an orthogonal embedding of a graph with a minimum number of bends is mapped to a minimum flow problem. Each edge is assigned an initial *capacity*, representing the maximum flow value for the edge. The flow on each edge $a_{v,f}$ leaving a node n_v represents the value of the angle incident with vertex v and face f; thus, the initial capacity of these edges is 4, as each angle can not be greater than $2 \cdot \pi$. The flow on each edge a_{f_1,f_2} represents the maximum number of bends of the edges which are incident with both faces f_1 and f_2. The initial capacity is therefore unbounded, as the number of bends is unknown a priori. The flow of the edges leaving the source node s is set to four, as four is the sum of the angles between any two pair of edges incident with a node. A similar technique can be used to maximize angles between edges.

Finally, we spend two words on *interactive drawing*. Many applications work on graphs that change over time and need the correspondent drawing to be constantly updated. Most algorithms modify the input graph, adding edges or nodes. If all edges are drawn as straight lines, it would be desirable that the new edges are straight lines too. A trivial updating algorithm recomputes the whole drawing after each modification. However, this results in waste of time, especially when the graph undergoes only slight updates. Moreover, destroying a drawing and creating another from scratch also destroys the so-called "mental map" of the user [ELMS91]. When a graph is laid out, in fact, the user memorizes the position of nodes and edges. If the drawing is changed too frequently, the user has no time for adapting to it and gets confused. In order to preserve the mental map, two approaches are generally adopted: animations, which highlight the updates, and minimization of changes. For further details, we refer the reader to [Bra01].

1.3 Graph Drawing in Information Visualization

In [CMS99] *visualization* is defined as:

> *the use of a computer-supported, interactive, visual representations of data to amplify cognition.*

Therefore, visualization is a mean to acquire or enhance the knowledge about a given phenomenon. An example is illustrated in Fig. 1.9, which is a plot of the tide phases relative to Greenwich in all the worlds oceans. This picture is a summary of a huge quantity of data about tides. Different colours are used to represent different phases of the tides. The blank curves are called *cotidal lines*, which pass through points which are at the same tide phase. Looking at the figure, we observe that the cotidal lines meet in some points, called *anphidromic points*, where tides are absent. This proves that if data are properly visualized, understanding and knowing new phenomena is quite immediate.

Most of the research in information visualization deals with non-structured data and aims, through visualization, at giving them a structure. However, many applications (such as the Web, software engineering, real systems and artificial intelligence) need to visualize structured data and this is usually achieved with a graph.

Graph Drawing mostly concentrates on optimizing algorithms to enforce some aesthetic constraints. Seldom, however, are tests carried out to evaluate the actual readability of the graph or only small graphs are involved in experiments [HMM00]. Information visualization typically deals with a great amount of data which results in very large graphs. In Section 1.2.5 we saw that some algorithms claim to be able to draw graphs with thousands of nodes. However, even if space were enough to contain thousand nodes, what kind of information

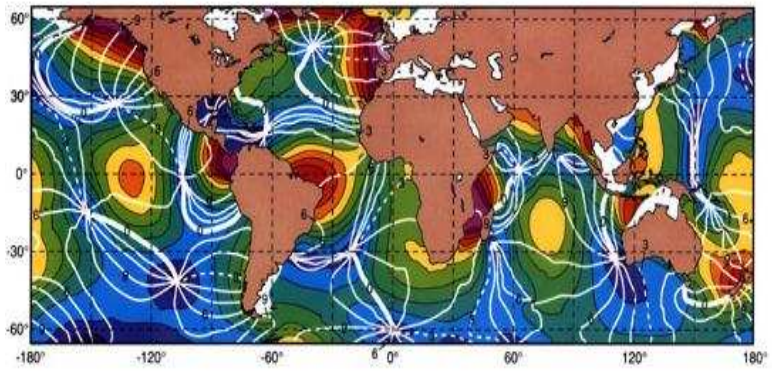


Figure 1.9: Cotidal chart. The blank lines represent the cotidal lines, which meet at the anphidromic points. Image taken from http://www.columbia.edu/itc/ldeo/v1011x-1/jcm/Topic2/Topic2.html.

do we hope to extract? It is clear that aesthetic rules improve the readability of a graph, but they are not enough.

In Information Visualization *interaction* and *navigation* are two key concepts. Depending on the context, a graph may or not be considered as a static entity. From a user's perspective, an underground map is a static entity. Conversely, the authorized staff's point of view is quite different. If a new station or line is added to the underground, it should be possible to modify the map. Moreover, each station may be associated with a set of information regarding its services, its maintenance state and so on; clicking on a node should display such information. This is a straightforward example of interaction. *Navigating* a graph means exploring the interactions which are described by the graph. Again, it clearly comes out that both interaction and navigation are cumbersome in large graphs.

Information filtering is a commonly used technique to reduce the amount of information to display. The goodness of the visualization depends on the quality of the filtering. A *spanning tree* is a trivial example of filtering on a graph; in fact, it contains all nodes of the graph, but only a small subset of its edges. Since it is a tree, it is also easier to draw. However, this solution is suitable only when not all the edges are equally significant. Another popular technique is *clustering*, which consists of grouping the nodes of the graph into classes called *clusters*. Vertices belonging to the same cluster may be visualized as a single node (usually referred to as *macro*), thus reducing the number of vertices to display. Moreover, a cluster can be further partitioned into subclusters, creating a hierarchy of clusters. Macros can be *expanded* or *collapsed* (for example with a mouse click) to show or hide the vertices it contains. Therefore, the graph still contains all information but they can be hidden when not needed. Clustering is definitely an effective solution for displaying large graphs. However, drawing a clustered graph is all but a straightforward

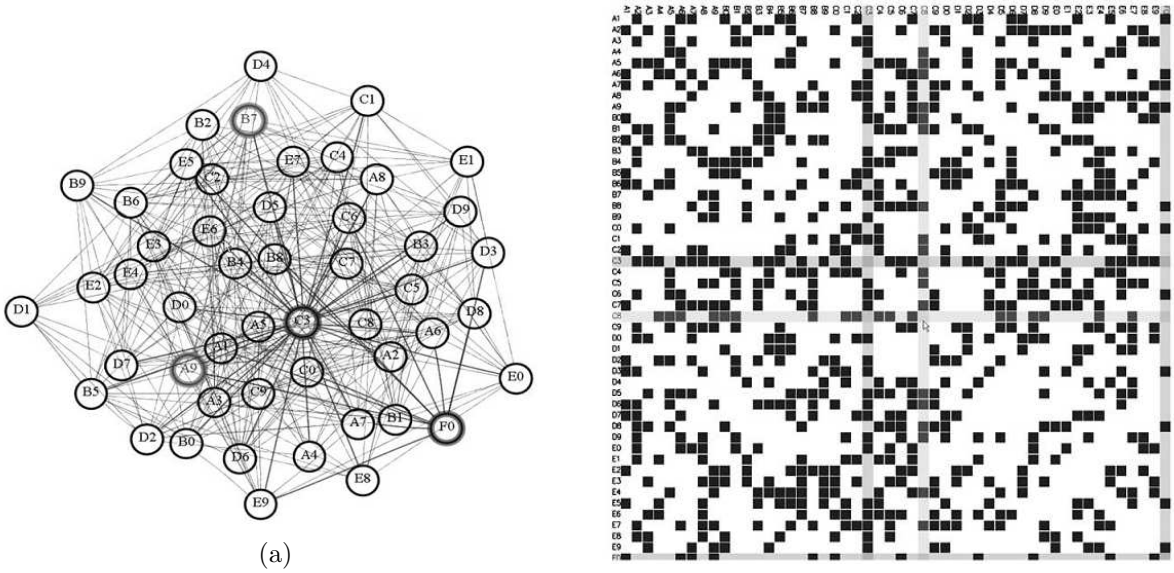
task. Moreover, critics claim that while navigating a cluster, users do not have a clear idea of the overall structure of the graph. In other words, they lose sight of the *context*. The *focus+context* techniques try to answer this need. *Fisheye distortion*, for example, displays the whole graph but enlarges only an area of interest, while showing other portions of the image with successively less details. The name derives from that this technique imitates the fisheye lens effect. Generally, focus+context techniques, such as fisheye distortion, are independent of the drawing algorithm and are applied only after the creation of the drawing. The advantage is that the algorithm performances are not undermined; however, applying a distortion to a drawing may alter its compliance with some aesthetic constraints. Only few techniques combine together the layout algorithm and the distortion; *hyperbolic view* is such an example [MB95].

Finally, *zoom* and *pan* are used to help navigation. Graphs are relatively simple drawings, from a purely geometric point of view. In fact, simple geometric entities are employed to draw nodes and edges. Therefore, zooming can be achieved by redrawing the part of the graph to enhance, rather than by zooming into the pixel image. As a result, aliasing problems can not occur. Two types of zoom are possible: geometric and semantic. In the first case, the geometric entities are enhanced; in the second case the content of the graph is changed. For example, while zooming in, the textual labels of some nodes are displayed and other hidden. Using zoom, pan and distortion introduces some problems which occur commonly in interactive environments. Since these issues are more related to computer graphics and are beyond the scope of this thesis, we skip them.

1.3.1 A Matrix-Based Representation

Few researches aim at evaluating the readability of a graph through experiments. Particularly interesting is the one described in [GFC05], which compares the traditional visualization of a graph as a collection of points and lines (the *node-link representation*) against a *matrix-based representation*. In a matrix-based representation, vertices are assigned a natural number in the interval $[1, n]$; the graph is visualized as a matrix such that a cell (i, j) is black if an edge links node i with node j, it is white otherwise (Fig. 1.10). This representation is particularly suitable for representing large graphs, as edge crossings never occur. In [BEW95], for example, it is used to show the load distribution of a telecommunication network. Another advantage is that vertices can be arbitrarily ordered to reflect their semantic meaning, whereas in a node-link drawing the node position is bounded by aesthetic constraints.

The experiments described in [GFC05] are, in our opinion, a wonderful example of how tests for readability of a drawing should be conducted. A group of participants was asked to perform seven different operations on a graph: estimating the number of nodes, estimating the number of edges, finding the most connected node, finding a node given its label, find-



(a) (b)

Figure 1.10: A graph with 50 nodes and 400 edges, visualized with (a) a node-link drawing and (b) a matrix-based representation. Figure taken from [GFC05].

ing an edge between two specified nodes and finding a path between two specified nodes. The experiment was carried out on randomly generated graphs. This is particularly important, as the readability of a graph improves if users are acquainted with the information associated with the graph. The results show that the matrix-based representation outperforms the node-link visualization on large graphs; with small graphs the performances are comparable.

1.4 Contributions

The aim of this thesis is to dust off a concept, known as *rectangular dualization*, which has been introduced in the early seventies. Basically, a *rectangular dual* of a graph G is a dissection of a rectangle into as many rectangles as the number of the nodes in G; each rectangle corresponds to one and only one node in G and two rectangles are adjacent if and only if the corresponding nodes are linked by an edge. Rectangular dualization proved to be successful in several domains, such as VLSI physical design and architecture, but in other domains (such as Graph Drawing) its expressive power has not been exploited as it should have been. Specific contributions of this thesis are:

1. A new algorithm for creating a rectangular dual of any planar graph is described

(Chapter 3). A planar graph must comply with some restrictive conditions to admit a rectangular dual. We believe that this fact, along with the lack of efficient algorithms, is the main reason why rectangular dualization did not gain the success he would have deserved. An implementation of the new algorithm is available in our software tool called *OcORD*.

2. We give simple algorithms for creating orthogonal drawings of plane graphs from their rectangular dual (Chapter 5). This is probabily the most original part of the thesis, as rectangular dualization has never been used for drawing graphs. We also created a new drawing style, called *bus-mode drawing*, which extends the benefits of an orthogonal drawing to graphs with unbounded degree. We do not present any experimental results assessing the drawing algorithms; we reserve to do it as a future work.

3. We apply rectangular dualization to the generation of 3D Virtual Worlds based on the metaphor of a 3D Electronic Institution (Chapter 5). 3D Electronic Institutions are a new method to give a full specification of multi-agent system with a high degree of interaction among participants. This technology was developed by the E-Market Research Group, which includes researchers from the University of Technology, Sydney, the Institute of Artificial Intelligence located in Barcelona, Spain and the e-Commerce Competence Centre (Vienna, Austria). We have an active collaboration with this group. Usually a 3D Electronic Institution is seen as a 3D virtual building, each room being devoted to a specific task. The building is represented as a plane graph (called *Performative Structure*), where nodes represent rooms and edges the adjacency relationship between them. The building is created from its floorplan, which is a rectangular dual of the Performative Structure.

4. We introduce *c-rectangular dualization* (Chapter 6) as the extension of rectangular dualization to hierarchically clustered graphs (HCGs). A c-rectangular dual could be itself a good representation of a HCG; moreover, orthogonal drawings of the HCG can be obtained by slightlty modifying the algorithms presented in Chapter 5. Creating a c-rectangular dual of a given HCG is all but an easy task; admissibility conditions are much stronger than in the case of a plain rectangular dual and enforcing then may requires the addition of many new vertices to the input graph.

5. We developed a software tool, named OcORD, which implements the main algorithms described in the thesis. The main features of OcORD are described in Chapter 7. Our aim is to turn OcORD into a graph drawing tool.

Chapter 2

Preliminaries

This chapter introduces basic graph theory, based on two well-known textbooks [Bol02, Die05]. In order to make easier the presentation of the algorithms, a data structure for representing a plane graph is described in details. Finally, special sections are devoted to inner-triangulated graphs and hierarchically clustered graphs, as they play a central role in the thesis.

2.1 Graphs

Definition 2.1. *A* graph *is a pair $G = (V, E)$ of sets such that $E \subseteq V^2$. The elements of V are called* vertices *(or* nodes *or* points*) of G and the elements of E are its* edges *(or* arcs *or* links*).*

The vertex set of a graph G is referred to as $V(G)$ and its edge set as $E(G)$. An edge $e = (u, v)$ is said to be *incident* with vertices u and v, and vertices u and v are the *endpoints* of e. e is also said to *join* or *connect* u and v. Two nodes u and v are *adjacent*, or *neighbours* if $(u, v) \in E(G)$. Usually, $N(v)$ refers to the set of the nodes adjacent to v. The number of the nodes $|V(G)|$ is the *order* of G, the number of edges $|E(G)|$ is its *size*.

Definition 2.2. *Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. If $V' \subseteq V$ and $E' \subseteq E$, then G' is a* subgraph *of G (G* contains *G'), denoted as $G' \subseteq G$.*

If $G' \subseteq G$ contains all the edges $(u, v) \in E$ with $u, v \in V'$, then G' is a subgraph of G *induced* by V' and is denoted as $G[V']$. If $U \subseteq V$, the graph $G - U$ denotes the graph $G[V \backslash U]$.

Definition 2.3. *A graph $G = (V, E)$ is called* r-partite *if V admits a partition into r classes such that every edge has its endpoints in different classes: vertices in the same partition class must not be adjacent. A 2-partite graph is usually referred to as* bipartite *graph.*

Definition 2.4. *A* directed graph *(or* digraph*) is a graph $G = (V, E)$ such that E is a set of ordered pairs of vertices.*

If $e = (u, v)$ is an edge of a directed graph, u is called *initial vertex* and v *terminal vertex* of e. If $(u, v) \in E$ and $(v, u) \notin E$, u is adjacent to v but v is not.

Definition 2.5. *A* multigraph *is a graph in which multiple edges and loops are allowed. A* multiple edge *(or* multiedge*) e is composed of two or more edges (called the* occurrences *of e) connecting the same two vertices. A* loop *is an edge connecting a vertex to itself.*

The *multiplicity* of a multiedge is the number of its occurrences; edges with multiplicity 1 are called *simple edges*. A *simple graph* has only simple edges. Henceforth, the term *graph* refers uniquely to a simple graph and the term *edge* refers to a simple edge, if not otherwise specified.

Definition 2.6. *The* degree *$d(v)$ of a vertex $v \in V(G)$ is the number $|E(v)|$ of edges incident with v.*

A vertex of degree 0 is *isolated*. If a graph contains only one isolated vertex is called a *singleton graph*. The number $\delta(G) = min\{d(v)|v \in V(G)\}$ is the *minimum degree* of G and $\Delta(G) = max\{d(v)|v \in V(G)\}$ is the *maximum degree* of G. If all the vertices of G have the same degree k, G is called $k-$*regular* or simply *regular*. A $3-$regular graph is also called *cubic* and a graph with maximum degree 3 is called *subcubic*.

A *complete graph* (or *clique*) is a graph in which each pair of nodes is connected by an edge. A $n-$clique is a clique with n vertices and is denoted as K_n. A complete bipartite graph $G = (V_1, V_2, E)$ is denoted as $K_{n,m}$, where $n = |V_1|$ and $m = |V_2|$.

2.2 Paths and Cycles

Definition 2.7. *A* path *is a graph $P = (V, E)$ of the form*

$$V = \{x_i | 1 \leq i \leq k\} \quad E = \{(x_i, x_{i+1}) | 1 \leq i < k\}$$

The vertices x_1 and x_k are the endvertices *of P and the number of the edges of P is the* length *of P. We also say that P is the path from x_1 to x_k (or between x_1 and x_k).*

Usually, a path P is denoted as the ordered sequence $(x_1, ..., x_k)$ of the vertices composing it, from the *initial vertex* x_1 to the *terminal vertex* x_k. Alternatively, the path can be also denoted by the ordered sequence $(e_1, ..., e_k)$ of the edges composing it. The vertices of the path (excluding the initial and the terminal one) are called *inner vertices* of P. Two paths are *independent* if they do not share any inner vertex. The *distance* $d_G(u, v)$ in G of two vertices u and v is the length of the shortest path from u to v.

Definition 2.8. *A* cycle *is a path such that the initial and the terminal vertex coincide.*

A cycle Π is usually denoted by its cyclic sequence of vertices $(x_1, ..., x_k, x_1)$. When no ambiguity arises with paths, the second occurrence of x_1 is omitted. Alternatively, a cycle can be also denoted as the cyclic sequence $(e_1, ..., e_k)$ of the edges composing it. The *length* of a cycle is indifferently the number of its vertices or edges. A $k-$*cycle* is a cycle of length k. If $k = 3$, the cycle is also called *triangle* and if $k = 4$ *quadrangle*. A multigraph may have 2-cycles (induced by multiedges) and 1-cycles (induced by loops). A *chord* of a cycle Π is an edge which joins two vertices of Π but is not an edge of Π; a *chordless* cycle is one with no chords.

Definition 2.9. *A* tree *is an* acyclic *graph, that is a graph containing no cycles.*

The vertices of degree 1 in a tree are its *leaves*, the other are called *non-terminal nodes*. Often is a vertex r of a tree elected as *root*; in this case the tree is said *rooted* at r. The vertices at distance k from r form the kth *level* of the tree; a node at the kth level is also referred to as $k-$*node*; the maximum level is the *height* of the tree. Let v be a $k-$node in a tree T rooted at r. If $v \neq r$, there is exactly one edge (v, u) such that u is a $(k-1)-$node; u is called the *parent* of v. If v is a non-terminal node, there is at least one edge (v, w) such that w is a $(k+1)-$node; w is a *child* of v. r is the only vertex with no parent and leaves are the only nodes with no children. A node w is a *descendant* of v (and v is an *ancestor* of w) if either w is a child of v or there is a path Π from v to w whose inner vertices are descendant of v. The *subtree* of T rooted at v is the tree $T[v]$ containing v and all the descendants of v.

2.3 Connectivity

Definition 2.10. *A graph G is* connected *if any two of its vertices are linked by a path in G. Otherwise the graph is* disconnected.

A *connected component* of G is a maximal connected subgraph of G. Here, maximal refers to the fact that it is not possible to add an edge or vertex to the subgraph while keeping it connected. A set $S \subseteq V(G)$ *separates* G if $G - S$ is disconnected; S is called a *separator*. If

S induces a cycle in G, S is said a *separating cycle*. If the removal of a vertex v disconnects G, v is a *cutvertex*; if the removal of an edge e disconnects G, e is a *bridge*. Notice that bridges are edges which belong to no cycle. A graph with no bridges is called *bridgeless*.

Definition 2.11. *A graph G is* $k-$connected*, $k \in \mathbb{N}$, if $|V(G)| > k$ and $G-X$ is connected for every set $X \subseteq V$ with $|X| < k$. A 2-connected graph is also called* biconnected *and a 3-connected graph is called* triconnected.

The greatest integer k such that G is $k-$connected is the *connectivity* $\kappa(G)$ of G. In a 1-connected graph, the term *block* (or *biconnected component*) refers to a maximal biconnected subgraph.

Definition 2.12. *A graph G is* l-edge-connected*, $l \in \mathbb{N}$, if $|V(G)| > 1$ and $G - F$ is connected for every set $F \subseteq E$ with $|F| < l$.*

The greatest integer l such that G is $l-$edge-connected is called *edge-connectivity* $\lambda(G)$ of G.

Lemma 2.1. $\kappa(G) \leq \lambda(G) \leq \delta(G)$

Proof. The reader is referred to [Die05]. □

Lemma 2.2. *In a biconnected graph all edges belong to at least one cycle. Alternatively, any biconnected graph is bridgeless.*

Proof. It is a trivial consequence of Lemma 2.1. □

2.4 Drawing of a Graph

A graph is an *abstract* (or *combinatorial*) entity. A geometric representation in the plane (also called a *drawing* or *embedding*) of a graph is usually achieved by drawing the vertices as points and edges as continuous curves. A drawing of a graph is called *plane* if no two edges meet in a point other than a common endpoint. If a graph has at least one plane drawing is called *planar*. A plane embedding is described by specifying the exterior vertices and by sorting $N(v)$ in clockwise or counterclockwise order around v, for every node v. Henceforth, we will use the terms "graph", "vertex" and "edge" to mean both the combinatorial entities and their drawings. Besides the vertex and edge set, a plane graph G has also a set $F(G)$ of faces. A *face* of G is a region of the plane bounded by edges and such that two points in it can be connected by a continuous curve that do not cross any edge or meet any vertex of G. The *frontier* of a face is a simple cycle in G. A $k-$face,

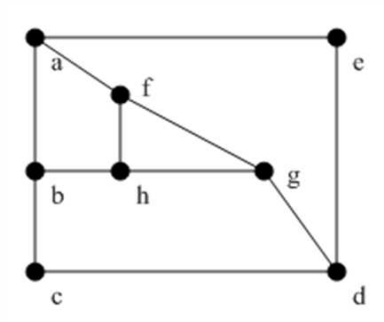


Figure 2.1: Graph with four faces: $f_1 = (a, f, g, d, e)$, $f_2 = (a, b, h, f)$, $f_3 = (f, h, g)$ and $f_4 = (b, c, d, g, h)$.

$k \in \mathbb{N}$, is a face whose frontier is a $k-$cycle. There is exactly one unbounded face which has no frontier and is called the *exterior face* of G. The other faces are called *inner faces* and are denoted by specifying their unique frontier (Fig. 2.1).

A vertex v (or an edge e) is said *incident* with a face f if v (e) belongs to the frontier of f. Any vertex (edge) incident with the exterior face is called *exterior vertex* (*exterior edge*); the other vertices (edges) are called *inner vertices* (*inner edges*). Every edge lying on at least a cycle is incident with exactly two faces; any edge lying on no cycle is incident with exactly one face. A cycle of G whose nodes and edges are exterior with the exterior face is called *exterior cycle*.

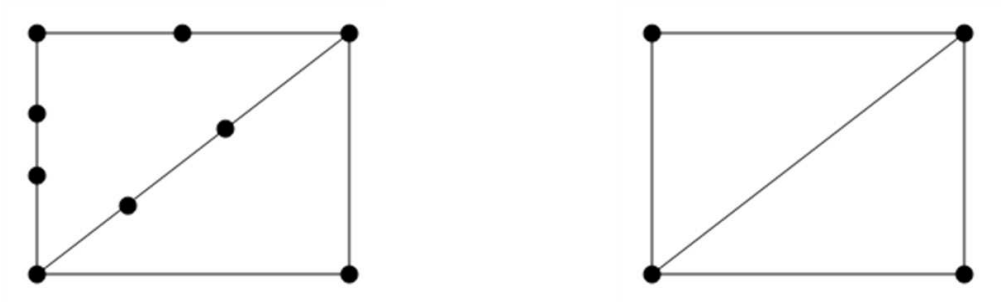
Definition 2.13. *A graph is called* outerplanar *if it can be embedded in the plane in such a way that all vertices are exterior. A graph G is $r-$outerplanar if, for $r = 1$, G is outerplanar or, for $r > 1$, G has a plane embedding such that if all vertices on the exterior face are deleted, the connected components of the remaining graph are all at most $(r-1)-$outerplanar.*

Henceforth, we will denote as n, m and l the number of nodes, edges and faces respectively. In a plane graph these three numbers are related by Euler's Formula.

$$n - m + l = 2$$

This easily implies that $m \leq 3n-6$ and $l \leq 2n-4$. Thus, $m \in O(n)$ and $l \in O(n)$. Notice that the relation $m \leq 3n-6$ is only a necessary condition for planarity. Necessary and sufficient conditions for planarity were first given by Kuratowski, in terms of *subdivisions*.

Definition 2.14. *Let X be a graph. A* subdivision *(Fig. 2.2) of X is the graph G obtained from X replacing the edges with independent paths between their ends, so that none of these paths has an inner vertex on another path or in X.*



(a) (b)

Figure 2.2: (a) A graph X. (b) A subdivision of X.

Theorem 2.1 (Kuratowski, 1930)**.** *A graph G is planar if and only if it contains neither a K^5 nor a $K^{3,3}$ nor a subdivision of both.*

A cycle Π in a plane graph divides the plane into an "inside" and an "outside" region; all vertices belonging to the inside region are said to be *enclosed* in the cycle. The set of the vertices enclosed in Π is denoted as $I(\Pi)$. If $I(\Pi) \neq \emptyset$, then Π is a *complex cycle*. The following fact is trivially true.

Lemma 2.3. *A complex cycle is a separating cycle.*

The converse is not true, as shown in Fig. 2.3.

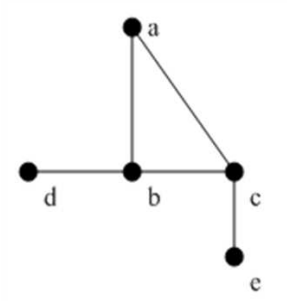


Figure 2.3: Cycle (a, b, c) is separating but not complex.

The same considerations hold for plane directed graphs with few differences. Each edge e is drawn as a continuous curve terminated by an arrow cap directed from the initial to the terminal vertex. Moreover, walking along e from the initial endpoint to the terminal endpoint, the face which is on the left is called the *left face* of e and the other *right face* of e.

2.4.1 Duality

Definition 2.15. *Let G be a plane graph. The* geometric dual *of G is the graph G^* obtained as follows:*

- *A vertex v_f is placed inside each face f of G.*
- *For every edge e incident with two faces f and g, an edge e^* (called the* dual edge *of e), crossing e, is added between vertices v_f and v_g.*
- *If e is incident with one face f, a loop, crossing e, is attached to v_f.*

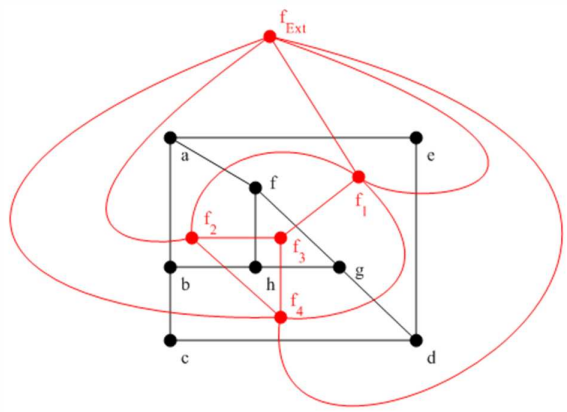


Figure 2.4: The graph of Fig. 2.1 (in black) and its geometric dual (in red). (f_1, f_{Ext}) and (f_4, f_{Ext}) are multiedges.

As shown in Fig. 2.4, G^* may be a multigraph. Summing up, each vertex of G^* corresponds to a distinct face of G, every edge of G^* corresponds to a distinct edge of G and every face of G^* corresponds to a distinct vertex of G. Consequently, the geometric dual of G^* is G.

Lemma 2.4. *The geometric dual of a plane graph is a plane connected graph.*

Proof. The proof can be found in [Har94]. □

Lemma 2.5. *The geometric dual of a biconnected plane graph is a biconnected plane graph.*

Proof. Suppose by contradiction that G^* is not biconnected. Let v be a cutvertex of G^* and suppose, without loss of generality, that removing v separates G^* into two connected components, whose vertex sets are A and B, $A \cap B = \emptyset$. Let $G_1^* = G^*[A \cup \{v\}]$ and $G_2^* = G^*[B \cup \{v\}]$. Clearly, $E(G_1^*) \cap E(G_2^*) = \emptyset$ and the intersection between $F(G_1^*)$ and $F(G_2^*)$ only contains the exterior face f_{Ext} of G_1^* and G_2^*. Consequently, in the geometric dual of G^* (which is G), $v_{f_{Ext}}$ is a cutvertex, which contradicts the biconnectivity of G. □

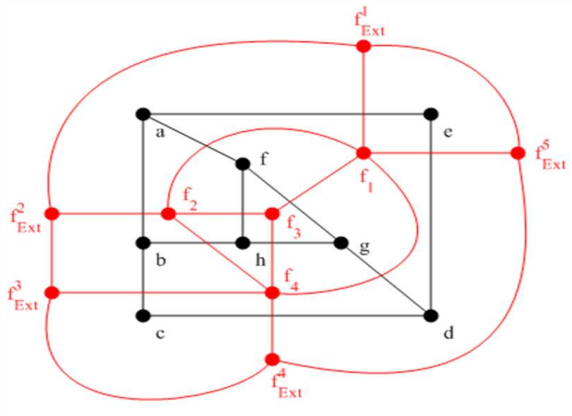


Figure 2.5: The graph of Fig. 2.1 (in black) and its extended geometric dual (in red).

Definition 2.16. *Let G be a plane biconnected graph with an exterior cycle $C = (e_1, ..., e_k)$, $e_i \in E(G)$, $1 \leq i \leq k$. The* extended geometric dual *(Fig. 2.5) of G is the graph G^*_{ext} obtained from G^* as follows:*

- *The vertex v_{f^0}, corresponding to the exterior face f^0 of G, is removed along with its incident edges.*
- *Let f^i denote the inner face incident with edge e_i, for $i = 1, ..., k$. A new vertex $v^i_{f^0}$ and a new edge $(v^i_{f^0}, v_{f^i})$ are added such that $v^i_{f^0}$ is in the exterior face of G and $(v^i_{f^0}, v_{f^i})$ is a straight edge crossing e_i.*
- *Edges $(v^1_{f^0}, v^2_{f^0})$, $(v^2_{f^0}, v^3_{f^0})$, ..., $(v^k_{f^0}, v^1_{f^0})$ are added in such a way that they do not cross any edge of G.*

The vertices of G^*_{ext} added in the exterior face of G form the exterior cycle of G^*_{ext}.

Two definitions of geometric dual are given for directed graphs.

Definition 2.17. *Let G be a plane directed graph. The* left-oriented geometric dual *of G is the geometric dual G^* of G such that each edge e^* is directed from v_l to v_r, where l and r are respectively the left and the right face incident with e in G.*

Definition 2.18. *Let G be a plane directed graph. The* right-oriented geometric dual *of G is the geometric dual G^* of G such that each edge e^* is directed from v_r to v_l, where l and r are respectively the left and the right face incident with e in G.*

2.5 Edge Contraction.

Edge contraction is a fundamental operation in this thesis. Let $e = (u, v)$ be an edge of a plane graph G. G/e is the graph obtained from G by contracting e into a new vertex v_e (Figure 2.6) which becomes adjacent to all the neighbours of u and v . Formally:

$$V(G/e) = (V(G)\backslash\{u, v\}) \cup \{v_e\}$$

and

$$E(G/e) = \{(x, y) \in E(G) | \{x, y\} \cap \{u, v\} = \emptyset\} \cup$$
$$\{(v_e, w) | (u, w) \in E(G)\backslash\{e\} \quad or \quad (v, w) \in E(G)\backslash\{e\}\}$$

e is called *contracting edge*. The converse operation is called *vertex split*.

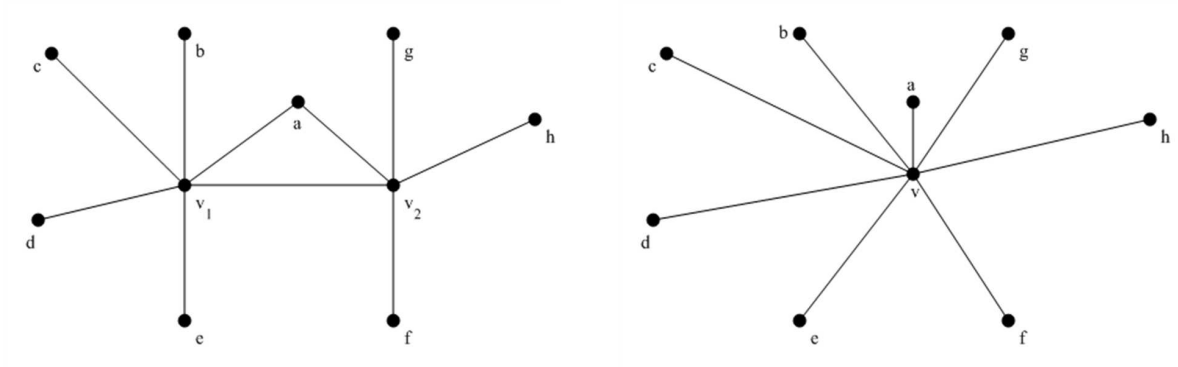


Figure 2.6: (a) A plane graph G. (b) G after contracting edge (v_1, v_2). Edges (v_1, a) and (v_2, a) are merged into a single edge (v, a).

Lemma 2.6. *Let e be a contracting edge and G be a plane graph. e is the edge of a $k-$cycle Π in G, $k > 0$, if and only if Π/e is a $(k-1)-$cycle in G/e.*

Proof. The contraction of an edge of a k-cycle decreases the length of the cycle by 1. Conversely, splitting a vertex of a $(k-1)$-cycle creates a k-cycle. □

2.6 Inner-Triangulated Graphs

A plane *inner-triangulated graph* (PTG) is a plane graph with each inner face being a triangle. A PTG with also the exterior cycle being a triangle is a *maximal planar graph*, in the sense that no (simple) edge can be added without introducing a crossing. PTGs should not be confused with *triangulated graphs* (also called *chordal graphs*), in which every cycle having length greater than three has a chord. Figure 2.7 makes this difference clearer.

(a) (b)

Figure 2.7: (a) A chordal graph which is not a PTG. (b) A PTG which is not chordal.

Lemma 2.7. *Let G be a biconnected PTG. The extended geometric dual G^*_{ext} of G is a plane cubic bridgeless graph.*

Proof. Since G is plane and biconnected (bridgeless by Lemma 2.2), G^* is plane bridgeless by Lemma 2.5. By construction, G^*_{ext} is still plane and bridgeless. The degree of each inner vertex of G^*_{ext} is 3 because each inner face of G is a triangle; each exterior vertex of G^*_{ext} is connected to two exterior vertices and one inner vertex by construction. Thus, G^*_{ext} is a cubic graph. □

Lemma 2.8. *Let G be a PTG and $x, y, v \in V(G)$. If x and y are inner vertices and consecutive in $N(v)$, then $(x, y) \in E(G)$.*

Proof. This is immediate, as each inner face of G is a triangle. □

Lemma 2.9. *Let e be an edge of a multi-PTG[1] G with no loops. G/e is a PTG with no loops if and only if e is a simple edge.*

Proof. If e were a multiedge, G/e would contain a loop. This establishes necessity.

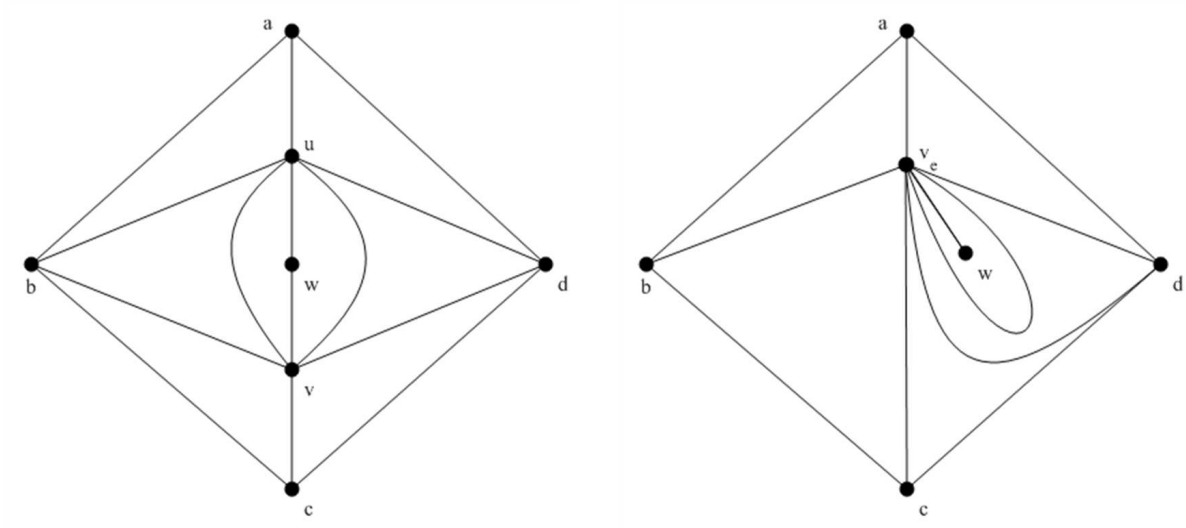
[1] A *multi-PTG* is a PTG containing multiedges and, possibly, loops.

Let u and v be the endpoints of e and let v_e be the vertex which replaces u and v in G/e. The faces incident with both u and v are precisely those incident with e, which are destroyed while contracting e. The faces which are not incident with neither u nor v are not concerned by the contraction. The frontier of faces (x, y, u), $x, y \in V(G)$, is a $3-$cycle (x, y, v_e) in G/e; similarly, the faces incident with v are $3-$faces in G/e. This proves sufficiency.

□

If the contracting edge $e = (u, v)$ is a multiedge with multiplicity $m > 1$, there are up to $2m$ faces which are incident with both u and v. The faces incident with the occurrence of e being contracted are destroyed. The other faces (Fig. 2.8) may become "degenerate" $2-$ or $3-$faces (that is, faces whose frontier contains a loop). The faces which are incident with either u or v are still $3-$faces in G/e; the remaining faces are not concerned by the contraction of e. Thus, the only faces which are not $3-$faces are those incident with the loops.

Definition 2.19. *A* degenerate *PTG is a plane multigraph with loops with all inner faces being $3-$faces, except for the faces incident with loops.*



(a) (b)

Figure 2.8: (a) A PTG G with one multiedge $e = (u, v)$. (b) G after contracting e. The faces incident with loop (v_e, v_e) are the only non-triangular faces.

Lemma 2.10. *Let G be a degenerate PTG and $e \in E(G)$ be not a loop. G/e is a degenerate PTG.*

Proof. The contraction of a simple edge $e = (u, v)$ does not alter the length of the faces incident with only u or only v and the faces which are not incident neither with u nor with v. If e is simple, at most two inner faces are incident with e. Let f be such a face. If f is a 3−face, it is destroyed by the contraction of e. If f is not a 3−face, it is incident with a loop. Consequently, f is incident with this loop also in G/e. Suppose now that e is a multiedge. There are up to $2 \cdot m$ faces incident with both u and v. The considerations done before hold for the faces incident with the occurrence of e being contracted. The faces incident with the other occurrences of e clearly are not 3−faces in G/e; however, any occurrence of e different than the contracting one becomes a loop in G/e. In any case, non-triangular faces are incident with loops. □

Lemma 2.11. *Let G be a PTG and Π a chordless cycle in G. The subgraph G_Π induced by the vertices of Π and those belonging to $I(\Pi)$ is a PTG.*

Proof. Π is the exterior cycle of G_Π and the vertices in $I(\Pi)$ are its inner vertices. As G is plane, the vertices adjacent to any vertex $v \in I(\Pi)$ are either vertices of Π or vertices of $I(\Pi)$. Consequently, the faces incident with v in G_Π are those incident with v in G. □

2.7 Hierarchically Clustered Graphs

Clustering refers to partitioning a set of objects into groups called *clusters*. In a graph, clustering can be applied to either vertices (*vertex clustering*) or edges (*edge clustering*). Although there are some interesting work on edge clustering [Pau93, CZQ+08, QZW07], vertex clustering is far more popular. In this thesis, we will only consider vertex clustering, henceforth referred to as simply *clustering*.

Definition 2.20. *A* k-partition *of a set C is a family of subsets $\mathscr{C} = (C_1, ..., C_k)$ of C such that:*

1. $\bigcup_{i=1}^{k} C_i = C$
2. $C_i \cap C_j = \emptyset$ *for* $i \neq j$

C_i are called parts, blocks *or* cells.

Definition 2.21. *A* clustered graph *(CG) (Fig. 2.9) is a pair $\mathscr{G} = (G, \mathscr{C})$, where G is called the* underlying graph *and $\mathscr{C} = (C_i | 1 \leq i \leq k)$ is a k-partition, $k > 0$, on the vertex set V. Each element in $\mathscr{C}$ is called* cluster *and k is the number of clusters.*

$G[C_i]$ is the subgraph of G induced by the vertices in C_i, $1 \leq i \leq k$, also referred to as cluster, with an abuse of notation.

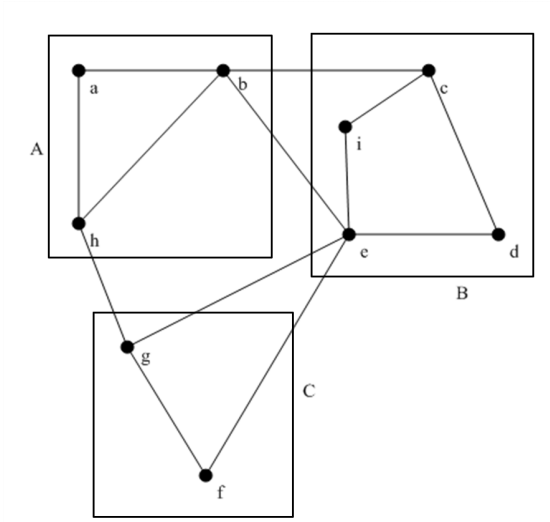


Figure 2.9: A clustered graph with three clusters.

A painstaking reader may have noticed that Definition 2.21 does not account for *subclustering*, which means hierarchically grouping the vertices of a cluster into *subclusters*. If subclusters are allowed, $\mathscr{G}$ is called *hierarchically clustered graph*.

Definition 2.22. *A* hierarchically clustered graph *(HCG) (Fig. 2.10)* $\mathscr{G} = (G, T)$ *consists of a graph* G *(called* underlying graph*) and a rooted tree* T *(called* hierarchy tree*) such that:*

1. *The root* ρ *of* T *represents the cluster containing all vertices of* G*.*
2. *The leaves of* T *are exactly the vertices of* V*.*
3. *Each non-terminal node* ν *of* T *represents a cluster whose vertices* $V(\nu)$ *are the leaves of the subtree rooted at* ν*.*

For each node ν of T, $G(\nu)$ denotes graph $G[V(\nu)]$, which, with an abuse of notation, is also called *cluster*. The distance of a node ν from ρ is the *level* of the cluster; the height of T is also called *depth* (denoted as $D(\mathscr{G})$) of $\mathscr{G}$. If $D(\mathscr{G}) = 1$, then $\mathscr{G}$ has no clusters and is called *plain graph*. In a HCG the intersection of two clusters A and B is not empty if and only if A is a subcluster of B or vice versa. In other words, either two clusters are *independent* or they totally overlap. HCGs, where clusters can partially overlap, are called *compound graphs*. We will not consider them.

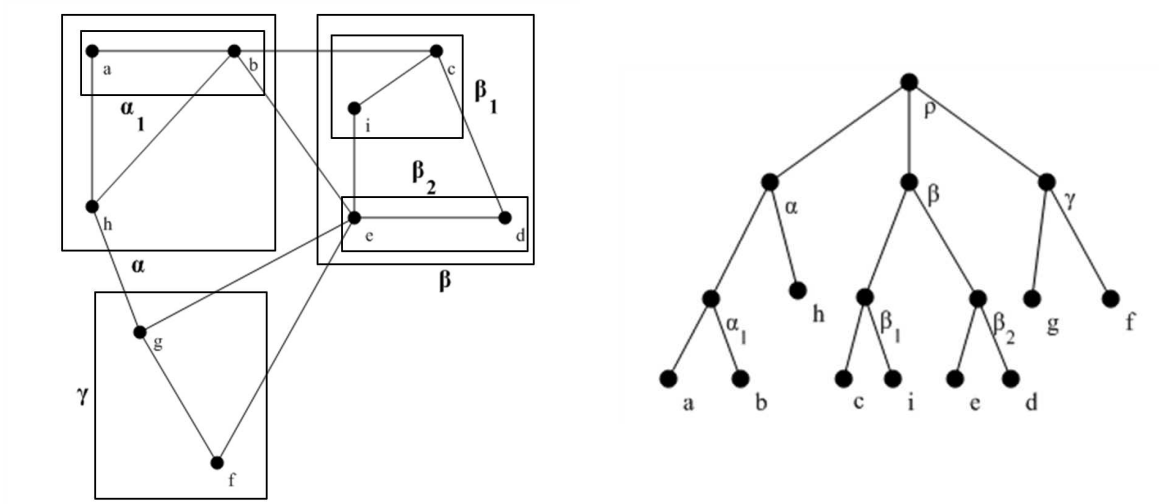


Figure 2.10: An HCG with its hierarchy tree.

Clearly, Definition 2.22 also includes Definition 2.21. The need of having two separate definitions comes out from the complexity of studying properties and algorithms on HCGs. Often is it easier to study properties or algorithms on CGs before extending them to HCGs. In Chapter 6 we will follow exactly this approach.

2.7.1 Embedding a Hierarchically Clustered Graph

The drawing (or embedding) of a HCG $\mathscr{G} = (G, T)$ is composed of the drawing of its underlying graph G and the drawing of the clusters. Usually, a cluster $\nu \in T$ is represented as a simple closed region R such that [FCE95a]:

1. The drawing of $G(\nu)$ is completely contained in the interior of R.
2. The regions of all subclusters of ν are completely contained in the interior of R.
3. The region of all other clusters are completely out of R (no partial overlaps are allowed).
4. If an edge links two vertices of $V(\nu)$, then the drawing of e is completely contained in R.

A cluster is *invalid* if its embedding does not comply with the previous conditions.

Lemma 2.12. *Let G be a plane graph and let Π be a cycle enclosing at least one node. If a cluster contains all nodes in Π, then it also contains all nodes in $I(\Pi)$.*

Proof. The lemma derives from that there is no way to draw a simple closed region containing the vertices of Π without containing the vertices of $I(\Pi)$ (Fig. 2.11). □

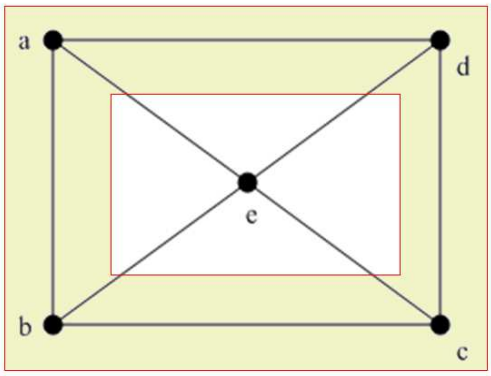


Figure 2.11: The yellow region represents a cluster C containing a, b, c and d. This is not a simple region, therefore C is invalid.

An *intercluster* edge in $\mathscr{G}$ links two vertices belonging to different clusters; the endvertices of an *intracluster* edge belong to the same cluster. An *inner vertex* is a vertex whose incident edges are all intracluster; a *boundary vertex* is incident with at least one intercluster edge. Intercluster edges are the only allowed to cross the boundary of a region; if they do more than once, there is an *edge-region crossing* (Fig. 2.12).

We can now extend the concepts of planarity and connectivity to HCGs [Fen97].

Definition 2.23. *A HCG $\mathscr{G} = (G, T)$ is* c-planar *if it has a drawing in the plane such that there are no edge crossings or edge-region crossings.*

Clearly, if $\mathscr{G}$ is c-planar, then G is planar; the converse is not true.

Definition 2.24. *A HCG $\mathscr{G} = (G, T)$ is* c-connected *if G is connected and each cluster induces a connected subgraph of G.*

In [FCE95b] the following characterization for c-planar graphs is given.

Theorem 2.2 (Eades, Feng)**.** *A c-connected graph $\mathscr{G} = (G, T)$ is c-planar if and only if G is planar and there exists a plane embedding $\mathscr{D}$ of G, such that for each node ν of T, all the vertices and edges of $G - G(\nu)$ are in the exterior face of the drawing of $G(\nu)$.*

Theorem 2.2 is used to describe a linear-time algorithm checking whether a given HCG is c-planar [FCE95b].

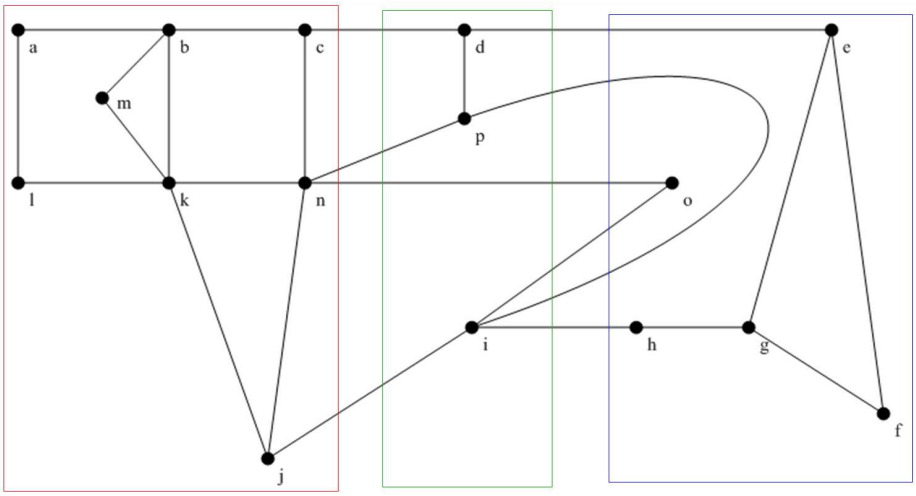


Figure 2.12: Drawing of a clustered graph with two edge-region crossings. (i, p) crosses the blue region and so does (n, o) with the green region.

2.7.2 Quotient Graph

The *quotient graph* (Fig. 2.13) is a useful representation of a CG.

Definition 2.25. *The* quotient graph *of a CG* $\mathscr{G} = (G, \mathscr{C})$, $\mathscr{C} = (C_i | 1 \leq i \leq k)$, *is the graph* Q *defined as follows:*

1. $V(Q)$ *contains a vertex* c_i *for each cluster* C_i. c_i *is usually referred to as* macro-vertex *or simply* macro.
2. *For each* $c_i, c_j \in V(Q)$, $1 \leq i \neq j \leq k$, $(c_i, c_j) \in E(Q)$ *if and only if there is* $v \in C_i$ *and* $w \in C_j$ *such that* $(v, w) \in E(G)$.

Let $\mathscr{G} = (G, \mathscr{C})$ be a plane c-connected CG with k valid clusters $C_1, ..., C_k$. We assume that G is a (multi) graph with no loops. Q can be obtained from $\mathscr{G}$ by recursively contracting all intracluster edges. The order in which edges are contracted does not matter. We can therefore assume that the intracluster edges of C_i are contracted only after contracting all the intracluster edges of C_{i-1}. Let $(Q^i | 1 \leq i \leq k)$ be the sequence of graphs defined as follows: $Q^0 \equiv G$ and Q^i is obtained from Q^{i-1} contracting all intracluster edges of C^i. Clearly, $Q^k \equiv Q$. In Q^i, the vertices of clusters $C_1, ..., C_i$ are replaced by macro-vertices $c_1, ..., c_i$ respectively.

Lemma 2.13. Q^i, $1 \leq i \leq k$, *is a plane multigraph such that every* $1-$*cycle is a* $1-$*face.*

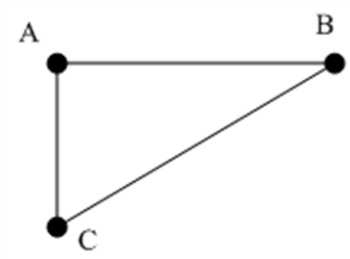


Figure 2.13: The quotient graph of the graph shown in Fig. 2.9.

Proof. By induction on i. The case $i = 0$ is trivial. By induction, Q^{m-1}, $m > 1$, is a plane multigraph with any loop being a $1-$face. Suppose that Π is a $1-$cycle which is not a $1-$face in Q^m. Π encloses at least one node, and no node enclosed in Π belongs to C^m. By Lemma 2.6, Π is obtained from the contraction of an intracluster edge of C^m lying on a $2-$cycle Σ enclosing the same vertices enclosed in Π. By Lemma 2.12, C^m is an invalid cluster. This contradiction completes the proof.

□

Any loop in Q^i can be only incident with macros $c_1, ..., c_i$. Figure 2.14 illustrates the effects of contracting the intracluster edges of a cluster. It is immediate to verify that contracting the edges in a different order, loop (c_1, c_1) does not come out. This means that the presence of $1-$faces in the quotient graph only depends on the order in which the edges are contracted. Therefore, they can be deleted, removing all loops. Thus, the following result holds.

Lemma 2.14. *Q^i, $1 \leq i \leq k$, is a plane multigraph with no loops.*

Proof. By construction and by Lemma 2.13. □

Corollary 2.1. *The quotient graph of a c-planar c-connected CG is a multigraph with no loops.*

2.8 Data Structure

Two data structures are traditionally used for graphs: adjacency matrices and adjacency lists. An *adjacency matrix* M for a graph G has as many rows and columns as the number n of vertices. Each cell $M[u, v]$, $u, v \in V(G)$, contains a 1 if $(u, v) \in E(G)$ and a 0 otherwise. Testing whether there is an edge between two given vertices is done in $O(1)$ time; however,

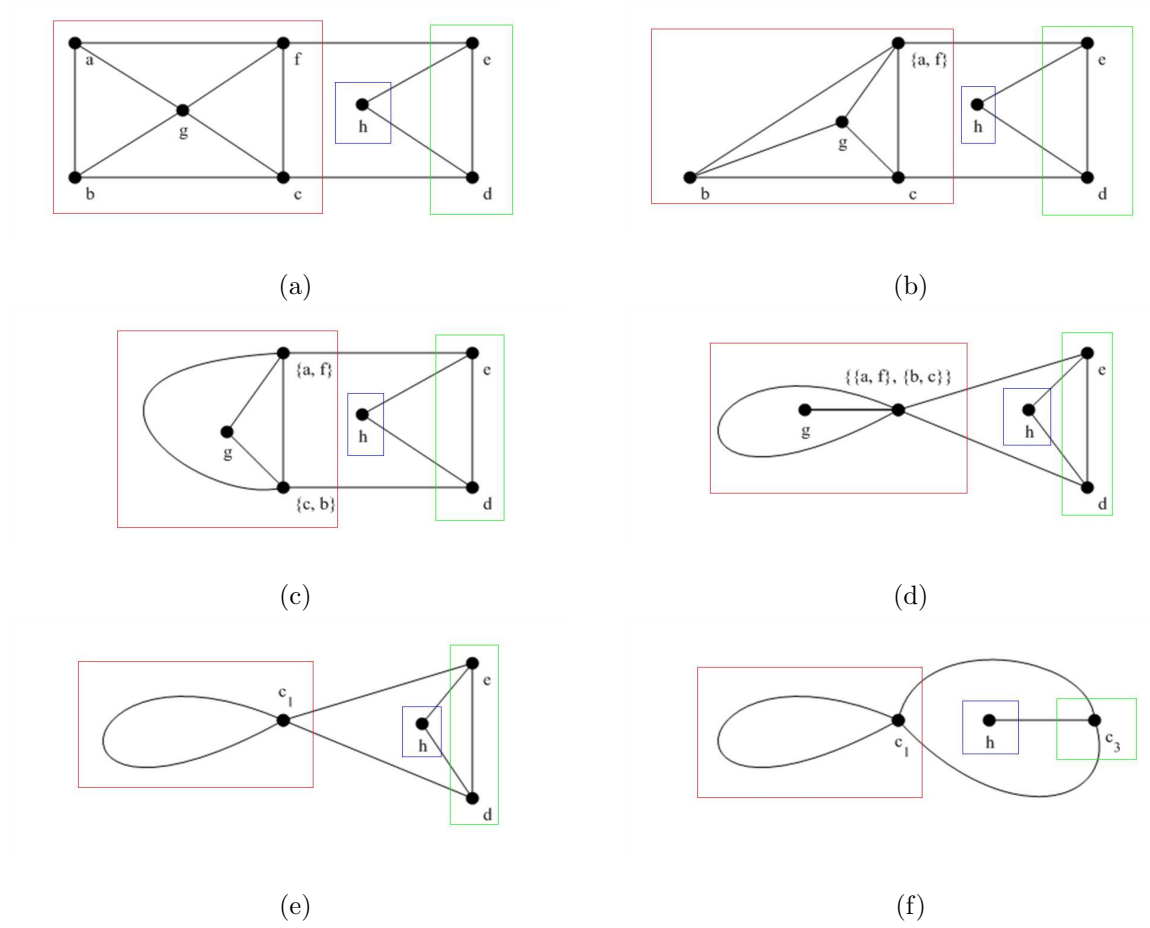


Figure 2.14: (a) A clustered graph with three clusters. (b) The contraction of edge (a, f) originates a complex triangle. (c) The contraction of edge (b, c) creates a multiedge. (d) The contraction of $(\{a, f\}, \{b, c\})$ creates a loop inducing a complex 1-cycle. Notice that the contraction of $(g, \{a, f\})$ (or $(g, \{c, b\})$) would not have generated the loop. (e) The contraction of the edge incident with g turns the complex $1-$cycle into a $1-$face. (f) Contracting edge (d, e) creates a multiedge inducing a complex $2-$cycle.

the space needed to store M is $O(n^2)$. An *adjacency list* of a vertex v (denoted as $Adj(v)$) contains the vertices adjacent to v. The space required to store the adjacency lists of all nodes is $O(2m)$, which, for plane graphs, is $O(n)$, by Euler's Formula. Testing whether two vertices are adjacent requires $O(|N(v)|)$ time. Similarly, an *incidence list* of a vertex v (denoted as $Inc(v)$) contains the edges incident with v.

Here we describe a data structure optimized for the algorithms described in this thesis. Let $\mathscr{G} = (G, T)$ be a c-planar c-connected HCG with k clusters. We assume that clusters can not partially overlap and each cluster contains at least two nodes. Therefore, the number of nodes in T is $O(n)$. The underlying graph G may be a multigraph with loops.

We assign each node v a natural number in the interval $[0, n-1]$ in such a way that, for each cluster $\nu \in T$, the set $V(\nu)$ can be represented as a single interval $[min, max]$, $min \leq max$. This is accomplished with a simple postorder visit of T. If a vertex is dynamically added to a cluster, enforcing this invariant might require renumbering all nodes. To prevent this, we allow (as done in [Rai04]) the vertices belonging to a cluster to be represented as a set of up to k_{max} intervals, where k_{max} is a predefined constant. In order not to make notation too heavy, v refers to both a node and the natural number assigned to it.

Each edge e is arbitrarily assigned a natural number in $[0, m-1]$ and, similarly, each cluster ν is identified with a natural number in $[0, k]$. Cluster 0 is the root of T and is called *root cluster*. e (or ν) refers to either an edge (a cluster) or to the natural number assigned to it.

Tree T is implemented as an array (which we call T too) composed of $k+1$ *cells* numbered from 0 to k. Each cell $T[\nu]$ is a non-terminal node ν of T and contains the following information:

- $V(\nu)$. This set is implemented as a circular double-linked list of up to r intervals $[min_1, max_1]$, ..., $[min_r, max_r]$, $r \leq k_{max}$, such that $min_i \leq max_i$ and $max_i < min_{i+1}$.
- $P(\nu)$. The parent of ν in T.
- $Ch(\nu)$. A double-linked list containing the children of ν in T. In $Ch(\nu)$, leaves occur before any non-terminal node.

$V(G)$ and $E(G)$ are explicitly represented as two separate arrays, which we denote as V_G and E_G respectively. V_G contains n cells, numbered from 0 to $n-1$. Cell $V_G[v]$, relative to node v, contains the following information:

- $Adj(v)$. The adjacency list of v, implemented as a circular double-linked list. The vertices in $Adj(v)$ appear in clockwise order around v.

- $Inc(v)$. The incidence list of v, implemented as a circular double-linked list. Each item of $Inc(v)$ contains an edge $e = (v, u)$ and a pointer to the occurrence of u in $Adj(v)$. In this way, it is possible to switch from an item of $Inc(v)$ to the corresponding item in $Adj(v)$ in $O(1)$ time. The edges in $Inc(v)$ are in clockwise order around v.
- $P(v)$. The parent cluster ν of v in T. Also, a pointer to the occurrence of v in $Ch(\nu)$ is kept.
- Flags. Set of further information about v, used in specific contexts.

Similarly, E_G contains m cells, one for each edge e. $E_G[e]$ is composed of the following information:

- $Ends(e)$. The endpoints u and v of e.
- $Inc(e)$. Pointers to the occurrence of e in $Inc(u)$ and $Inc(v)$.
- Flags. As above.

The information associated with each vertex, edge and cluster require a fixed amount of space. Since each vertex has at most one parent in T and the number of nodes is $O(n)$, the space required by T is $O(n)$. The space needed to store V_G and E_G is $O(m+n)$. Since we only consider c-planar graphs, this data structure requires $O(n)$ space.

2.8.1 Common Operations

This section describes common operations on $\mathscr{G} = (G, T)$. We assume that arrays T, V_G and E_G are dynamic, thus space can be allocated when needed. Any new cell is added at the end of the array. We assume that the following operations on a circular double-linked list L are given.

1. EMPTY(L). Creates a new empty list.
2. BACKINSERT(l, L). Inserts item l at the end of L.
3. FRONTINSERT(l, L). Inserts item l as first element of L.
4. INSERT(l_1, l_2, L). Inserts item l_1 into L after l_2.
5. CONCAT($L_1, e_1, e_2, L_2, e_3, e_4$). Inserts into L_2 items e_1 to e_2 of list L_1 between items e_3 to e_4.

6. DELETE(l, L). Deletes item l from L.
7. FIRST(L). Pointer to the first item in L.
8. LAST(L). Pointer to the last item in L.
9. NEXT(l, L). Returns the item following l in L.
10. PREV(l, L). Returns the item preceding l in L.

Procedures BACKINSERT, FRONTINSERT and INSERT return a pointer to the newly inserted item. The cost of all these operations is $O(1)$.

Edge Insertion

INSERTEDGE$(\mathscr{G}, u, v, p_u, p_v)$ links vertices u and v with a new edge e. An occurrence of u is added to $Adj(v)$, an occurrence of v is added to $Adj(u)$ and an occurrence of e is added to $Inc(u)$ and $Inc(v)$. p_u is the correct position of e in $Inc(u)$ (and consequently the correct position of v in $Adj(u)$); p_v is the correct position of e in $Inc(v)$ (and consequently the correct position of u in $Adj(v)$). Clearly, no change is required in T. The cost of Procedure 2.1 is $O(1)$.

Algorithm 2.1 Edge Insertion

```
1: procedure InsertEdge(𝒢, u, v, p_u, p_v)
2:     Create a new edge e at the end of E.
3:     Ends(e) ← (u, v)
4:     Insert(u, p_v, Adj(v))
5:     Insert(v, p_u, Adj(u))
6:     e_v ← Insert(e, p_v, Inc(v))
7:     e_u ← Insert(e, p_u, Inc(u))
8:     Inc(e) ← (e_u, e_v)
9: end procedure
```

Edge Deletion

DELETEEDGE$(\mathscr{G}, e)$ deletes edge $e = (u, v)$ from $\mathscr{G}$. u is deleted from $Adj(v)$, v is deleted from $Adj(u)$ and e is deleted from $Inc(u)$ and $Inc(v)$. Finally, the content of cell $E_G[m-1]$ is moved to $E_G[e]$, to prevent E_G from having unused cells. Edge $m-1$ is renumbered as e; the occurrences of $m-1$ in the incidence lists of its endpoints must be renumbered accordingly. No change is required in T. The cost of Procedure 2.2 is $O(1)$.

Algorithm 2.2 Edge Deletion

```
 1: procedure DeleteEdge(𝒢, e)
 2:     (u, v) ← Ends(e)
 3:     (e_u, e_v) ← Inc(e)
 4:     Delete(v, Adj(u))
 5:     Delete(u, Adj(v))
 6:     Delete(e_u, Inc(u))
 7:     Delete(e_v, Inc(v))
 8:     (u, v) ← Ends(m − 1)
 9:     (e_u, e_v) ← Inc(m − 1)
10:     Renumber as e the occurrence pointed by e_u in Inc(u)
11:     Renumber as e the occurrence pointed by e_v in Inc(v)
12:     E_G[e] ← E_G[m − 1]
13: end procedure
```

Cluster Creation

CreateCluster$(\mathscr{G}, \nu, \mu)$ creates an empty cluster ν as child of μ in T. The new node ν is a leaf, although it does not represent a vertex of the underlying graph. The data structure remains inconsistent until at least two vertices are added to ν. The cost of Procedure 2.3 is $O(1)$.

Algorithm 2.3 Cluster Creation

```
1: procedure CreateCluster(𝒢, ν, μ)
2:     Empty(V(ν))
3:     P(ν) ← μ
4:     Empty(Ch(ν))
5:     BackInsert(ν, Ch(μ))
6: end procedure
```

Node Insertion

InsertNode$(\mathscr{G}, v, \nu)$ adds a new node v to cluster ν. Clearly, $Adj(v)$ and $Inc(v)$ are empty. v is added as first item of list $Ch(\nu)$. Finally, for each ancestor μ of ν in T, $V(\mu)$ is to be modified. Let $[min, max]$ be the last interval in $V(\mu)$; as v is a new node, $v \geq max + 1$. Thus, if $v = max + 1$, then interval $[min, max]$ is changed to $[min, v]$; otherwise, a new interval $[v, v]$ is added at the end of $V(\mu)$. Since v is contained in up to $D(\mathscr{G})$ clusters, the cost of Procedure 2.4 is $O(D(\mathscr{G}))$.

Algorithm 2.4 Node Insertion

```
1: procedure InsertNode(𝒢, v, ν)
2:     Empty(Adj(v))
3:     Empty(Inc(v))
4:     P(v) ← ν
5:     FrontInsert(v, Ch(ν))
6:     μ ← P(ν)
7:     while μ ≠ 0 do
8:         [min, max] ←Last(V(μ))
9:         if max = v − 1 then
10:            Last(V(μ)) ← [min, v]
11:        else
12:            BackInsert([v, v], V(μ))
13:        end if
14:    end while
15: end procedure
```

Subcluster Test

isSubCluster$(\mathscr{G}, \nu, \mu)$ tests whether ν is a subcluster of μ. It suffices to check whether a node $v \in V(\nu)$ also belongs to $V(\mu)$. Since $V(\mu)$ contains at least k_{max} intervals, Procedure 2.5 requires $O(k_{max})$ time.

Algorithm 2.5 Subcluster Test

```
1: function isSubCluster(𝒢, ν, μ): boolean
2:     [v, w] ← First(V(ν))
3:     for all [min, max] ∈ V(μ) do
4:         if v < min then
5:             break
6:         else if min ≤ v ≤ max then
7:             return true;
8:         end if
9:     end for
10:    return false;
11: end function
```

Edge Contraction

ContractEdge(G, e) contracts an edge $e = (u, v)$ of a plain graph G. Vertex v_e which replaces u and v after the contraction is saved in $V_G[v]$.

Algorithm 2.6 Edge Contraction

```
 1: procedure ContractEdge(G, e)
 2:     (u, v) ← Ends(e)
 3:     e_{1,u} = (u, u_1) ← Prev(e, Inc(u)); e_{2,u} = (u, u_2) ← Next(e, Inc(u))
 4:     e_{1,v} = (v, v_1) ← Prev(e, Inc(v)); e_{2,v} = (v, v_2) ← Next(e, Inc(v))
 5:     if u_1 = v_2 then
 6:         DeleteEdge(G, e_{1,u}); e_{1,u} = (u, u_1) ← Prev(e, Inc(u))
 7:     end if
 8:     if u_2 = v_1 then
 9:         DeleteEdge(G, e_{2,u}); e_{2,u} = (u, u_2) ← Next(e, Inc(u))
10:     end if
11:     DeleteEdge(G, e)
12:     for all edge e in Inc(u) do
13:         (x, u) ← Ends(e); Ends(e) ← (x, v)
14:     end for
15:     concat(Inc(u), e_{2,u}, e_{1,u}, Inc(v), e_{1,v}, e_{2,v})
16:     concat(Adj(u), u_2, u_1, Adj(v), v_1, v_2)
17:     Mark V_G[u] as deleted
18: end procedure
```

Let $e_{1,u}$ and $e_{2,u}$ be the edge preceding and following e in $Inc(u)$ respectively; similarly, let $e_{1,v}$ and $e_{2,v}$ be the edge preceding and following e in $Inc(v)$ (Fig. 2.15). If $u_1 = v_2$, edge $e_{1,u}$ (or $e_{2,v}$) is deleted; similarly, if $u_2 = v_1$, edge $e_{2,u}$ (or $e_{1,v}$) is deleted. Then, e is deleted using Procedure 2.2. Since v_e is saved to $V_G[v]$, the endpoints of all edges incident with u must be updated (which requires $O(|N(u)|)$ time). The new incidence list of v_e is created by inserting into $Inc(v_e)$ items $e_{2,u}$ to $e_{1,u}$ of $Inc(u)$ between $e_{1,v}$ and $e_{2,v}$. $Adj(v_e)$ is created similarly. Finally, $V_G[u]$ is marked as deleted: thus, this cell remains unused. The cost of Procedure 2.6 is $O(|N(u)|)$.

Quotient Graph

Function QuotientGraph creates the quotient graph Q of a CG $\mathscr{G} = (G, \mathscr{C})$ with k clusters. In the data structure of Q, the macro-vertex c_i corresponding to cluster C_i is assigned the same natural number assigned to C_i.

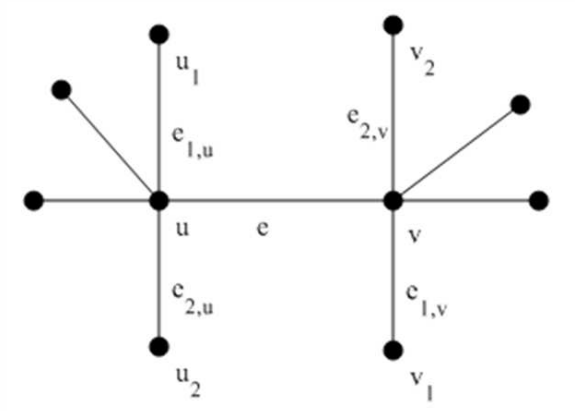


Figure 2.15: Contraction of edge e. If $u_1 = v_2$, then either $e_{1,u}$ or $e_{2,v}$ is deleted; similarly, if $u_2 = v_1$, then either $e_{2,u}$ or $e_{1,v}$ is deleted.

The idea of the algorithm is simple: for each cluster C, the intracluster edges of a predefined node $v \in C$ are iteratively contracted. First, the data structure of Q is initialized with the nodes and edges of the underlying graph G. This takes $O(n)$ time. Next, array V_Q is ordered so that $V_Q[i]$ contains a vertex of cluster i, $1 \leq i \leq k$. To this extent, each node $1 \leq i \leq n$ is visited and the following cases occur:

Algorithm 2.7 Quotient Graph Computation

```
1: function QuotientGraph(𝒢_𝒞 = (G, 𝒞)): Graph
2:     Initialize Q with G
3:     Order V_Q so that V_Q[i] contains a vertex of cluster i, 1 ≤ i ≤ k
4:     for all 1 ≤ v ≤ k do
5:         Initialize L_v with the intracluster edges incident with v
6:         for all e = (w, v) in L_v do
7:             ContractEdge(Q, e)
8:             for all intracluster edge e_w incident with w do
9:                 BackInsert(e_w, L_v)
10:            end for
11:        end for
12:    end for
13:    return Q
14: end function
```

1. The node in $V_G[i]$ belongs to cluster i. The node in $V_G[i+1]$ is considered.

2. The node in $V_G[i]$ belongs to cluster j, $i \neq j$, and the node in $V_G[j]$ belongs to cluster i. $V_G[i]$ and $V_G[j]$ are swapped and the vertex $V_G[i+1]$ is considered.

3. The node in $V_G[i]$ belongs to cluster j, $i \neq j$, and the node in $V_G[j]$ belongs to cluster j. The node in $V_G[i+1]$ is considered.

4. The node in $V_G[i]$ belongs to cluster j and the node in $V_G[j]$ belongs to cluster k, $i \neq j \neq k$. $V_G[j]$ is temporarily stored in $V_G[0]$ and $V_G[i]$ is moved to $V_G[j]$. While $V_G[V_G[0]] \neq V_G[0]$, nodes $V_G[0]$ and $V_G[V_G[0]]$ are swapped. When $V_G[V_G[0]] = V_G[0]$, the two values are swapped and the next vertex $V_G[i+1]$ is considered.

Using $V_G[0]$ as a temporary location ensures that a node is moved only once in V_Q. Each time a node v is moved, in fact, the adjacency lists of the nodes in $Adj(v)$ (and the edges in $Inc(v)$) must be updated (which requires $O(N(v))$ time). Finally, ordering array V_Q takes $O(n)$ time.

After the initialization, for every node v, $1 \leq v \leq k$, the intracluster edges of every node are iteratively contracted. To this extent, an auxiliary list L_v, which stores the intracluster edges of v, is used. Testing whether an edge $e = (v, w)$ is intracluster is accomplished in constant time checking whether $P(v) = P(w)$. Therefore, the initialization of L_v requires $O(|N(v)|)$ time. Let $e = (v, w)$ be an intracluster edge to contract. The contraction of e deletes v and w and creates, as seen before, a new vertex which is stored in $V_Q[v]$. Thus, we can still denote the new vertex as v. The contraction of e costs $O(|N(w)|)$ time. As a result of the contraction, v inherits from w some edges. As seen before, in $Inc(v)$ the edges of w are between the edge preceding and that following e. Therefore, adding to L_v the inherited intracluster edges takes $O(|N(w)|)$ time. Finally, contracting a cluster ν requires $O(|V(\nu)|)$ time. Since clusters in a clustered graph do not overlap, the cost of contracting all clusters is $O(n)$.

2.9 Problems in Graph Theory

This section defines some classic problems in graph theory which this thesis comes across.

Definition 2.26. *A* matching *is a set of edges in a graph G such that no two of them have an endpoint in common.*

Given a graph G and a matching M, if $e \in M$ then e is *matched*, otherwise is *free*. A vertex v is called *covered* if at least one of its incident edges is matched, *uncovered* otherwise. M is said *maximum* if it has the largest cardinality among all possible matchings of G. An $M-$*alternating path* is a path where edges are alternatively matched and free. If the endvertices of an $M-$alternating path Π are both free, Π is called an $M-$*augmenting path.*

Table 2.1: Algorithms for maximum matching

Year	Author	Complexity
1965	Edmonds	$O(n^4)$
1965	Witzgall and Zahn	$O(mn^2)$
1967	Balinski	$O(n^3)$
1974	Kameda and Munro	$O(mn)$
1975	Even and Kariv	$O(n^{2.5})$
1976	Gabow	$O(n^3)$
1976	Lawler	$O(n^3)$
1980	Pape and Conradt	$O(n^3)$
1980	Micali and Vazirani	$O(\sqrt{n}m)$
1983	Gabow and Tarjan	$O(mn)$

Theorem 2.3 (Berge, 1957)**.** *Let G be a graph and M a matching. M is a maximum matching if and only if there is no $M-$augmenting path in G.*

If the matched edges of an $M-$augmenting path Π are replaced in M by the free edges of Π, a new matching M' is obtained such that $|M|' = |M| + 1$. To compute a maximum matching, this procedure must be iterated until there are augmenting path. Unfortunately, finding the augmenting paths is not trivial. Table 2.1 gives a list of the most important algorithms for maximum matching in general graphs.

A *perfect matching* (or $1-$factor) is a matching such that each each vertex is covered. A perfect matching is the largest possible matching and contains $n/2$ edges. Not every graph admits a perfect matching, as stated by the famous Tutte's Theorem.

Theorem 2.4 (Tutte, 1947)**.** *Let G be a graph and $q(G)$ be the number of its connected components of odd order. G has a perfect matching if and only if $q(G - S) \leq |S|$, for all $S \subseteq V(G)$.*

Particularly interesting is Petersen's Theorem, which concerns cubic graphs.

Theorem 2.5 (Petersen, 1891)**.** *Every bridgeless cubic graph has a perfect matching.*

A nice proof of Petersen Theorem is due to Frink [Fri26]. This proof was used in [BBDL01] to describe a linear-time algorithm for computing a perfect matching in a cubic bridgeless graph.

A problem strictly related to matching is *covering*. A *vertex covering* of a graph G is a set of vertices C such that each edge of G is incident with at least one vertex in C. An *edge covering* is a set of edges C such that each node is incident with at least one edge in C.

It is easy to prove that a perfect matching is a *minimum edge covering* (an edge covering with the smallest possible cardinality).

A relatively new concept in graph theory is *tree decomposition*, introduced by Robertson and Seymour in their fundamental work on graph minors [RS86].

Definition 2.27. *A* tree decomposition *of G is a pair (T, χ), where $T = (I, F)$ is a tree with nodes I and edges F, and $\chi = \{X_i : i \in I\}$ is a family of subsets of V (called* bags*), one for each node of T, such that:*

1. $\bigcup_{i \in I} X_i = V$
2. *for every edge $(u, v) \in E$, there is an $i \in I$ with $v \in X_i$ and $u \in X_i$*
3. *for all i, j, $k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$*

The *width* of a tree decomposition is $max_{i \in I}|X_i| - 1$. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of G. The notion of *treewidth* is important, as it was proved that many NP-hard problems can be actually solved in polynomial time on graphs with bounded treewidth [Bod93].

Chapter 3

Rectangular Dualization

In this chapter we introduce *rectangular dualization* and some concepts that are closely related. We describe in details two applications which motivate the importance of rectangular dualization. Finally, we describe a new algorithm which creates a rectangular dual representation of any given plane connected graph.

3.1 Introduction

A rectangular dual of a plane graph G is a dual representation of G, which is part of a more general class of graph layouts, known as *rectilinear duals*.

Definition 3.1. *A* rectilinear dual *of a plane graph G is a pair (Γ, f) such that:*

1. *$\Gamma = \{R_1, R_2, ..., R_n\}$ is a set of simple non-overlapping rectilinear regions*[1]*, which form a partition of a rectangle R (called* enclosure rectangle*);*
2. *$f : V(G) \rightarrow \Gamma$ is a one-to-one function associating each vertex of G to a distinct region in Γ;*
3. *Two vertices u and v are adjacent in G if and only if the associated regions $f(u)$ and $f(v)$ are adjacent* [2]*.*

The *complexity* of a rectilinear region is the total number of the vertices of the bounding polygon; the *complexity* of a rectilinear dual is the maximum complexity of its regions.

[1] A *rectilinear region* is a region of the plane bounded by a polygon whose sides are parallel to the $x-$ or $y-$axis.

[2] Two regions are adjacent if they have one side or a segment of it in common.

A *rectangular dual* $\mathscr{R}(G) = (\Gamma, f)$ of G (Fig. 3.1) is a rectilinear dual of G such that the regions in Γ are all rectangles.

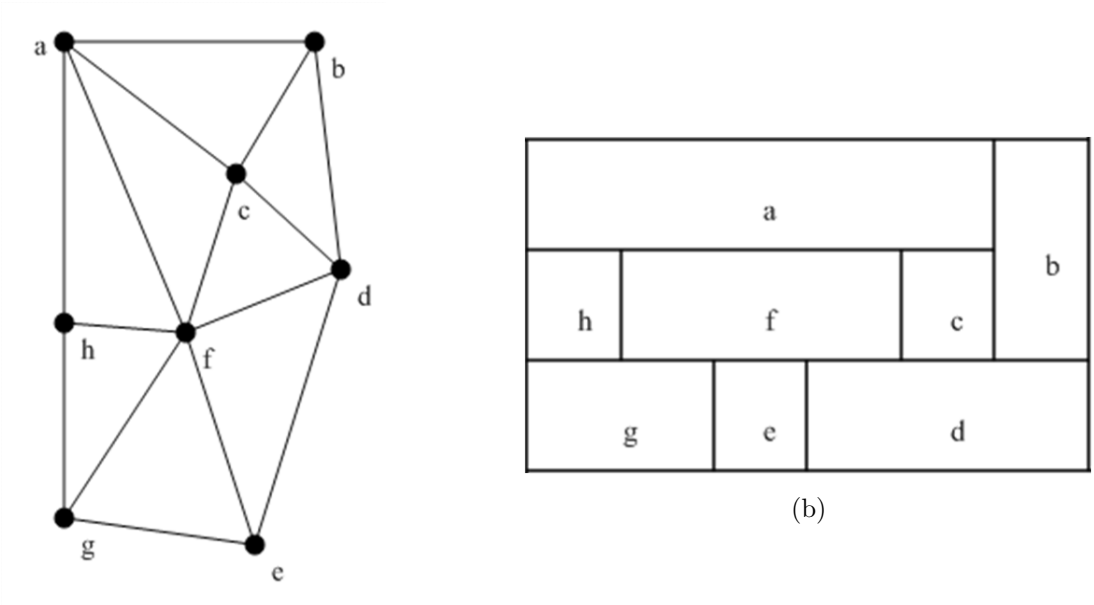


Figure 3.1: (a) A plane graph G. (b) A rectangular dual of G.

The term "dual" in the previous definition is not accidental. Consider the graph in Fig. 3.1 and add four exterior vertices; you will obtain the graph shown in Fig. 3.2 and denoted as G_4. The rectangular dual of G is the geometric dual of G_4 with the following properties:

- Each edge is either a vertical or a horizontal line segment.
- Each face (including the exterior one) is a rectangle, which does not necessarily mean that is a $4-$face. The term "rectangle" refers to the shape of the face, not to the length of its frontier.
- Exactly four vertices (called *corner vertices*) have degree 2; the remaining have degree 3[3].

Not every plane graph, though having a geometric dual, admits a rectilinear or rectangular dual. Liao et al. [LLY03] proved that any PTG has a rectilinear dual. The admissibility conditions for rectangular dualization are fare more restrictive, because in a rectangular

[3]Theoretically, it would be possible for a node to have four incident edges. However, we only consider rectangular duals such that no four rectangles meet at the same point.

dual all regions must have the same shape. Usually, a graph admitting a rectangular dual is referred to as *rectangular graph*. In general, the rectangular dual of a rectangular graph is not unique.

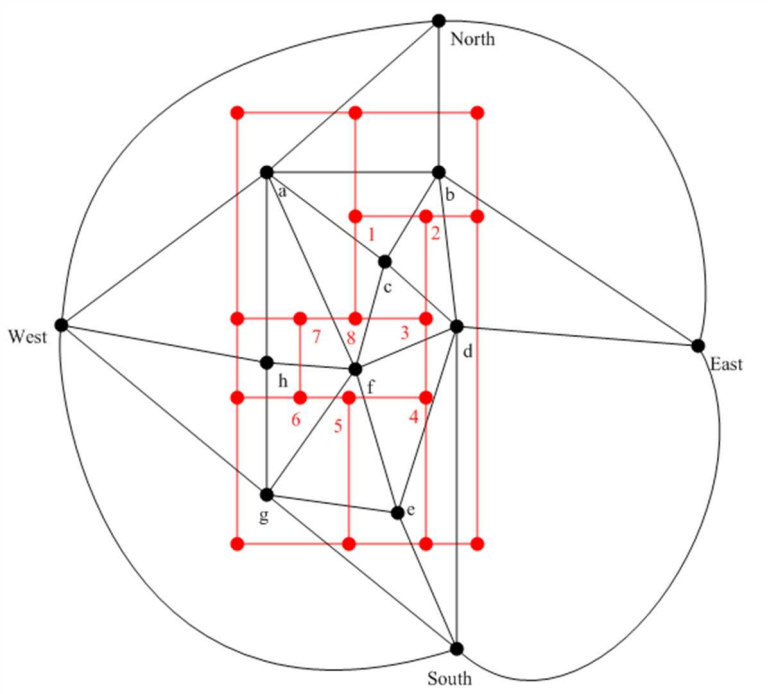


Figure 3.2: The graph G_4 (in black) obtained from graph G (Fig. 3.1) adding four exterior vertices. The geometric dual of G_4 (in red) is a rectangular dual of G.

The history of rectangular dualization goes back a long way, at least as far as when Ungar [Ung53], who defined a *saturated plane map* as a finite set of non-overlapping regions that partition the plane, complying with the following conditions:

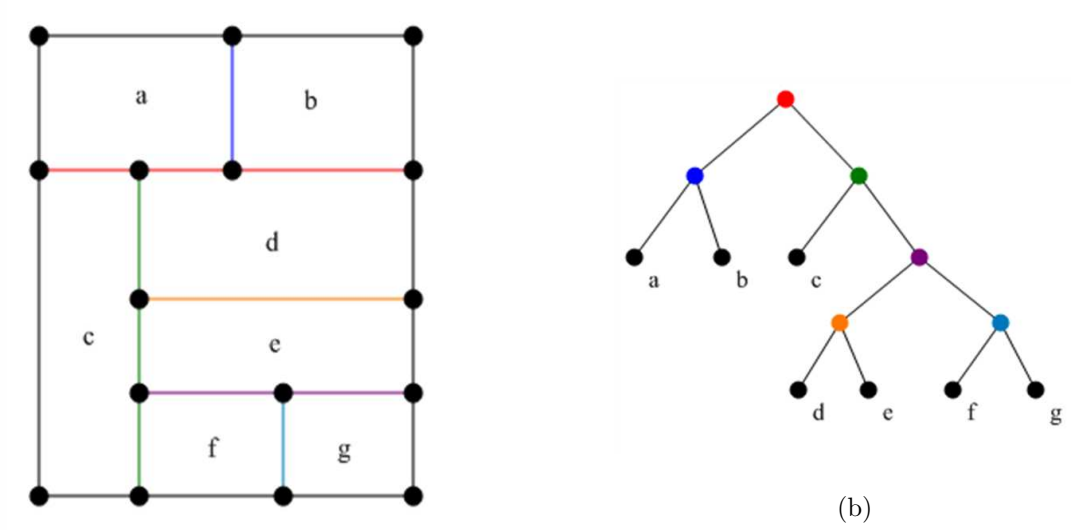
- Precisely one region is infinite.
- At most three regions meet in any point.
- Every region is simply connected (it has no holes).
- The union of any two adjacent regions is simply connected.
- The intersection of any two regions is either empty or a simple arc.

Ungar also proved that a rectangular dissection (that is a rectangular dual) is isomorphic to a saturated plane map, which does not contain any set of three regions whose union is multiply connected (that is a region with a hole). This result is dusted off in [BGPV08] to give the admissibility conditions for a special case of rectangular dual, known as *rectangular layout*. The first formulation of rectangular dualization in graph-theoretic terms is due to Grason, who described an automated approach to the space planning problem in architecture [Gra71]. From then, lots of papers have been describing efficient algorithms for creating floorplans of buildings. The popularity of rectangular dualization spread with the advent of VLSI circuits and the urgent need of tools which eased the design process. At the same time, many researchers concentrated on a more rigorous mathematical formalization [BS86, He97, KH94, KK84, LL90, YS95].

3.2 Related Work

3.2.1 Sliceable Rectangular Dual

A rectangular dual $\mathscr{R}$ is *sliceable* if it is a rectangle or can be partitioned into two sliceable rectangular duals by a horizontal or vertical slice (Fig. 3.3). A *slice* is a set $S = \{e_1, ..., e_k\}$



(a) (b)

Figure 3.3: (a) A sliceable rectangular dual $\mathscr{G}$. Edges belonging to the same slice have the same colour. (b) The BSP tree associated to $\mathscr{R}$. Each non-terminal node has the same colour as the corresponding slice.

of edges of $\mathscr{R}$ such that all e_i's lie on the same line and e_1 and e_k are incident with an exterior edge. Slices define a hierarchy, which can be represented with a Binary Space Partition (BSP) tree. The leaves of the BSP tree are the rectangular regions of $\mathscr{R}$; each node ν corresponds to a rectangle $R(\nu)$ and the subtree rooted at it contains the rectangles enclosed in $R(\nu)$. Finally, each node is labelled with the slice $l(\nu)$ that divides $R(\nu)$ into two sliceable sublayouts.

The hierarchical structure of a sliceable rectangular dual helps in solving some problems more efficiently than in non-sliceable rectangular duals. An example is the *rectangular dual area optimization problem*, which asks for determining the size of each rectangle to minimize the area of the rectangular dual. This problem was shown to be NP-complete for general rectangular duals [Sto83], whereas polynomial-time algorithms are known for sliceable ones [Sto83, Ott83, WS89]. An $O(n^2 \log n)$-time algorithm, transforming an arbitrary rectangular dual into a sliceable one, is described in [Sar93]. However, this is not always possible, as there exist a class of rectangular duals which are *inherently non-sliceable* (Fig. 3.4) [SKB88, SKB91].

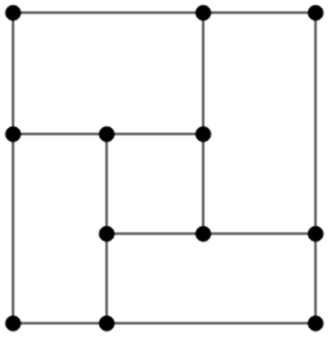


Figure 3.4: A inherently non-sliceable rectangular dual.

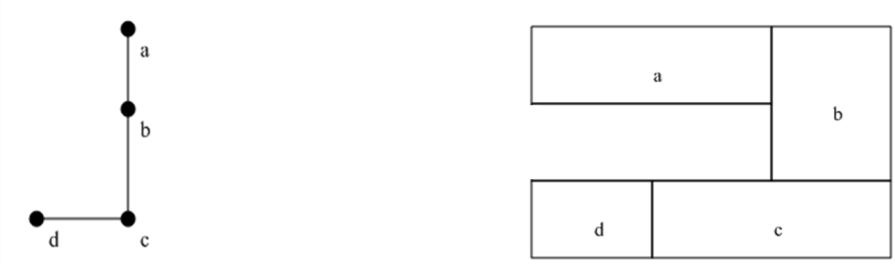
If a rectangular graph admits a sliceable rectangular dual is called *sliceable graph*. An exact characterization of sliceable graphs is still unknown. Yeap and Sarrafzadeh proved that if a rectangular graph contains no complex $4-$cycle, then it admits a sliceable rectangular dual (but the converse is not true) [YS95] . Dasgupta and Sur-Kolay refined this condition, proving that a sliceable rectangular dual exists if any complex $4-$cycle is maximal (it is not enclosed in other complex $4-$cycles) [DSK97].

In [YS95] an algorithm is described for creating a sliceable rectangular dual. A proper slice S is looked for in the input graph G, so that removing the edges of S separates G into two connected components G_l and G_r, which admit a sliceable rectangular dual. The procedure is then iterated on G_l and G_r. The computational cost (dominated by the search

of the complex 4−cycles) is $O(n \log n + hn)$, h being the height of the tree describing the hierarchy.

3.2.2 Rectangular Layouts

A *rectangular layout* is a collection of rectangles whose union is not a rectangle (Figure 3.5). The geometric dual graph of a rectangular layout is a *contact graph of rectangles*. Given a set of objects in the space (rectangles or triangles, for example), the associated *contact graph* has a vertex for each object and an edge between two vertices if and only if the corresponding objects touch in some prescribed fashion. In 1936 Koebe proved that any planar graph can be expressed as the contact graph of disks in the plane.



(a) (b)

Figure 3.5: (a) A plane graph G. (b) A rectangular layout of G.

In [BGPV08] the authors claim several advantages of rectangular layouts over rectangular duals. A graph admits a rectangular layout if and only if it contains no complex 3−cycles; admissibility conditions for rectangular dualization are stronger (Section 3.6). Trees, indeed, do not admit a rectangular dual but do admit a rectangular layout. The area of a rectangular layout is, in general, smaller than the area of a rectangular dual. There are infinite graphs which have a $O(n)$-area rectangular layout and a $O(n^2)$-area rectangular dual.

Although these advantages are undoubted, a rectangular dual is more general than a rectangular layout. As an evidence of this, the algorithm proposed in [BGPV08] uses a rectangular dual to create a rectangular layout.

3.2.3 Rectangular Dual of Directed Graphs

While many words have been spent on rectangular dualization for undirected graphs, only one paper (at least in our knowledge) deals with directed graphs [BGV07]. Let G be a

directed plane graph; $\mathscr{R}(G)$ is a rectangular dual (Γ, f) of G such that for all $u, v \in V(G)$, $(u, v) \in E(G)$ if and only if $f(u)$ is above or to the left of $f(v)$. Clearly, many variations of this definition are possible. The generalization of rectangular dualization to directed graphs has been motivated by the need of developing a graphical user interface to a relational database support system [BGV07]. The admissibility conditions do not differ much from those given in the case of undirected graphs.

3.2.4 Cartograms

A *cartogram* is a geographic map consisting of a collection of regions, whose area is proportional to some predefined value. In a *rectangular cartogram* all regions are rectangular-shaped (Fig. 3.6).

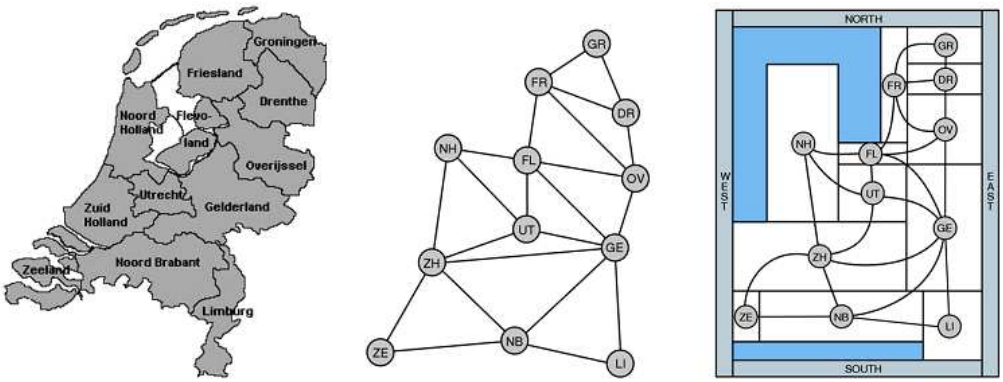


Figure 3.6: From left to right: the provinces of the Netherlands, the corresponding adjacency graph and cartogram. Taken from [dBMS08].

The first known cartograms appeared in the economic geography textbooks (1868 - 1875) by Emile Levasseur [Tob04]. However, only in 1934 Erwin Raisz gave cartograms academic attention with his seminal book "The rectangular statistical cartogram". The aim of a cartogram is to give an immediate and clear idea of how a statistical value varies from region to region. Therefore, it is not excessive to affirm that cartograms belong to information visualization. Raisz mostly concentrated on rectangular cartograms, as a rectangle is an easy shape, whose area can be quickly estimated by visual inspection.

Recently, cartograms were dusted off and formalized in graph-theoretic terms [dBMS08, vKS07]. Let $G = (V, E, w)$ be a vertex-weighted plane graph, where $w : V \to \mathbb{N}$ is a function assigning an integer weight to any vertex. A *rectangular cartogram* of G is a rectangular dual (Γ, f) of G such that the area of each rectangle $f(u)$ equals $w(u)$. A rectangular cartogram can be also called *weighted rectangular dual*. In order to create a

cartogram, such as the one in Fig. 3.6, an *adjacency graph* is first created; this graph contains a node for each region and an edge between two nodes if and only if the corresponding regions are adjacent. Next, a weighted rectangular dual is computed.

Clearly, it is impossible to create a rectangular cartogram for any plane graph with arbitrary weights. Usually, an approximation is looked for, trying to minimize two parameters: *cartographic error* and *adjacency error*. The first refers to how well the area of each rectangle $f(u)$ approximates $w(u)$, for each vertex $u \in V$. When minimizing the cartographic error, two rectangles $f(u)$ and $f(v)$, with $(u, v) \in E$, may result not adjacent; this improves the adjacency error. Efficient algorithms creating approximated rectangular cartograms are presented in [vKS07].

The only way to obtain a cartogram with neither adjacency nor cartographic error is to allow regions to have shapes different than rectangle, although the union of all regions must fit a rectangle. In other words, the cartogram is thought of as a weighted rectilinear dual of G and is called *rectilinear cartogram*. Every plane graph admits a rectilinear dual of complexity at most 8 [LLY03]. A rectilinear cartogram with complexity at most 8 can be always computed for a special subclass of the sliceable graphs [RMN09]. In [dBMS08] this result is improved by proving that:

- Every sliceable graph has a rectilinear cartogram with complexity at most 12.
- Every rectangular graph has a rectilinear cartogram with complexity at most 20.
- Every PTG has a rectilinear cartogram with complexity at most 40.

Three algorithms are described for each of the previous cases. Let G be a vertex-weighted sliceable graph, whose weights are normalized so that they sum to 1. A sliceable rectangular dual $\mathscr{R}$ (and the corresponding BSP) is created using the algorithm described in [YS95]. $\mathscr{R}$ is scaled and stretched so that it becomes a partition of the unit square. Next, the area of each rectangular region is adjusted right. This is achieved by traversing the BSP tree in a top-down fashion and moving the slice $l(\nu)$, at each node ν, to make the area of $R(\nu)$ equal to the sum of the required areas of the regions represented by the leaves below ν. An heuristic procedure sets right any adjacency which may have altered while adjusting the areas. Finally, a further adjustment of the areas is done, while preserving the adjacencies. The algorithm works in $O(n \log n)$ time.

A cartogram for the second and third case is created with the same procedure, after turning the input graph into a sliceable graph.

3.3 Applications

In this section we discuss two important applications, which motivated the development of a theory of rectangular dualization. Although this thesis highlights mainly the impact of rectangular dualization on graph visualization (Chapter 5), we believe that also these two applications may benefit from our results.

3.3.1 Space Layout Planning

The earliest known application of rectangular dualization is *space layout planning*, which is a typical problem in architectural design, although it can also be generalized to "all disciplines that deal with the organization of elements in three dimensional space" [Fre80]. Space allocation in buildings has to respond to aesthetic requirements, as well as specific constraints imposed by the activity performed within the building. Whereas a computer program can help on the first aspect only to a certain extent (though some authors would claim differently), it can suggest quickly and efficiently a high number of different solutions. The architectural design is driven by dimensional and topological constraints. The first impose restrictions on room surface, length, width and orientation; the latter depend on the organization of the work in the building. For instance, two rooms may be required to be adjacent because are devoted to the same activity.

Space layout planning is either an ill-posed or an over-constrained problem or both. It is ill-posed in the sense that the constraints initially are not fully defined and formulated; often is an architect puzzled with a number of ideas on how to design the building. It is over-constrained in the sense that the constraints can be so many that no single best solution (or no solution at all) exists. The first known computational approach to the problem is called SLAP1, a software developed in 1965 by Thomas Anderson, a student in civil engineering at the University of Washington [Hom06]. This program performs a topological sorting of a group of activities so that to minimize the total cost of circulation between two activities. A number of different approaches have been proposed; Galle classified them into five categories:

- **On-line machine control and appraisal of the designer's layout proposal**. This kind of approach is human-centred in that it is the designer that devises a layout and the software limits itself to evaluate it. According to Maver, in fact, humans are far superior than machines in solving real-world problems [Mav70].
- **Stepwise automatic layout generation**. This approach put more responsibilities on software, which creates layouts being driven by the designer decisions.

- **Nonexhaustive automatic generation**. The software generates a high number of solutions based on a set of given constraints. These approaches are purely heuristic.
- **Exhaustive automatic generation**. It was introduced by Grason, who laid the very first foundations of rectangular dualization. Exhaustive generation is a challenging goal, which the algorithm developed by Grason reaches only with small inputs.
- **Automatic generation of (quasi-) optimal layouts**. These are operational research approaches. The goal is to minimize (or maximize) one or more objective functions, which should express the quality of a floorplan. As it is normal, there is no agreement among authors on the criteria to judge the quality of a layout.

It is striking how many buildings are rectangular-shaped; this is not simply a freak occurrence. In 2006, Steadman explained that rectangles allow architects to size and position rooms in a more flexible way than other geometric shapes [Ste06]. Rectangles and circles are the most used shapes in buildings. Generally, circular buildings characterize nomadic populations, whereas sedentary people live in rectangular houses [BS07]. As an evidence of this, circular buildings are in general constructed with lightweight materials, which are not thought for lasting longtime. Finally, there would be also psychological reasons behind the choice of a rectangle [SH84]. These studies confirm that rectangular dualization is a good choice for creating floorplans for buildings.

Graph-Theoretical Approaches

In an optimal floorplan of a building, rooms are placed so that to comply with a set of predefined constraints. The building is assumed to be rectangular and so are rooms. Under these assumptions, a floorplan is then a rectangular dissection. The geometric dual of a floorplan is a plane graph which contains a node for each room and an edge between any two nodes if and only if the corresponding rooms are adjacent. This graph is usually referred to as *adjacency graph*, as it describes the adjacencies between rooms. Grason [Gra71] was the first to characterize the adjacency graph as the dual graph of a floorplan. Although he never mentioned *rectangular dualization*, a rectangular dual of the adjacency graph is a floorplan. For this reason, Grason is considered the pioneer of rectangular dualization.

The algorithm proposed by Grason aims at building the adjacency graph by adding nodes one by one until a complete graph is obtained. The complete graph must be *physically realizable*, meaning that it admits a rectangular dual. The incomplete graphs obtained at each stage may not be physically realizable, but they have to fulfil some properties which prevent the creation of a non-physically realizable complete graph. A complete graph G is physically realizable if the following are true:

1. G is plane.

2. All nodes are *well-formed*. Since each room r has four walls, each node of G has at least four incident edges. Moreover, the rooms adjacent to r may be north, west, south or east of r. Accordingly, four types of edges are defined for each case (blue outward, red inward, blue inward and red outward respectively). The edges incident with a well-formed node v appear in clockwise order around v as: a set of blue outward edges, a set of red inward edges, a set of blue inward edges and a set of red outward edges. We will come across this property in Section 3.7.1, when talking of Regular Edge Labelling.

3. All faces are well-formed. Grason lists five types of well-formed regions, according to their shapes and the type of the edges bounding them: four are triangular and only one is rectangular[4].

The admissibility conditions for rectangular dualization described in the next chapter are very similar to Grason's. However, Grason did not prove them and only guessed them from experience.

In 1978, Ruch proposed another interesting approach [Ruc78]. Most of software tools force architects to specify all constraints at the very beginning of the design process. This makes the problem ill-posed or overconstrained. Ruch's algorithm goes through three steps. First, the adjacency graph is created. Among all possible embeddings, the algorithm chooses the one which best complies with the initial requirements and with interactively specified constraints. If, for instance, a room must be adjacent to an external wall, its corresponding node must be incident with the exterior face. Second, a "bubble diagram" is generated; each node of the adjacency graph is transformed into a circle with the required area. Finally, the bubble diagram is transformed into a rectangular layout, by substituting each circle with their circumscribed square. A further step may be required to eliminate room overlapping.

3.3.2 VLSI Physical Design

The development of integrated circuits revolutionized the hardware industry more than any other technology. The creation of the large circuits in use in today's electronic equipment, in fact, would not have been possible without integrated circuits. Cost and performance are only two advantages of integrated circuits over discrete ones. A discrete circuit is created by adding all components (transistors, capacitors, resistors and diodes) by hand: all connections must be intact for the circuit to work. A single error on a connection may cause the circuit to stop and then fail. The task becomes more difficult as the circuit size increases. Another problem is the length of the wires connecting each component.

[4]Grason allows floorplans to have four rectangles meeting at a common point.

The greater is the length, the slowest is the circuit. Integrated circuits came to solve all these problems. All components are part of the same monolithic block of semiconductor material and the connections are added as a layer on top of it. Nowadays, integrated circuits are mass-produced using a technique called *photolitography*; this allows a circuit to be replicated as many times as needed.

VLSI (Very Large Scale Integration) is a technology which makes possible the fabrication of integrated circuits with millions of transistors. The Intel 80886 microprocessor, released in 2007, has 1 billion transistors. Designing such a complex circuit would not be possible with pencil and paper; special software, known as CAD (Computer-Aided Design), automate most of the design task and let users concentrate mostly on high-level details. Early CAD software aimed at placing logic components on printed circuit boards, while minimizing the number of transistors. This approach was suitable for small integrated circuits. Today's design automation software also feature powerful tools for simulating the circuit behaviour before actually realizing it. Moreover, it makes no sense minimizing the number of transistors in VLSI circuits, whereas it is highly desirable to save interconnection wires.

Usually the design of a VLSI circuit, due to its complexity, is subdivided into four abstraction levels: architectural design, logic design, physical design and fabrication. The first level is mostly carried out by an expert engineer with no or little help from computers. If the circuit being designed is a processor, typical decisions taken at this stage regard the instruction set of the processor, the level of cache memory, the interface offered by the processor to external components and so on. After defining the architecture, the data and control path are designed. The *data path* of a circuit includes functional blocks (ALUs, multipliers, adders...) and storage blocks (RAMs, shift registers,...); the *control path* generates all the control signals needed to make the circuit work. At the logic design level, the components of the circuit are selected so as to minimize the total cost and maximize performances. Typically, a VLSI circuit is realized by placing on a chip a set of pre-designed circuit components known as *macro-cells*. The third level is an important and complex stage preceding fabrication, as it determines how the circuit will be physically realized. Particularly important is how macro-cells are laid out on the chip and how they are interconnected. Several approaches have been proposed to create a layout of a circuit: full-custom layouts, gate array design style, standard-cell design style, macro-cell, PLA (Programmable Logic Array) and FPGA (Field-programmable Gate-Array). For a full discussion, the interested reader is referred to [SY99]. The problem of finding an optimal layout is strictly related to the space layout planning described in the previous section. It is a complex optimization problem, in that the objective functions to minimize (or maximize) may be in conflict: for example, minimizing the total length of the wires may cause a congestion of wires in certain zones. For this reason, the physical design is subdivided into different subphases; one of these is floorplanning.

VLSI Circuit Floorplanning

At the floorplanning level, the circuit is seen as a collection of interconnected rectangular blocks (the macro-cells), which have to be placed on the chip in such a way that some objective functions are minimized (or maximized). In its easiest formulation, floorplanning asks for an optimal placement of the blocks which minimizes the total area. Unfortunately, this was proved to be a NP-hard problem, except for sliceable floorplans, as said in Section 3.2.1. Other parameters, that should be minimized, are wire length and delay.

The approaches to VLSI floorplanning are traditionally classified into three categories: constructive, iterative and knowledge-based. A *constructive* algorithm tries to create a feasible floorplan starting from a seed module. The subsequent modules are then selected in a predefined order and added one at a time to the floorplan. *Iterative* methods starts from an initial floorplan and perturb it until a feasible one is obtained or no further improvement can be achieved. Finally, a *knowledge-based* approach implements a knowledge expert system, usually consisting of three elements: a knowledge base, a set of rules and an inference engine. The knowledge base contains the data describing the problem and its current state; the set of rules state how data in the knowledge base have to be manipulated to reach an optimal solution; and the inference engine oversees the application of the rules to the knowledge base. Popular algorithms are: cluster growth, simulated annealing and rectangular dualization.

Cluster Growth creates a floorplan adding a module at a time in a greedy fashion. A seed module is placed into the floorplan; each subsequent module addition is a locally optimal solution, which does not necessarily lead to a global optimal solution. The method goes through two steps. First, the order in which modules are added is determined by using linear ordering, which is a well-known NP-hard problem. Second, a topological layout is created. To this extent, several approaches have been proposed. One approach consists in placing the first module in one corner of the floorplan and making the floorplan grow on the upper and right sides, while satisfying shape constraints and other criteria. Other approaches make the floorplan grow row by row.

Simulated Annealing is an iterative method, which starts from a seed floorplan and perturb it in either a direct or indirect fashion. In the first case, the algorithm works directly on the physical layout, thus on the coordinates and sizes of each rectangular module. Modules may overlap in the intermediate solutions. In the second case, the algorithm uses an abstract representation of the floorplan (such as an adjacency graph); the advantage is that any intermediate solution is a feasible solution (modules do not overlap).

Finally, *rectangular dualization*, as cluster growth, is a constructive method. Five steps are necessary. First, the circuit is modelled as a linear graph, whose vertex set represents the set of the modules and the edge set the wires connecting the modules. A planarization procedure is applied if the graph is not planar. Next, the planarized graph must be

modified to comply with the admissibility conditions for rectangular dualization; finally, a rectangular dual is computed. Faces in the rectangular dual represent the macro-cells of the circuit and edges are the interfaces between modules. The advantages of rectangular dualization over the other methods are the following:

- Computing a rectangular dual is relatively easy, if compared, for instance, with simulated annealing.
- The whole procedure can be carried out in linear time.
- There are computationally-feasible algorithms which enumerate all rectangular duals of a given plane graph [TST89]. Knowing all possible solutions can be very helpful for an engineer to decide the best one.

However, rectangular dualization only enforces adjacency constraints. A postprocessing step is needed to fulfil any other requirement.

3.4 Necessary Conditions

As seen in Section 3.1, the vertices of a rectangular dual $\mathscr{R}(G) = (\Gamma, f)$ of a plane graph G have all degree 3, except for the four corner vertices. Consequently, since G is the geometric dual of $\mathscr{R}(G)$, G is a PTG (Fig. 3.7). Moreover, since each face in $\mathscr{R}(G)$ has at least four edges, the degree of each vertex of G is at least 4. These are only necessary conditions; in fact, there are PTGs with minimum degree 4 which are not rectangular graphs.

In Fig. 3.7, G is a multigraph; any multiedge is incident with v_{ext} which is the vertex corresponding to the exterior face of $\mathscr{R}(G)$. The other vertices can not have any incident multiedge, as otherwise two faces of $\mathscr{R}(G)$ would share more than one edge. To get rid of multiedges, we modify G as in Fig. 3.8. Four construction vertices ($North$, $West$, $South$ and $East$) are added to form a new exterior cycle of G. Let $u \in V(G)$ be an inner vertex of G and R be the enclosure rectangle of $\mathscr{R}(G)$; the following properties hold:

- u is adjacent to $North$ if and only if $f(u)$ is incident with the top side of R.
- u is adjacent to $West$ if and only if $f(u)$ is incident with the left side of R.
- u is adjacent to $South$ if and only if $f(u)$ is incident with the bottom side of R.
- u is adjacent to $East$ if and only if $f(u)$ is incident with the right side of the R.

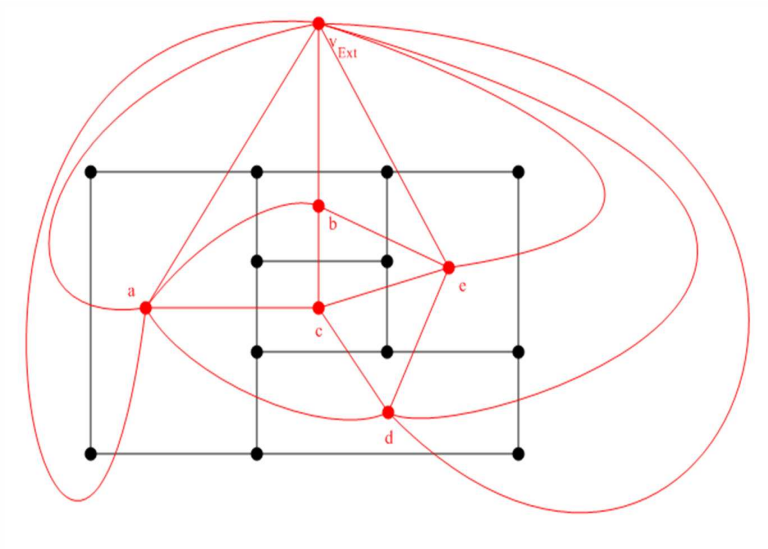


Figure 3.7: The geometric dual G (in red) of a rectangular dual (in black) is a PTG. G may be a multigraph and any multiedge is incident with v_{Ext}.

Henceforth, we assume that G is a PTG with an exterior chordless $4-$cycle ($4-$PTG). The exterior vertices are only construction vertices and they are associated with no rectangle in $\mathscr{R}(G)$.

Let $r_1, r_2 \in \Gamma$ be two adjacent rectangles in $\mathscr{R}(G)$. We define four binary relations on Γ:

- $north(r_1, r_2)$ if and only if r_1 is above r_2.
- $west(r_1, r_2)$ if and only if r_1 is to the left of r_2.
- $south(r_1, r_2)$ if and only if r_1 is below r_2.
- $east(r_1, r_2)$ if and only if r_1 is to the right of r_2.

Accordingly, for any vertex v, $N(v)$ can be partitioned into four subsets:

- $north(v) = \{u \in N(v) | north(f(u), f(v))\}$
- $west(v) = \{u \in N(v) | west(f(u), f(v))\}$
- $south(v) = \{u \in N(v) | south(f(u), f(v))\}$
- $east(v) = \{u \in N(v) | east(f(u), f(v))\}$

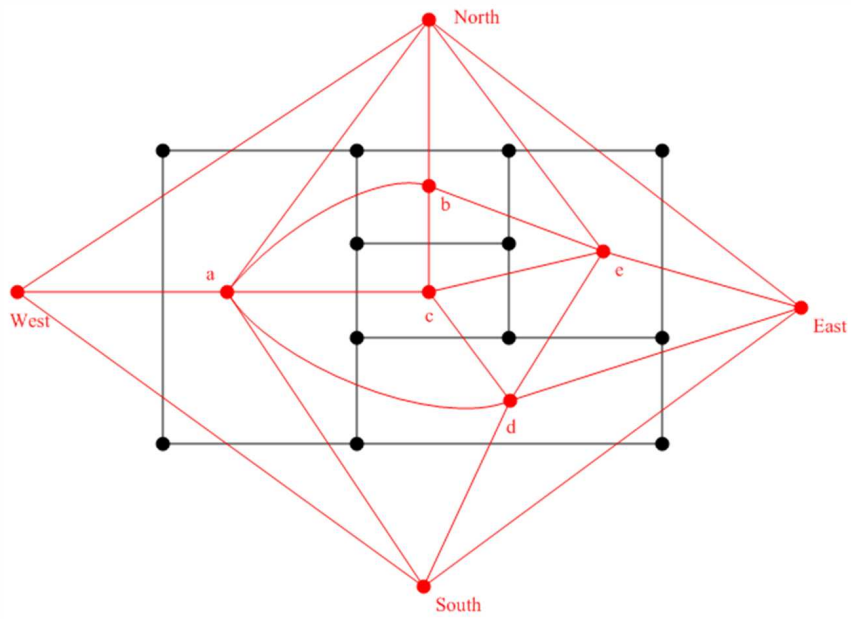


Figure 3.8: To get rid of the multiedges of Fig. 3.7, four exterior vertices replace v_{Ext}.

3.5 Rectangular Dualization as a Matching Problem

A 4−PTG G admits a rectangular dual if and only if the edges of its geometric dual G^* can be drawn as either vertical or horizontal segments. Therefore, the problem of finding a rectangular dual of G reduces to the problem of finding an "assignment of directions" to all edges in G^*.

In [LL90] the direction of the edges of G^* is determined giving each face of G a proper orientation. In $\mathscr{R}$(G), each vertex of degree 3 forms a "T" shape, composed of two 90° angles and one 180° angle. Each such a vertex v is considered to be oriented towards a face f if the 180° angle is incident with face f. Thus, vertex v is assigned to face f. Each exterior vertex of degree 3 is always assigned to the exterior face. The corner vertices form a "L" shape composed of a 90° angle and a 270° angle. Since the 270° angle is always directed towards the exterior face, the corner vertices are always assigned to the exterior face. Dually, this means that in G the inner-faces incident with no exterior vertex are assigned one of the vertices of its frontier. This determines the orientation of the inner edges of G^* and, consequently, the orientation of the exterior ones.

Theorem 3.1 (Lai - Leinwand, 1990)**.** *A 4−PTG G admits a rectangular dual if and only if each inner face of G incident with no exterior vertex can be assigned to one of the vertices of its frontier in such a way that each inner vertex v has exactly $d(v) - 4$ 3−faces assigned*

to it.

This theorem suggests an algorithm for rectangular dualization [LL90]. A bipartite graph $\mathscr{B}(G)$ with classes F_G and V_G is created from G as follows: F_G contains a vertex for each inner face of G incident with no exterior vertex; V_G contains exactly $d(v) - 4$ vertices for each inner vertex v of G; an edge links $f \in F_G$ and $v \in V_G$ if and only if the vertex of G corresponding to v is on the frontier of the face corresponding to f. A perfect matching in $\mathscr{B}(G)$ corresponds to a perfect assignment of faces to vertices (Fig. 3.9).

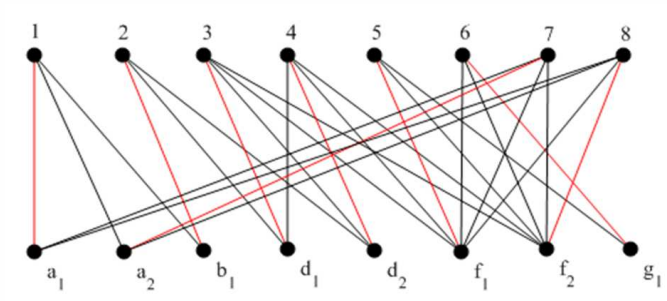


Figure 3.9: The BSP of the graph of Fig. 3.2. As the degree of a is 6, there are two occurrences a_1 and a_2 associated to a. The same hold for the other vertices. The edges in red belong to a perfect matching M. The rectangular dual corresponding to M is shown in red in Fig. 3.2. As $(1, a_1) \in M$, the 180° angle of the T formed by vertex 1 is directed towards a.

The admissibility conditions can be restated as follows.

Theorem 3.2 (Lai - Leinwand, 1989)**.** *A 4−PTG G is a rectangular graph if and only if there is a perfect matching in $\mathscr{B}(G)$.*

3.6 Admissibility Conditions

Theorem 3.1 and 3.2 state the admissibility conditions in terms of region assignments and existence of a perfect matching in a bipartite matching, but do not relate the existence of a rectangular dual to the topology of G. Lai-Leinwand and Kozminski-Kinnen independently gave different (but equivalent) conditions in this sense.

Theorem 3.3 (Kozminski - Kinnen, 1984)**.** *A PTG is a rectangular dual if and only if it admits a 4−completion.*

A 4−*completion* of a graph G is any plane graph H such that graph H', obtained from H by deleting the exterior vertices of H, is isomorphic to G and:

1. H has exactly four exterior vertices of degree at least 3.
2. The degree of all inner vertices of H is at least 4.
3. All faces of H are $k-$faces, with $k \leq 4$.
4. All faces incident with an exterior vertex are triangular.
5. The length of all complex cycles in H is at least 4.

Stated in other words, Theorem 3.3 affirms that G admits a rectangular dual if and only if it can be turned into a $4-$PTG H (as done in the previous section), so that H has no complex triangles.

In rectangular dualization literature, complex triangles are usually referred to as separating triangles. This is due to the fact that in a $4-$PTG a $3-$cycle is separating if and only if it is complex cycle. We now prove it.

Lemma 3.1. *Any simple $4-$PTG is $3-$connected.*

Proof. Let G be a simple $4-$PTG. We create the *face-vertex incidence graph* $FV(G)$ of G as follows (Fig. 3.10):

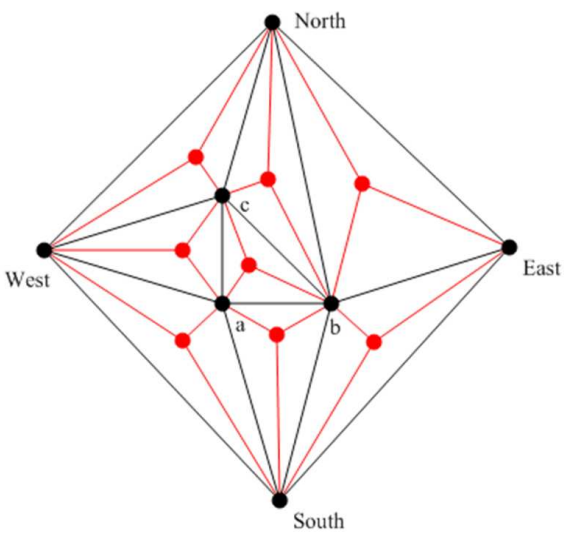


Figure 3.10: A $4-$PTG and its face-vertex incidence graph (FVIG). The vertices of the FVIG are the red and the black ones and the edges are the red ones.

- A new vertex v_f is added within each face f (including the exterior one).

- Each new vertex v_f is linked to the vertices on the frontier of f.
- The edges of G are deleted

In [dFdM01], the following result was proved.

Lemma 3.2. *Let G be a biconnected plane graph. G is $3-$connected if and only if each $4-$cycle of $FV(G)$ is not complex.*

Clearly, G is biconnected. Moreover, all faces of $FV(G)$ are bounded by $4-$cycles. Suppose that there is a complex $4-$cycle (v, f_1, w, f_2), where v and w are vertices of G and f_1 and f_2 are faces of G. This implies that either f_1 or f_2 is not incident with edge (v, w). But then f_1 (or f_2) would not be an inner face of G. If f_1 is the exterior face, (v, w) would be a chord. This contradiction completes the proof. □

The following result is proved in [DDGC97].

Lemma 3.3. *A $4-$PTG G is $4-$connected if and only if:*

1. *G has no complex triangles.*
2. *No interior vertex is connected to two or more non-consecutive vertices on the exterior cycle.*

Remark 3.1. *By Lemma 3.3, it is not true (as stated in [KH94]) that a PTG with no separating triangles is $4-$connected.*

Lemma 3.4. *Any $3-$cycle in a $4-$PTG is separating if and only if it is a complex triangle.*

Proof. Suppose that a non-complex $3-$cycle Δ is separating. Thus, the removal of Δ separates G in at least two components. A vertex may be added within Δ, without destroying planarity. Then, the removal of Δ separates the graph into at least three components. This is a contradiction; in fact, G is $3-$connected (Lemma 3.1) and, as proved in [dFdM01], removing a $3-$cycle in a $3-$connected graph separates the graph into exactly two components. □

Henceforth, we will use the term *separating triangle* instead of *complex triangle*.

Lai and Leinwand were the first who gave the admissibility conditions in terms of separating triangles.

Theorem 3.4 (Lai - Leinwand, 1984)**.** *A PTG admits a rectangular dual if and only if it contains no separating triangles.*

We prefer this formulation of the admissibility conditions, as they are easier to test. Finally, we use the conditions stated by Kant and He [KH94], as they directly refer to 4−PTGs.

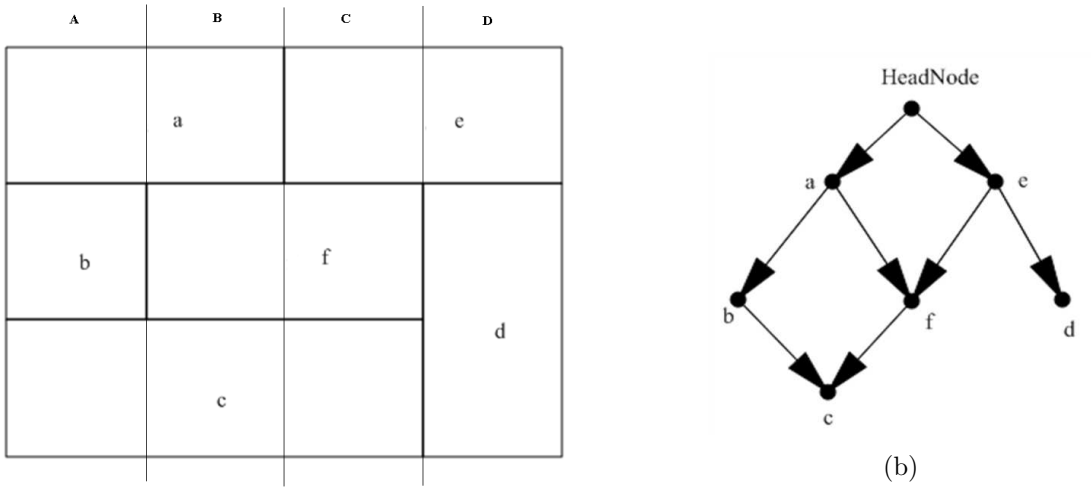
Theorem 3.5 (Kant - He, 1992)**.** *A 4−PTG has a rectangular dual with four rectangles on the boundary if and only if it contains no separating triangles.*

A 4−PTG with no separating triangles is referred to as a *Proper Triangular Planar graph* (PTP) [KH94].

3.7 Algorithms for Rectangular Dualization

As described in Section 3.3.1, the first algorithm for rectangular dualization is due to Grason. More precisely, his algorithm looks for a rectangular cartogram, as the area of each rectangle must equal a predefined value.

Kozminski and Kinnen described a $O(n^2)$-time algorithm based on the admissibility conditions stated in Theorem 3.3 [KK84]. Three major steps can be identified. The first phase checks whether there exists a 4−completion of the input graph G; this is achieved with simple numerical checks, which take $O(n)$ time. The second step creates a 4−completion (if one exists), which again takes $O(n)$ time. Finally, a vertical and horizontal orientation is assigned to each edge of the geometric dual. The 4−completion is divided into smaller 4−completions, using a *divide-and conquer* approach; this corresponds to cutting the rectangular dual into smaller pieces. The orientation of the edges of the dual of each



(a) (b)

Figure 3.11: (a) A rectangular dual with four columns. (b) The corresponding PDG.

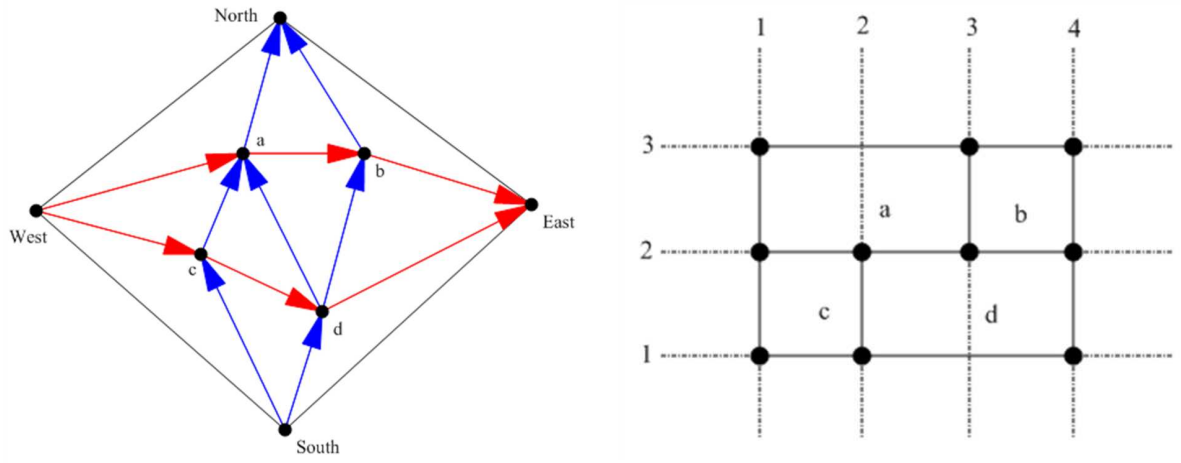
4−completion is determined and the results are then merged along the cut lines. This step requires $O(n^2)$ time. This result was improved to linearity in [KK88].

The Bhasker-Sahni algorithm creates a rectangular dual $\mathscr{R}(G)$ of a PTP graph G from its *path digraph* (PDG) [BS86]. The PDG of $\mathscr{R}(G)$ is obtained by subdividing $\mathscr{R}(G)$ into columns such that no column contains a vertical edge (Fig. 3.11). PDG is a directed graph, containing a node for each rectangle in $\mathscr{R}(G)$; (i, j) is an edge in PDG if and only if $north(i, j)$ in $\mathscr{R}(G)$. A special node, called *HeadNode*, is linked to the nodes corresponding to the rectangles which are north of all other rectangles. A rectangular dual is obtained from a DFS visit of PDG, starting from *HeadNode*. The children of a node are visited from left to right and are assigned proper coordinates. The algorithm works in $O(n)$ time.

Finally, algorithms for enumerating all rectangular duals are described in [KK88, TMSS89].

3.7.1 The Kant-He Algorithm

We describe in more details the Kant-He algorithm [KH94], as we chose to implement it in OcORD. From our perspective, it has two advantages. First, if compared to the others, it is "implementation-friendly"; second, it can be easily generalized to c-rectangular dualization, as we will describe in Chapter 6. A rectangular dual of a PTP graph is



(a) (b)

Figure 3.12: (a) A REL on G. (b) The corresponding rectangular dual. The Kant-He algorithm computes the equations of the four straight lines bounding each rectangle.

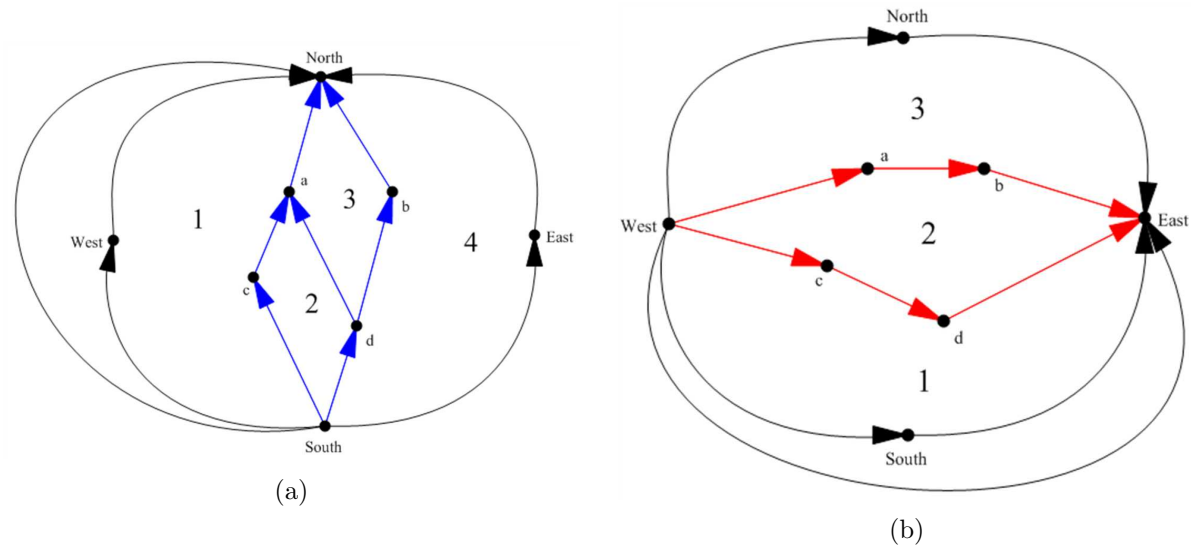
computed from a regular edge labelling (REL) of the edges. In the following definition, notation $(u, v) \in (T, \leftarrow)$ means that (u, v) belongs to set T and is directed from v to u; $(u, v) \in (T, \rightarrow)$ means that (u, v) belongs to T and is directed from u to v.

Definition 3.2. *A* regular edge labelling $REL(G) = (T_1, T_2)$ *of a PTP graph* G *is a partition of the inner edges of* G *into two subsets* T_1 *and* T_2 *of directed edges such that (Fig. 3.12):*

1. *For each inner vertex* v*, the edges incident with* v *appear in counterclockwise order around* v *as follows: a set of edges* $(v, u) \in (T_1, \rightarrow)$*; a set of edges* $(v, u) \in (T_2, \leftarrow)$*; a set of edges* $(v, u) \in (T_1, \leftarrow)$*; a set of edges* $(v, u) \in (T_2, \rightarrow)$*;*
2. *For each node* v*:* $(v, North) \in E(G) \Rightarrow (v, North) \in (T_1, \rightarrow)$*;* $(v, West) \in E(G) \Rightarrow (v, West) \in (T_2, \leftarrow)$*;* $(v, South) \in E(G) \Rightarrow (v, South) \in (T_1, \leftarrow)$*;* $(v, East) \in E(G) \Rightarrow (v, East) \in (T_2, \rightarrow)$*.*

Two edges $e_1 = (v, u)$ and $e_2 = (w, u)$ incident with a common vertex u have the *same label* in a REL if they are both in T_1 or T_2 and are both leaving u or entering u. We denote this as $(v, u) \sim_{(T_1,T_2)} (w, u)$.

In [KH94], Kant and He proposed two different algorithms to compute a REL of a PTP graph in linear time. We omit the description of these methods and we rather concentrate on how a rectangular dual $\mathscr{R}(G) = (\Gamma, f)$ is created from (T_1, T_2). Every rectangle in $\mathscr{R}(G)$ is bounded by four straight lines. What we need is to find the equations of these lines. To this extent, two directed graphs G_1 and G_2 are created from T_1 and T_2 respectively (Fig. 3.13). G_1 is the graph which consists of the edges of T_1, the four exterior



(a) (b)

Figure 3.13: (a) G_1. (b) G_2. In both graphs, each face is labelled with its maximum distance from the source face (the one with no label). Since $left(a) = 1$, $right(a) = 3$, $below(a) = 2$ and $above(a) = 3$, the rectangle $f(a)$ is bounded by lines $x = 1$, $x = 3$, $y = 2$, $y = 3$ (Fig. 3.12 (b))

edges (directed as $(South, West)$, $(West, North)$, $(South, East)$ and $(East, North)$) and

a new edge $(South, North)$. G_2 is composed of the edges in T_2, the four exterior edges (directed as $(West, South)$, $(South, East)$, $(West, North)$ and $(North, East)$) and a new edge $(West, East)$. For any vertex v in G_1 and G_2, there is exactly one face which separates the incoming from the outgoing edges of v in clockwise direction; moreover, there is only one face which separates the outgoing from the incoming edges in clockwise direction. In

(a) (b)

Figure 3.14: (a) $left(v)$ and $right(v)$. (b) $above(v)$ and $below(v)$.

G_1, the first face is denoted as $left(v)$ and the second one as $right(v)$; in G_2, the first face is denoted as $above(v)$, the second as $below(v)$ (Fig. 3.14). Thus, each vertex is incident with four special faces. The algorithm now computes an integer coordinate for each such face. Two graphs G_1^* and G_2^* are created. G_1^* is the left-oriented geometric dual of G_1, where the direction of the dual edge of $(South, North)$ is reversed. Similarly, G_2^* is the right-oriented geometric dual of G_2, where the direction of the dual edge of $(West, East)$ is reversed. For each node v of G_1^*, the length $d_1(v)$ of the longest path between the right face of edge $(South, North)$ and v is computed; similarly, for each node w of G_2^*, the length $d_2(v)$ of the longest path between the left face of $(West, East)$ and v is computed. For each node v, $(d_1(left(v)), d_2(below(v)))$ and $(d_1(right(v)), d_2(above(v)))$ are respectively the coordinates of the left bottom vertex and the right top vertex of rectangle $f(v)$.

Theorem 3.6 (Kant - He, 1992)**.** *Let G be a PTP graph and $\{T_1, T_2\}$ be a REL of G. For each vertex v of G, we assign v the rectangle $f(v)$ bounded by the four lines $d_1(left(v))$, $d_1(right(v))$, $d_2(above(v))$ and $d_2(below(v))$. Then the set $\{f(v)|v \in V(G)\}$ form a rectangular dual of G.*

Lemma 3.5. *Let v and w be two adjacent nodes in G. The following hold:*

1. *$(v, w) \in (T_1, \rightarrow)$ if and only if $north(w, v)$ (and $south(v, w)$)*
2. *$(v, w) \in (T_2, \rightarrow)$ if and only if $west(v, w)$ (and $east(w, v)$).*

Proof. It is trivial to verify that $(v, w) \in (T_1, \rightarrow)$ if and only if $above(v) = below(w)$ and $(v, w) \in (T_2, \rightarrow)$ if and only if $right(v) = left(w)$. The thesis follows from Theorem 3.6. □

3.8 Our Approach

In this section we describe an algorithm which creates a rectangular dual of any plane graph, after enforcing the admissibility conditions [ADQB07, AQP09]. The algorithm runs in linear time and aims at modifying the input graph as little as possible.

Let G be a plane connected graph. Our algorithm goes through the following steps:

1. **Biconnectivity Test and Augmentation**. Some edges are added to G to make it biconnected, if not yet.
2. 4−**completion**. Four vertices $North$, $West$, $South$ and $East$ are added to form a new exterior cycle of G. These nodes are connected to the vertices lying on the former exterior cycle.
3. **Separating Triangle search and break**. Any separating triangle of G is eliminated ("broken", using the notation in [LL88]).
4. **Triangulation**. The inner faces of G are triangulated in linear time [BKK94].After this step, G is a rectangular graph.
5. **Rectangular Dualization**. A rectangular dual of G is created, using the linear-time algorithm described in Section 3.7.1.

The output $\mathscr{R}(G) = (\Gamma, f)$ of the algorithm is not properly the rectangular dual of G (Fig. 3.15). In fact, some rectangles may turn out to be adjacent, though the corresponding

(a) (b)

Figure 3.15: (a) A plane graph G with one separating triangle $\Delta = (a, b, c)$. (b) The rectangular dual of G. C_1 is a gate, corresponding to the crossover vertex added on edge (a, c) to break Δ. Rectangles a and c are adjacent through C_1. Rectangles a and d are adjacent, although their corresponding vertices are not. Edge (a, d), in fact, has been added to triangulate face (a, d, b, c).

vertices are not. However, if two vertices are adjacent in G, the corresponding rectangles

are adjacent too, provided that G has no separating triangles. Indeed, breaking a separating triangle $\Delta = (u, v, w)$ is accomplished by removing one of its edges (say (u, v)), adding a new vertex c (called *crossover vertex*) and linking c to u and v. Thus, because of the crossover vertex, u and v are not adjacent anymore. Consequently, in $\mathscr{R}(G)$ rectangles $f(u)$ and $f(v)$ will be adjacent only through rectangle $f(c)$ (called *gate*). Thus, we allow the following weaker definition of rectangular dual.

Definition 3.3. *A* (weak) rectangular dual *of a plane connected graph G is a pair (Γ, f) such that:*

1. *$\Gamma = \{R_1, R_2, ..., R_n\}$ is a set of simple non-overlapping rectangles which form a partition of a rectangle R (called* enclosure rectangle*);*
2. *$f : V(G) \rightarrow \Gamma$ is a one-to-one function associating each vertex of G to a distinct rectangle in Γ;*
3. *If two vertices u and v are adjacent in G, then either $f(u)$ and $f(v)$ share an edge or part of it, or they are adjacent to the same gate.*

Not only does adding crossover vertices break the adjacencies of the input graph, but also increases the area of $\mathscr{R}(G)$, as each crossover vertex is mapped to a gate. Thus, it is important to minimize the number of crossover vertices used to break all separating triangles. We will analyze in depth this problem in the next chapter.

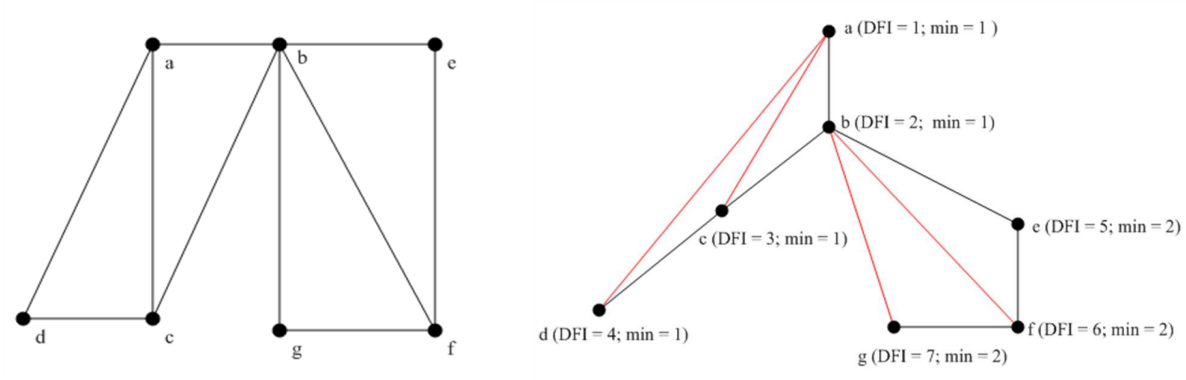
3.8.1 Biconnectivity Test

A graph is biconnected if and only if it has no cutvertex. Therefore, testing the biconnectivity of G reduces to finding any cutvertex in G. The brute force algorithm is the easiest way to accomplish this. For each vertex v, if $G - \{v\}$ is disconnected, then v is a cutvertex. However, this approach has at least two drawbacks. First, it requires $O(n^2 + nm)$ time, as each connectivity test requires $O(n + m)$ time. Second, it does not find the blocks of G.

A better algorithm uses the DFS tree of G. A *DFS tree T* is created while performing a DFS visit of G as follows (Fig. 3.16):

- For each node v, a node t_v is added to T.
- Let v be the currently visited vertex. If there is an edge $e = (v, u)$ such that u has not been visited yet, add an edge $t_e = (t_v, t_u)$ to T. t_v is considered as the father of t_u in T. t_e is called a *tree edge*.

- If no edge connects v to a non-visited vertex, for each edge e, which has not been traversed yet, incident with v, an edge t_e is virtually added to T and is called *back edge*.



(a) (b)

Figure 3.16: (a) A connected component with two blocks. (b) The DFS tree created from a DFS visit starting from vertex a. Back edges are in red. Since $min(e) = DFI(b)$, b is a cutvertex.

The vertices of T are assigned a number (the DFS Index, DFI in short), based on the order in which the corresponding nodes in G are discovered in the DFS visit. Therefore, the root of T has $DFI = 1$, the second node visited by DFS has $DFI = 2$ and so on (Fig. 3.16). The cutvertices of G are then characterized in terms of DFI and the following properties.

Algorithm 3.1 Biconnectivity Test

```
1: function isBiconnected(G: Graph): (boolean, list of blocks)
2:     if |V(G)| = 1 then
3:         return (true, [])
4:     else
5:         T: DFS tree, v: vertex of G
6:         add t_v to T
7:         return DfsTree(G, T, v, 0)
8:     end if
9: end function
```

Lemma 3.6. *In a DFS tree T, the following properties hold:*

1. *If the root of T has two or more incident edges, then the corresponding node in G is a cutvertex.*

2. *If a node t_v in T, different than the root, has a child t_u such that no back edge links a descendant of t_u to an ancestor of t_v, then v is a cutvertex.*

We then define, for any vertex t_v, the *lowest ancestor* of t_v ($low(t_v)$) as the ancestor of t_v with the lowest DFI reachable from v with a path containing at most one back edge and zero or more tree edges of the subtree rooted at t_v. Thus, each vertex t_v is assigned another number $min(t_v) = DFI(low(t_v))$ (Fig. 3.16). By Lemma 3.6, v is a cutvertex if t_v has a child t_w such that $min(t_w) \geq DFI(t_v)$. Since DFS visits the vertices and the edges of each block consecutively, a stack is used to keep track of the biconnected component being traversed. If G is not biconnected, Function $isBiconnected(G)$ (Algorithm 3.1) returns the list of the blocks in G; each block is a list of edges. In function $DfsTree$ (Algorithm 3.2), $pop(S)$ returns the first item in stack S (and removes it from S); $push(e, S)$ inserts e as the first item of S.

The cost of the algorithm is dominated by the creation of the DFS tree, which is accomplished in linear time.

3.8.2 Planar Biconnectivity Augmentation

The Planar Biconnectivity Augmentation (PBA) problem asks for adding a minimum number of edges to G to make it biconnected, while maintaining G planar. PBA has been shown to be NP-complete in [KB91]; some approximated algorithms have been given in [KB91, FM98].

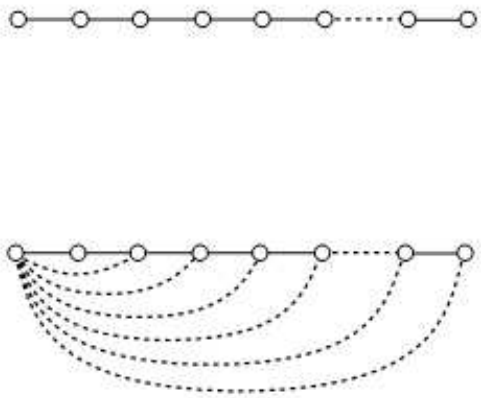


Figure 3.17: A vertex which receives $O(n)$ extra edges. Taken from [Kan93].

Algorithm 3.2 DFS Visit

```
1: function DfsTree(G: graph, T: tree, v: Node, dfi: integer, CutV: integer)(()boolean, list of Blocks)
2:     S: stack, LB: list
3:     CutV ← 0; dfi ← dfi + 1; DFI(v) ← dfi; min(v) ← dfi
4:     for all edge e = (v, w) incident with v do
5:         if e has not been visited yet then
6:             insert(e, S)
7:             Add t_w and edge (t_v, t_w) to T
8:             DfsTree(G, T, w, dfi, CutV)
9:             if min(w) ≥ DFI(v) then
10:                CutV ← CutV + 1
11:                do e_1 ← pop(S); BackInsert(e_1, L); while(e_1 ≠ e)
12:                BackInsert(L, LB)
13:            else
14:                min(v) ← minimum(min(v), min(w))
15:            end if
16:        else
17:            if DFI(w) < DFI(v) and w is not parent of v in T then
18:                push((v, w), S)
19:                min(v) ← minimum(DFI(w), min(v))
20:            end if
21:        end if
22:    end for
23:    if CutV > 0 then return (false, LB)
24:    else
25:        return (true, [])
26:    end if
27: end function
```

However, minimizing the number of edges is not our primary concern. In fact, adding an edge between two vertices u and v will cause rectangles $f(u)$ and $f(v)$ to be adjacent in $\mathscr{R}(G)$, but will not increase the area of $\mathscr{R}(G)$. For this reason, we implemented the algorithm described in [Rea87], with the improvements proposed in [Kan93]. The idea is trivial: if u and w are consecutive in $N(v)$ and belong to different blocks, then edge (u, w) is added. To make G biconnected, this procedure is applied to all consecutive neighbours of all vertices of G. However, as noted in [Kan93], the degree of a single vertex may receive $O(n)$ new incident edges (Fig. 3.17). The following improvements are then applied:

- The embedding of G is changed so that all neighbours of v, belonging to the same block, are consecutive in $N(v)$.
- The vertices are visited in DFS order.
- If v has $d(v)$ blocks, then adding $d(v) - 1$ edges between the blocks collapses the blocks into one.

With these modifications, procedure BICONNECT (Algorithm 3.3) makes G biconnected in $O(n)$ time, while adding to each vertex at most 2 extra edges [Kan93].

Algorithm 3.3 Biconnectivity Augmentation

```
1: procedure BICONNECT(G: Graph)
2:     create an embedding of G such that all neighbours of each vertex v, belonging to
   the same block, appear consecutively in Adj(v)
3:     for all cutvertex v ∈ V(G) (visited in DFS order) do
4:         for all w ∈ Adj(v) do
5:             if u ← Next(w, Adj(v)) belongs to a different block then
6:                 p_w ← Prev(v, Adj(w))
7:                 p_u ← pointer to the occurrence of v in Adj(u)
8:                 InsertEdge(G, w, u, p_w, p_u)
9:                 if (v, w) was added to G earlier then
10:                    DeleteEdge(G, (v, w))
11:                end if
12:                if (v, u) was added to G earlier then
13:                    DeleteEdge(G, (v, u))
14:                end if
15:            end if
16:        end for
17:    end for
18: end procedure
```

Algorithm 3.4 Procedure finding the separating triangles

```
1: function FINDST(G: Graph) : list of separating triangles
2:     L: list
3:     for all v ∈ V(G) in decreasing degree do
4:         Mark all vertices in Adj(v)
5:         for all u ∈ Adj(v) do
6:             for all w ∈ Adj(u) do
7:                 if w is marked then
8:                     BackInsert((v, w, u), L)
9:                 end if
10:            end for
11:            Unmark u
12:        end for
13:        Delete v
14:    end for
15:    F ← 3−faces of G
16:    Sort L and F by lexicographic order
17:    Delete from L all triples belonging to F
18: end function
```

3.8.3 Searching for Separating Triangles

In [LL88], Lai and Leinwand describe a $O(n^2)$ procedure to enumerate all separating triangles in G. They base upon the idea proposed in [ST81], which enumerates all cycles in a graph by means of *chord edges* [5]. The procedure first creates a spanning tree (which requires $O(n)$ time); a chord edge (u, v) is on the boundary of a separating triangle if there are two edges (u, w) and (v, w) such that any two of u, v and w are not consecutive in the adjacency list of the third vertex. Therefore, for any chord edge, the adjacency lists of its endpoints are intersected in $O(n)$ time. Since there are $O(m)$ chord edges, the total cost is $O(n^2)$.

We used a different approach (Algorithm 3.4). A separating triangle is a 3−cycle which is not a face. In [CN85] a simple nice linear-time algorithm for finding all 3−cycles is described. The nodes of G are sorted by decreasing degree, which can be accomplished using Bucket Sort, a well-known linear-time sorting algorithm. For each node v the following operations are performed: first, the neighbours of v are marked, then, for each marked node w, if w is adjacent to a marked node u, then (v, w, u) is a 3−cycle. After visiting all its neighbours, v is removed from G, so that any separating triangle is found only once.

Each vertex of G can be assigned a natural number; consequently, we can assume that

[5] Edges not belonging to a spanning tree.

each triple (v, w, u) in L is such that $v < w < u$. These triples can be lexicographically sorted using Radix Sort, another linear-time sorting algorithm. Finally, a list F of $3-$faces is created and lexicographically sorted in the same way. We remark that finding all faces of a plane graph can be trivially accomplished in linear time. Since L and F are sorted, they can be searched in parallel (thus in linear time) to remove from L any item of F.

Chapter 4

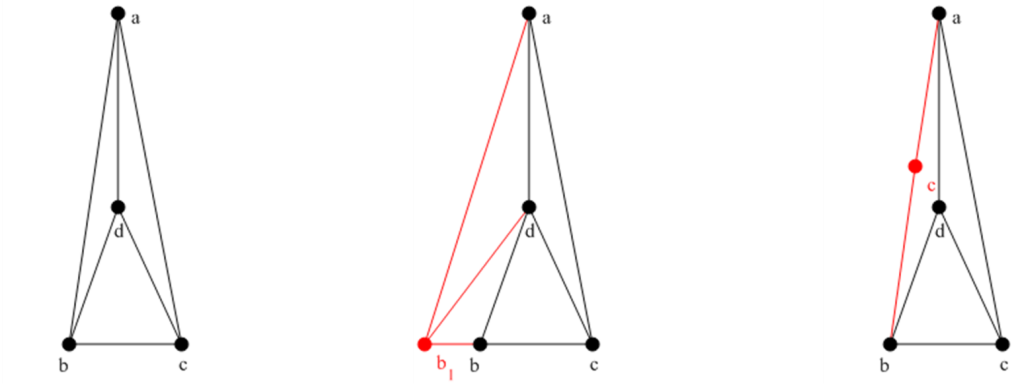
The Curse of the Separating Triangles

In this chapter we present the Optimal Separating Triangle Breaking (OSTB) problem and we propose some algorithms to solve it. Given a plane graph G, OSTB asks for a minimal set O of edges such that each separating triangle has at least one edge in O. Adding a crossover vertex on each edge of C *breaks* all separating triangles in G, while minimizing the number of crossover vertices. We present a detailed characterization of OSTB in terms of well-known problems in graph theory. Finally, we describe easy and efficient heuristic algorithms.

4.1 Introduction

In order to break the separating triangles (ST in the following) in a plane graph G, two approaches have been proposed: *vertex split* and *break edge* (Fig. 4.1). *Vertex split* was proposed in [He97] to create rectilinear duals of G with complexity at most 8. Splitting a vertex v of a ST $\Delta = (u, v, w)$ turns Δ into a complex 4−cycle (Lemma 2.6). The *break edge* approach consists in adding a *crossover vertex* c on an edge of Δ (say (u, v)), replacing (u, v) with edges (u, c) and (c, v). Also in this case, Δ is turned into a 4−complex cycle. We adopt the second approach, as more intuitive. *Vertex split* is advantageous only when it comes to create a rectilinear dual; in fact, only using *vertex split* the obtained rectilinear dual is guaranteed to have complexity at most 8 [He97].

In the previous chapter we have seen that each new crossover vertex increases the area of the rectangular dual of a plane graph G. Since G may have $O(n)$ STs (Section 4.2), adding a crossover vertex for each ST is not a satisfactory solution. Often do STs share one edge,



(a) (b) (c)

Figure 4.1: (a) A ST Δ. (b) Δ expanding (splitting) vertex b. (c) Δ is broken adding a crossover vertex c on (a, b).

and in this case we say that they are *adjacent* (Fig. 4.2) ; thus, just one crossover vertex may be enough to break many STs.

The Optimal Separating Triangle Breaking (OSTB) problem can be stated as follows:

> Optimal Separating Triangle Breaking (OSTB):
> *Input*: A plane graph $G = (V, E)$
> *Output*: A minimal set O of edges of G such that each ST has at least one edge in O.

The title of this chapter sounds a bit dramatic, but effectively emphasizes the negative impact of STs on rectangular dualization. OSTB is the only step of the algorithm described in Section 3.8, for which no linear time algorithm has been given yet. It was first deemed as a NP-hard problem [LL88, RBM01], as, indeed, it seems very difficult to reduce OSTB to tractable problems. The only known polynomial-time algorithm [AAV00] maps the problem into a minimum edge covering in a hierarchically clustered graph. However, since one of the goals of this thesis is to prove the effectiveness of rectangular dualization in graph visualization, and most graph drawing algorithms run in linear time, we need a linear time algorithm solving OSTB or giving an acceptable approximate solution.

Besides rectangular dualization, another application of OSTB is *connectivity augmentation*. It is a fundamental graph-theoretic problem, which received a lot of attention amongst researchers [Hsu00, Jor93, Jor95]. The aim is to add a set of edges to a graph to reach a given degree of connectivity. The problem becomes harder if the set of edges to add is required to be the smallest one, or if the augmentation has to preserve the planarity of the input graph. The first problem is sometimes referred to as the *smallest augmentation*

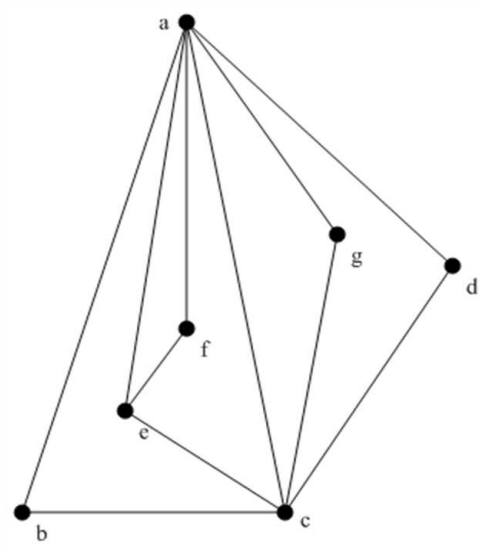


Figure 4.2: Three STs incident with edge (a, c). (a, e, c) is enclosed in (a, b, c)

problem, the second one as the *planar augmentation* problem. Many contexts require a graph to have a given connectivity degree. One trivial example has been shown in the previous chapter; the algorithm that forces a graph to admit a rectangular dual, in fact, requires the input graph to be biconnected. Moreover, a graph may be used to represent a communication network: the higher is the connectivity degree, the more reliable is the network [JG86]. Finally, also statistical data security exploits connectivity augmentation [Gus88].

4−connected planar graphs are particularly attractive, as they have a lot of interesting properties. In 1956, Tutte proved that they are Hamiltonian [Tut56]. In 1989, Chiba and Nishizeki described a linear time algorithm that finds all Hamiltonian cycles in them [CN89]. Hamiltonian graphs have interesting properties, which are discussed in [Bro02]. Furthermore, a 4−connected maximal planar graph fulfils the rectangular dualization admissibility conditions. Finally, many graph drawing algorithms easily handle 4−connected graphs [KH94, EK05, He96]. There are several algorithms augmenting 3−connected graphs to 4−connectivity. The one described in [Hsu00] was the first to add a smallest set of edges, while running in linear time. Planarity, however, may not be preserved, as required by most graph drawing algorithms. The approach in [EK05, He96, KH94] turns the input graph into a maximal planar graph while breaking all STs; in fact, every maximal planar graph is four-connected if and only if it contains no STs [dFdM01].

4.2 More on Separating Triangles

In this section we introduce useful definitions and properties of STs.

First of all, we estimate the number of crossover vertices needed to break all STs. The worst case occurs when all STs are *independent*, that is no two of them are adjacent. In this case, in fact, a crossover vertex is added for each ST. When looking for an upper bound on the number σ of STs in G, we must be aware that a ST can be enclosed (or *nested*) in other STs (Fig. 4.2).

Definition 4.1. *Let Δ_1 and Δ_2 be two STs in a plane graph G. If $I(\Delta_1) \subset I(\Delta_2)$, then Δ_1 is* enclosed *in Δ_2.*

Let us denote as $\mathscr{I}(\Delta)$ the set of the STs enclosed by Δ. If a vertex v is enclosed in Δ and in no other triangle of $\mathscr{I}(\Delta)$, v is called a *child* of Δ (and Δ is the *father* of v). Similarly, Δ_1 is a child of Δ (and Δ is the father of Δ_1) if it is enclosed in Δ and in no other ST of $\mathscr{I}(\Delta)$.

Lemma 4.1. *Every vertex (and every ST) in G has at most one father.*

Proof. Suppose that a vertex $v \in V$ has two distinct fathers Δ_1 and Δ_2. $v \in I(\Delta_1)$ and $v \in I(\Delta_2)$. Thus $I(\Delta_1) \cap I(\Delta_2) \neq \emptyset$. Since G is plane, Δ_1 and Δ_2 can not partially overlap. Thus, either $\Delta_1 \in \mathscr{I}(\Delta_2)$ or $\Delta_2 \in \mathscr{I}(\Delta_1)$. In any case, v can not be the child of both Δ_1 and Δ_2. □

The hierarchy of STs in G is usually described by a rooted tree T (called the $\mathcal{ST}$*-tree*), defined as follows (Fig. 4.3):

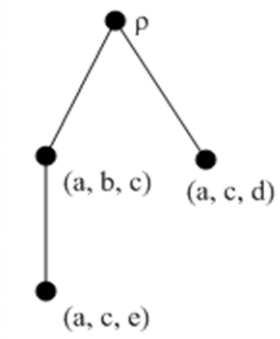


Figure 4.3: The $\mathcal{ST}-$tree of the graph shown in Fig. 4.2.

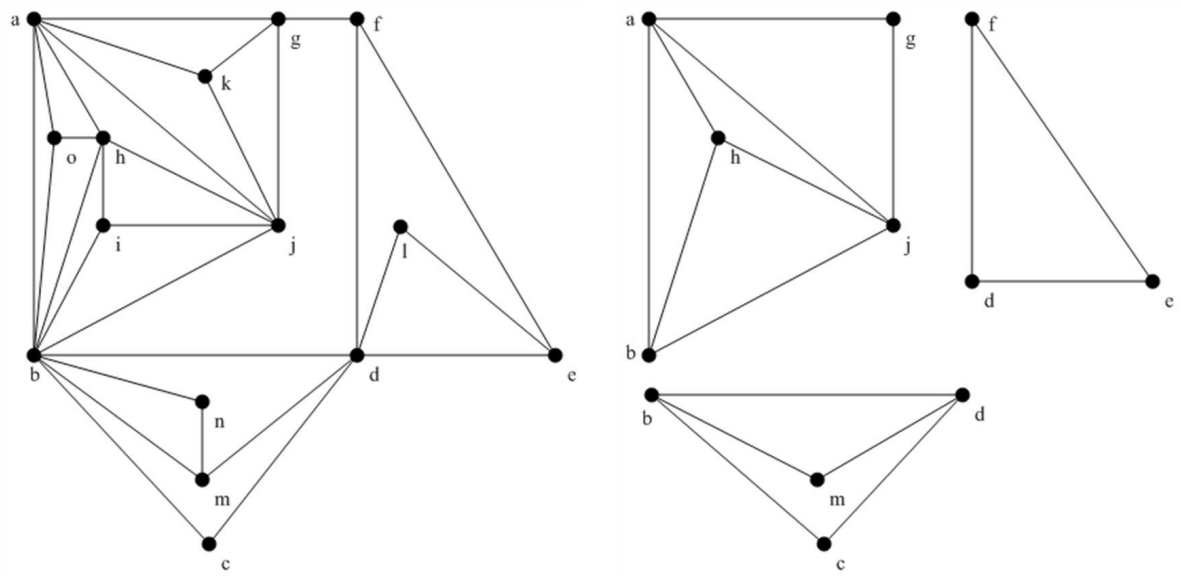
- ρ is the root of T.
- For every ST Δ, T has a node t_Δ.
- For every ST Δ, enclosed in no ST, T has an edge (ρ, t_Δ)
- For every child Γ of Δ, T has an edge (t_Δ, t_Γ).

If $d_T(\rho, t_\Delta)$ denotes the distance of t_Δ from ρ in T, $d(\rho, t_\Delta) - 1$ is the *nesting level* of Δ and its children vertices in G. If k is the maximum nesting level of the STs of G, G is called a *$k-$level containment graph.*

Lemma 4.2. *The number of STs in G is $O(n)$.*

Proof. Every ST Δ has at least one child. In fact, it encloses at least one vertex v and either this is a child or there is a triangle enclosing v which is enclosed by Δ but in no other triangle enclosed by Δ. But now at least one vertex of this triangle is a child of Δ. Since each ST has at most one father, $\sigma \in O(n)$. □

To simplify the solution to OSTB, we introduce the concept of *island* (Fig. 4.4 (a)), which



(a) (b)

Figure 4.4: (a) A plane graph with three islands. I_1 is composed of triangles (a, b, j), (b, h, j), (a, g, j) and (a, b, h); I_2 contains (b, c, d) and (b, d, m); I_3 contains (d, e, f) (b) The three islands as graphs. This representation is ambiguous, as it is not clear which $3-$cycles correspond to STs in G: for instance, (a, b, h) is a ST, while (a, h, j) is not.

is similar to the definition of *connected component*. We denote as $\mathcal{ST}$ the set of all STs in G.

Definition 4.2. *A* $\mathcal{ST}$-path *is a sequence* $\Pi = (\Delta_1, e_1, \Delta_2, ..., e_{r-1}, \Delta_r)$, $\Delta_i \in \mathcal{ST}$, *for* $1 \leq i \leq r$, *and* $e_j \in E(G)$ *for* $0 < j < r$, *such that* Δ_i *and* Δ_{i+1} *share edge* e_i, *for* $0 < i < r$. Π *is said to* join Δ_1 *and* Δ_r.

Definition 4.3. *Any set* $S \subseteq \mathcal{ST}$ *is* $\mathcal{ST}$-connected *if for every pair of STs* Δ_1 *and* Δ_2 *in* S *there is an ST-path joining* Δ_1 *and* Δ_2.

Definition 4.4. *An* island S *is a maximal* $\mathcal{ST}$*-connected set of STs.* $|S|$ *is the* size *of the island.*

As such, this definition is useless. We prefer to think of an island as a graph, rather than a set. To this extent, given an island S, we create graph I (also called island) by deleting the edges incident with no ST in the subgraph induced by the vertices of the STs in S (Fig. 4.4). We observe that a solution to OSTB in one island is independent of the solutions in the other islands. Therefore, without losing generality, we can assume that G contains just one island of STs.

4.2.1 The Bubble Graph

It is difficult to immediately understand by visual inspection how many STs an island I contains. Estimating the maximum nesting level of I is even more challenging. Moreover, I may contain some 3−cycles that do not correspond to any ST of G (Fig. 4.4 (b)).

For this reason, we devised a different (but equivalent) visualization called *bubble graph* (Fig. 4.5). The bubble graph $B(I)$ of I is the graph $B(I)$, which contains a node v_e for each edge e in I and an edge (v_e, v_f) if and only if e and f are on the frontier of the same ST. A 3−cycle representing a ST in $B(I)$ is visualized as a circle, or *bubble*, hence the name of the graph (Fig. 4.5 (b)). Thus, a 3−cycle is a ST if and only if it is represented as a bubble. In this way, no ambiguity arises. Once again, we remark that visualization matters!

In $B(I)$, OSTB is restated as follows:

> Optimal Separating Triangle Breaking (OSTB):
> *Input*: A bubble graph $B(I)$, I being an island of STs
> *Output*: A minimal set O of vertices of $B(I)$ such that each bubble is incident with at least one vertex in O.

The reader should not confuse $B(I)$ with the line graph $L(I)$, in which each vertex represents an edge of I, but two vertices are adjacent if and only if their corresponding edges

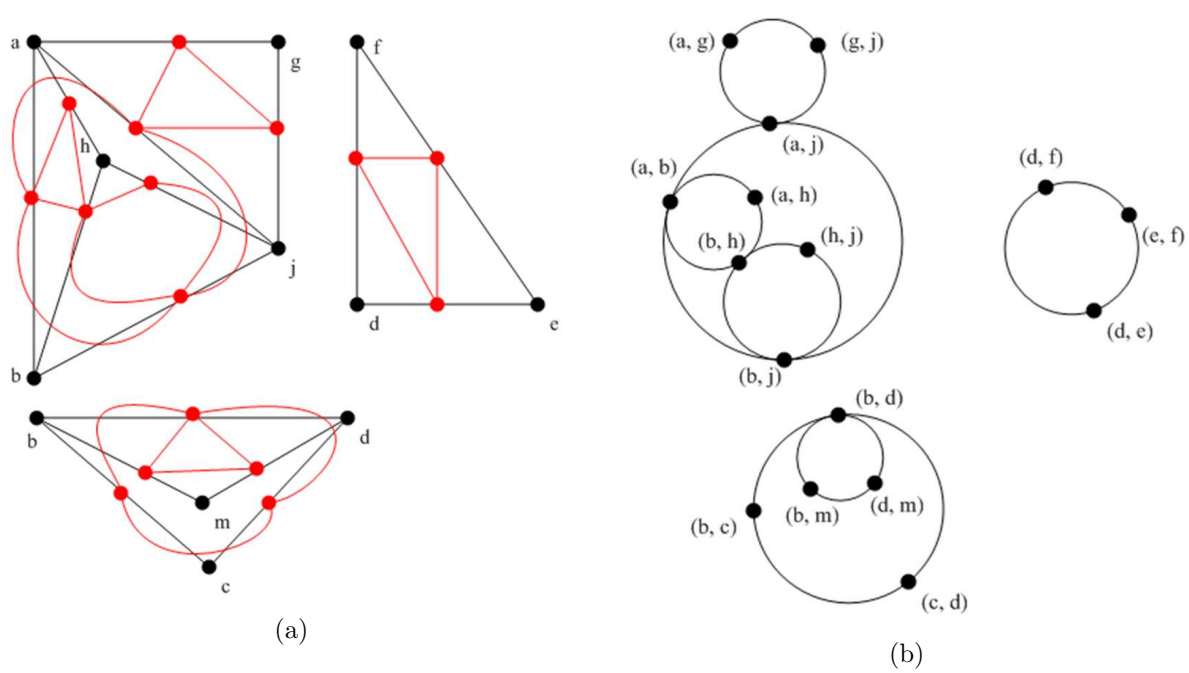


Figure 4.5: (a) The bubble graph (in red) built on the islands shown in Fig. 4.4. (b) Visualization of the bubbles as circles. $((a,b),(b,h),(b,j))$ is a $3-$cycle but not a bubble. Thus, it does not represent a ST.

share an endpoint. Thus, referring to Fig. 4.5, vertices (h,j) and (g,j) are adjacent in $L(I)$, but are not adjacent in $B(I)$.

Lemma 4.3. *If I is a simple plane island, the following properties hold:*

1. ***
2. *$B(I)$ is plane and connected.*
3. *The degree of each vertex of $B(I)$ is even.*
4. *Two distinct bubbles of $B(I)$ share at most one vertex.*
5. *Let $\beta = (u,v,w)$ be a bubble in $B(I)$. At least two vertices on the frontier of β have the same nesting level.*

Proof. 1. By construction and by the fact that no two STs partially overlap.

2. Each vertex v has two incident edges for each bubble incident with v.

3. If two bubbles shared two vertices, the corresponding STs in I would share two edges, thus they would be coincident.

4. Let α be the father of β. If α and β do not share any vertex, all vertices of β are children of α and have the same level. If α and β share vertex u, then v and w are children of α and have the same level. Since two bubbles share at most one vertex, the proof is complete.

□

Lemma 4.4. *If I is a simple graph, then $B(I)$ contains no clique K_n, $n > 3$.*

Proof. The planarity of $B(I)$ implies that $n < 5$ (Theorem 2.1). Suppose that $B(I)$

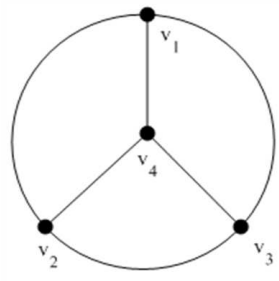


Figure 4.6: A K_4 in the bubble graph

contains a K_4 (Figure 4.6). $\alpha = (v_1, v_2, v_3)$ is a bubble corresponding to a ST in I. At least one of the 3−cycles enclosed in α is a bubble, otherwise there would not be vertex v_4. But each of these 3−cycles share two vertices with α, which contradicts Lemma 4.3. □

4.3 Previous Approaches

In 1988, Lai and Leinwand conjectured that OSTB was NP-complete [LL88]. This conjecture conditioned the subsequent algorithms, which indeed looked for approximated solutions. Tsukiyama *et al.* [TKS86] described an algorithm that turns OSTB into a bridge-connected augmentation problem, a NP-complete problem for which some approximation algorithms are known [LR01].

In [SY93] Sun and Yeap proved the NP-completeness of a variant of OSTB called *weighted OSTB* (WOSTB). Given an integer $B > 0$ and a PTG H with positive weight on each edge, WOSTB asks for a set $E' \subseteq E(H)$ such that the sum of the weights of the edges in E' is at most B. At the same time, they described a $O(n^{1.5})$ algorithm which solves OSTB

in a 0−level containment graph G. They proved, in fact, that the solution to OSTB is a minimum edge covering in a graph, which we call the $ST-$graph. The $ST-$graph $S(I)$ of an island I contains a vertex v_Δ for each ST Δ and an edge between two nodes if and only if the corresponding STs are adjacent (Fig. 4.7). Clearly, $S(I)$ is plane and connected. The algorithm first computes a maximum matching M on $S(I)$; as seen in Chapter 2.9, the

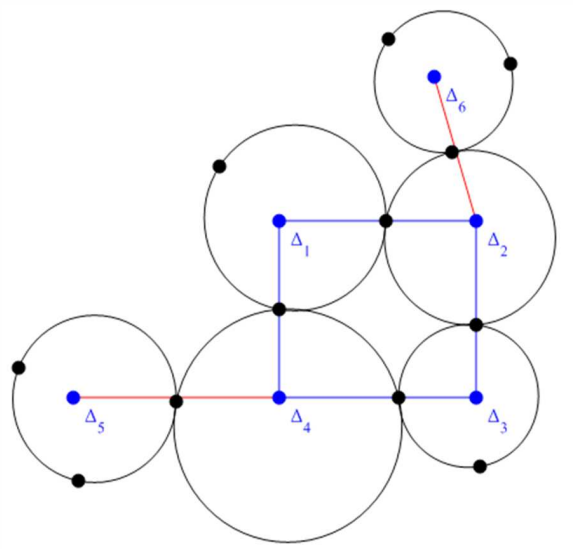


Figure 4.7: An island of STs (in black) and the corresponding $ST-$graph (in blue). The red edges belong to a maximum matching M. M does not break Δ_1 and Δ_3.

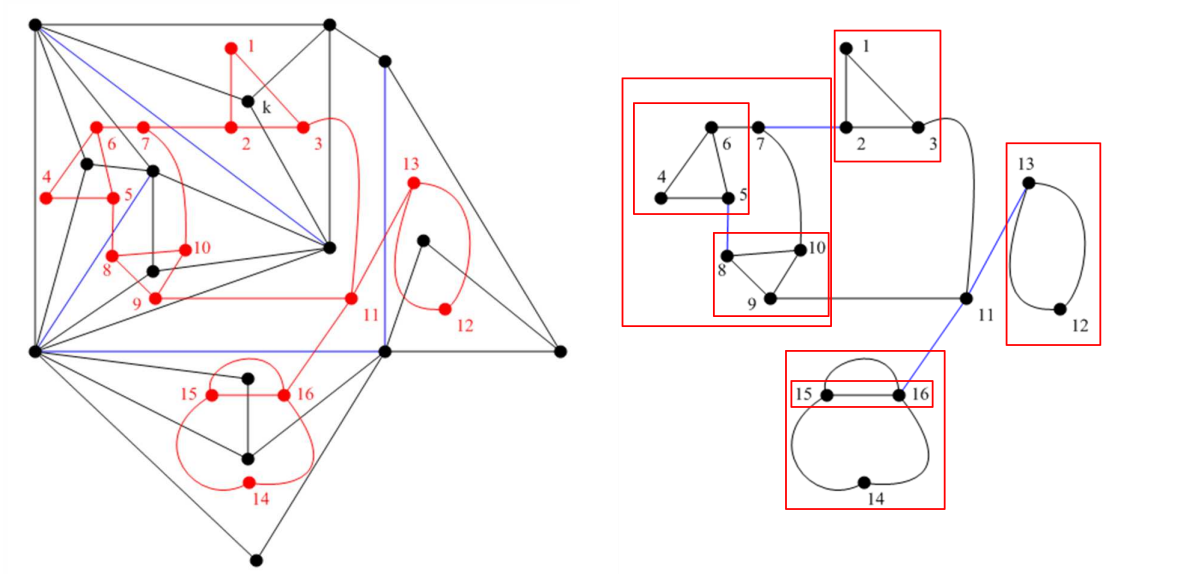
best known maximum matching algorithm for general plane graphs runs in $O(n^{1.5})$. Next, for each edge $(v_\Delta, v_\Gamma) \in M$, a crossover vertex is added on the edge which Δ and Γ share. If M is not a perfect matching, some vertices in $S(I)$ are free. Moreover, the neighbours of a free vertex are all matched, otherwise M would not be maximum. This means that the corresponding STs in I are still unbroken and that no two of them share an edge. A crossover vertex for each such ST is then added.

The algorithm described in [AAV00] disproves the Lai-Leinwand conjecture, as it solves OSTB in $O(n^3)$ time. This approach goes through three steps (Fig. 4.8):

Dualization. The *geometric dual* G^* of G is constructed.

Structuring. A HCG $\mathcal{G}$ is obtained creating, for each ST Δ, a cluster containing the vertices of G^* corresponding to the faces of G which are children of Δ.

Covering. OSTB is then mapped to a *minimum edge covering* on $\mathcal{G}$, asking for a minimum set C of edges in $\mathcal{G}$ such that each cluster is incident with at least one edge in C.

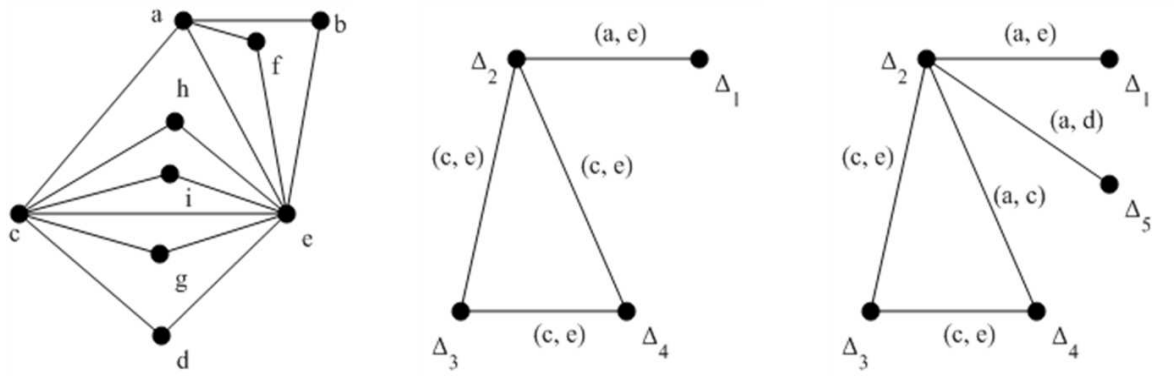


(a) (b)

Figure 4.8: (a) The plane graph G of Fig. 4.4 (in black) and its geometric dual G^* (in red). A solution to OSTB is obtained adding a crossover vertex on the blue edges. (b) The structured graph $\mathscr{G}$. Each cluster contains the vertices of G^* corresponding to faces enclosed in the corresponding ST. The blue edges (which are the dual of the blue edges in (a)) are a solution to a minimum edge covering on $\mathscr{G}$.

We also found a proof of NP-completeness, which, however, we do not agree with [RBM01]. This proof reduces OSTB to a *edge label covering* (ELC) problem. A new graph G' (called *adjacency graph*) is created from G, with a vertex v_Δ for each ST Δ in G and an edge $e = (v_{\Delta_1}, v_{\Delta_2})$ if and only if Δ_1 and Δ_2 share an edge (u, v) in G. (u, v) is used as the label of e (Fig. 4.9). A *minimal edge label covering* in G' is a minimal set L of labels such that each node is incident with at least one edge having a label in L. ELC is NP-complete; thus, in order to prove the NP-completeness of OSTB, ELC should reduce to OSTB. However, the authors only show that a solution to ELC in G' is also a solution to OSTB, but nothing say about the converse.

Heuristic algorithms for OSTB are presented in [RMB04]. One of these (MaxWeight) is described in Section 4.5.2. The others are schemes based on simulated annealing and valley descending. Some experimental results are presented, obtained on a test set of 25 randomly generated graphs. The results show that simulated annealing and valley descending outperform MaxWeight, while being more difficult to implement.



(a) (b) (c)

Figure 4.9: (a) Four STs: $\Delta_1 = (a, b, e)$, $\Delta_2 = (a, c, e)$, $\Delta_3 = (h, c, e)$, $\Delta_4 = (d, c, e)$ (b) The adjacency graph. (c) An instance of ELC which can not be reduced to an instance of OSTB (as Δ_2 should have four edges).

4.4 Relation to Classic Problems in Graph Theory

In this section we describe a reduction of OSTB to two well-known problems in graph theory: perfect matching and red/blue dominating set. In the first case, a solution is found in linear time, but it may not be optimal. In the second case, the solution is always optimal, but the computational cost of the algorithm may be unfeasible; this, however, only in cases that are very unlikely to occur in practice.

4.4.1 Matching in Cubic Graphs

The $ST-$graph (Section 4.3) of $0-$level containment graph is plane and subcubic. Unfortunately, there is no linear-time algorithm for maximum matching in subcubic graphs. Moreover, the $ST-$graph of $k-$level containment graphs may be neither planar nor subcubic. In Chapter 2 we have seen that cubic bridgeless graphs always admit a perfect matching (Petersen's Theorem) and that a linear-time algorithm for finding it exists [BBDL01]. Let G be a biconnected PTG. We observe that the extended geometric dual G^* is a plane bridgeless cubic graph (Lemma 2.7).

In this section, we prove that a perfect matching in G^* provides a solution to OSTB, which, however, is not guaranteed to be optimal. Stated in other words, let $M \subset E(G^*)$ be a perfect matching in G^*; the set C of the dual of the edges in M is a solution to OSTB, but C is not necessarily minimal. The proof of this fact is trivial and relies upon the following property.

Lemma 4.5. *Let G be a biconnected PTG. Any ST Δ encloses an odd number of $3-$faces.*

Proof. The subgraph G_Δ induced by the vertices of the frontier of Δ and the vertices enclosed in Δ is a maximal planar graph. By Euler's Formula, the number of inner-faces in G_Δ is $2n-3$, which is odd. □

By Lemma 4.5, the subgraph of G^* induced by the vertices corresponding to the faces enclosed in Δ has an odd number of vertices. This implies that the dual of at least one edge of the frontier of Δ belongs to M. Therefore, each ST of G is incident with at least one edge in C (Fig. 4.10). The main drawback of this approach is that a cubic graph generally

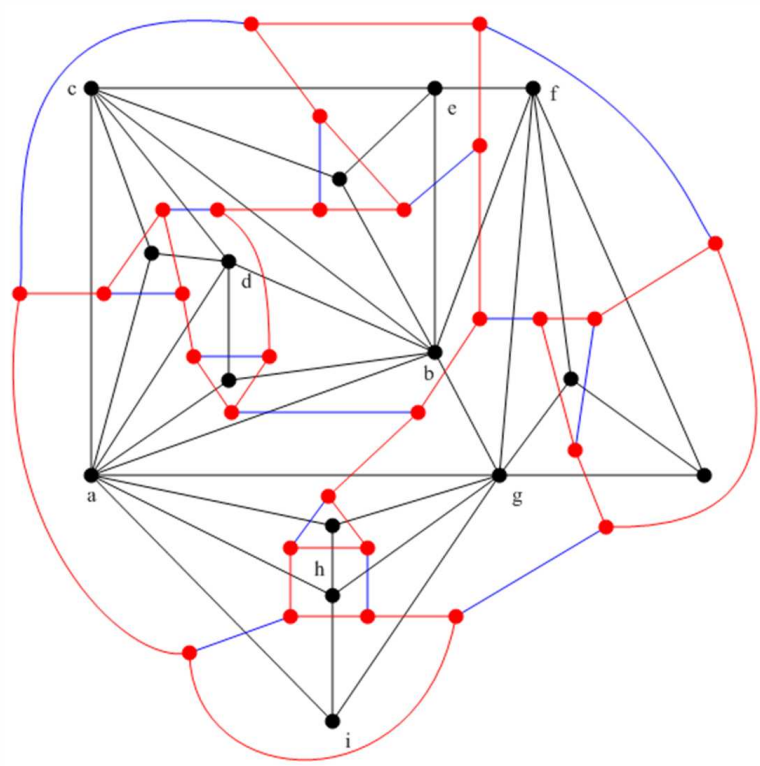


Figure 4.10: The graph of Figure 4.4 (with additional edges to make it a PTG) with its extended geometric dual G^*(in red). The blue edges belong to a perfect matching in G^*. Correspondingly, to break STs, a crossover vertex is added on six edges ((a,b), (c,d), (b,e), (f,g), (g,h), (a,i)), whereas an optimal solution only requires four.

has many perfect matchings and only few provide a solution to OSTB (Fig. 4.11). To narrow down the solution space to only accept good solutions, we assign each edge e of G (and, dually, of G^*) a *weight*, which represents the number of STs incident with e. It is not difficult to modify the algorithm in [BBDL01] in such a way that at each step the edge with maximum weight, among those eligible for matching, is chosen. Still, the obtained

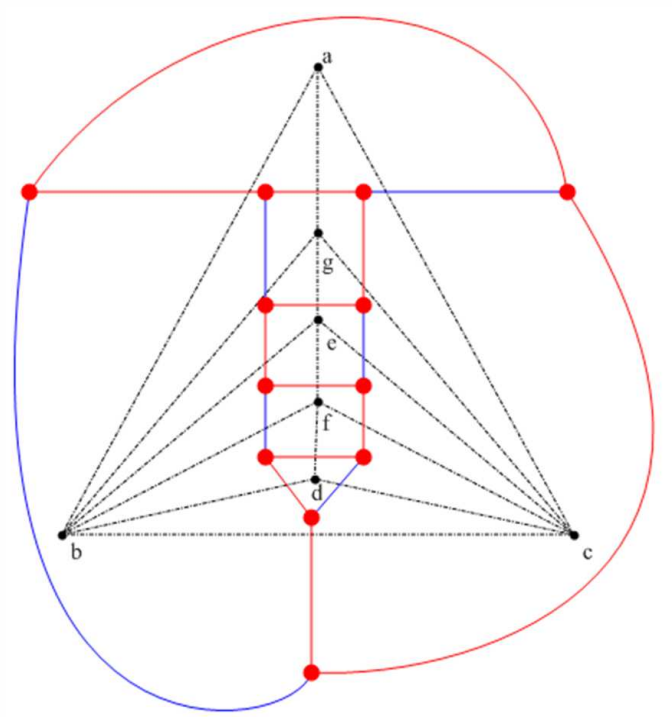


Figure 4.11: Four STs incident with only one edge ((b, c)). The solution corresponding to the matched edges (in blue) contains 4 edges, whereas only one would have been enough. Using weights, only perfect matchings corresponding to the optimal solution are computed.

solution may not be optimal, but is better than the ones obtained on an unweighted graph. We remark that this approach does not necessarily compute a maximum-weighted perfect matching; even if this was the case, a maximally-weighted perfect matching may not be an optimal solution neither.

4.4.2 Minimum Red-Blue Dominating Set

Let H be a bipartite graph with vertex classes V_{red} and V_{blue}. A *red/blue dominating set* is a subset $D \subseteq V_{blue}$ such that every vertex in V_{red} is adjacent to at least one vertex of D. A *minimum red/blue dominating set* (MRBDS) is a red/blue dominating set having minimum cardinality. Let $B(I)$ be the bubble graph of an island I; solving OSTB in $B(I)$ reduces to finding a MRBDS in the *Triangle-Edge Incidence Graph* I'. I' is a bipartite graph with vertex classes V_{ST} and V_E. V_{ST} has a node v_Δ for each bubble Δ and V_E has a node for each vertex of $B(I)$; $(v_\Delta, v_e) \in E(I')$ if and only if e is on the frontier of Δ (Fig. 4.12).

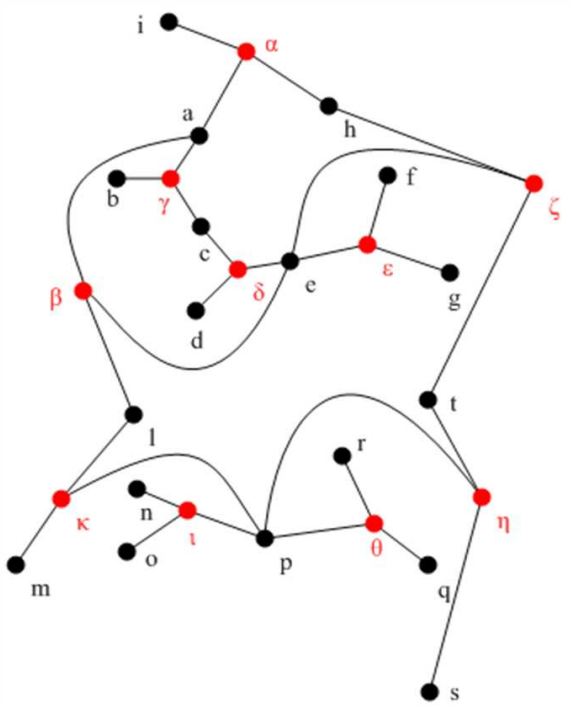


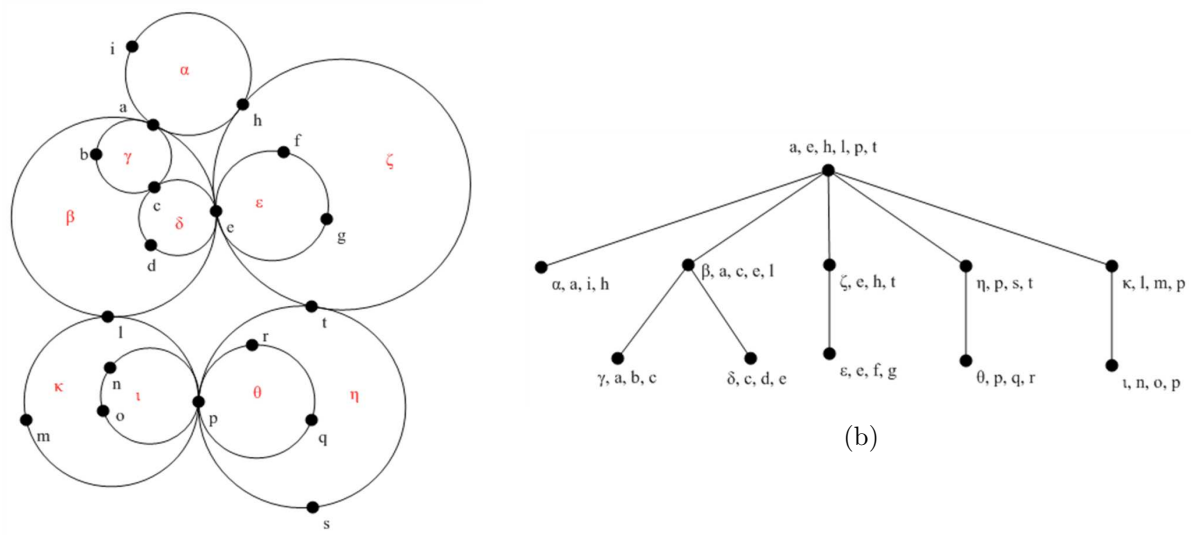
Figure 4.12: The TEIGs of the islands in Fig. 4.13 (a).

Finding a minimum red/blue dominating set in general bipartite graphs is known to be $W[2]$-hard, that is fixed parameter intractable [Fer06]. Downey and Fellows gave an $O(12^l n)$ algorithm solving the problem in bipartite planar graphs, where l is the cardinality of the smallest red/blue dominating set [DF96]. For bipartite graphs with bounded treewidth, the problem is fixed parameter tractable [AN02].

Theorem 4.1. *Given a bipartite graph H and a tree decomposition of H of width at most k and N vertices,* Minimum Red/Blue Dominating Set *can be solved in time $O(3^k \cdot N)$ on H.*

The $ST-$tree T of $B(I)$ is not a good candidate as a tree decomposition (Fig. 4.13). Indeed, to obtain a tree decomposition of I', each node v_Δ of T, corresponding to bubble Δ, is associated with a bag $\mathcal{B}(v_\Delta)$ containing v_Δ, v_x, v_y and v_z, x, y and z being the vertices incident with Δ. If Δ_1 and Δ_2 are children of Δ, the items belonging to $\mathcal{B}(v_{\Delta_1}) \cap \mathcal{B}(v_{\Delta_2})$ must also belong to $\mathcal{B}(v_\Delta)$. As a result, the cardinality of $\mathcal{B}(v_\Delta)$ may be large, even if $B(I)$ contains few bubbles.

Finding a good tree decomposition of I' is challenging even if I is a $0-$level containment island. And yet in this case, I' is a plane subcubic bipartite graph, which means that



(a) (b)

Figure 4.13: (a) An island I. (b) Tree decomposition based on the $ST-$tree of I.

belongs to a very special class of graphs. Surprisingly enough, such graphs seem not to have a bounded treewidth. However, in [ABFN00] we found an interesting result that relates treewidth to $r-$outerplanarity.

Theorem 4.2. *Let G be a plane $r-$outerplanar graph. A tree decomposition of G with $O(n)$ nodes and width at most $3r-1$ can be found in $O(rn)$ time.*

In the worst case, I' is an $r-$outerplanar graph with $r \in O(n)$. In fact, the smallest $0-$level containment island I such that I' is $2-$outerplanar contains 8 STs (Fig. 4.14); the smallest I such that I' is $3-$outerplanar has $8+8=16$ STs. In general, the smallest $0-$level containment island I such that I' is $r-$outerplanar has $8(r-1)$ STs. In Section 4.5 we will show that in randomly generated graphs r is usually small. In other words, in all practical cases, I' has bounded treewidth and the algorithm described in this section works in linear time.

4.5 Heuristic Algorithms

Since it is difficult to find a simple linear-time algorithm for OSTB, we devised a set of heuristic approaches, based on some interesting properties of the bubbles. An heuristic, by definition, is not an exact algorithm and thus it does not guarantee an optimal solution in any possible case. However, in this specific context, the instances of OSTB which are likely

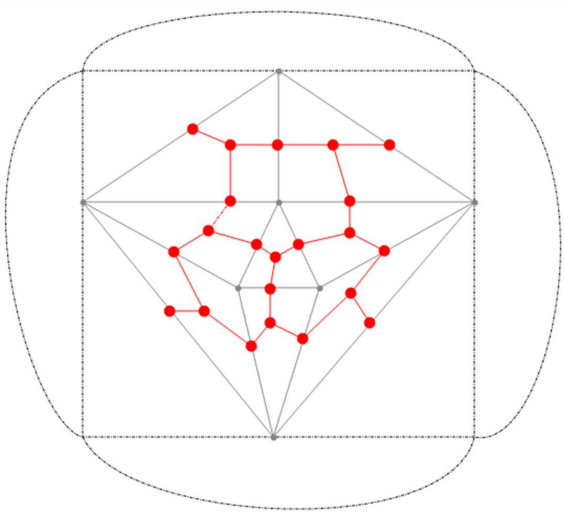


Figure 4.14: The 8 grey triangles form the smallest $0-$level containment island I such that I' (in red) is $2-$outerplanar. Adding the 8 dashed triangles, we turn I' into a $3-$outerplanar graph, with I having 16 triangles. Iterating the procedure, if I contains $8(r-1)$ triangles, I' is $r-$outerplanar.

to occur in practice are a small subset of the set of all instances. We observe, in fact, that STs do not occur frequently in plane graphs. It was estimated that in randomly generated PTGs the expected number of STs is approximately 16% of the number of nodes [YS91]. The PTGs generated with the OcORD random graph generator confirm this figure. Since PTGs are the most dense planar graphs, we can conclude that the expected number of STs in a planar graph is at most 16% of the number of nodes. The complexity of OSTB, however, does not strictly depends on the overall number of STs, but on the size of the largest *island*. Again, the PTGs generated with OcORD show that islands are small on average.

The aim of the heuristic algorithms we describe in this section is to find the optimal solution in almost all practical cases. Let $B(I)$ be the bubble graph of an island I. Each vertex v of $B(I)$ is assigned a *weight* $W(v) = d(v)/2$ (where $d(v)$ is the degree of v), which represents the number of bubbles v is incident with. Since $d(v)$ is even and positive (Lemma 4.3), $W(v) \in \mathbb{N}$. Alternatively, $W(v)$ can be seen as the number of STs incident with the edge of I corresponding to v. A bubble β is a $k-$bubble, $0 \leq k \leq 3$, if k of its vertices have weight > 1. For instance, in Fig. 4.13 ϵ, θ and ι are $1-$bubbles, ζ and β are $3-$bubbles and the remaining are $2-$bubbles. In order to simplify the notation, we remark that $1-$weighted vertices are simplicial[1], while the others are not. Henceforth, we will refer to $1-$weighted vertices as simplicial and to $k-$weighted vertices as *non-simplicial*.

[1]A vertex is *simplicial* if its neighbours induce a clique

4.5.1 Pruning

Based on weights, we prove two properties, which induce a powerful method (called *pruning*) for reducing the number of bubbles in $B(I)$ (Fig. 4.15).

Lemma 4.6. *Any vertex of a* $0-$*bubble belongs to an optimal solution of OSTB.*

Proof. Trivially true, as a $0-$bubble corresponds to an isolated ST. □

Lemma 4.7. *Let* $\beta = (x, y, z)$ *be a* $1-$*bubble and* x *its non-simplicial vertex. If* y *(or* z*) belongs to an optimal solution* O*, then the set obtained from* O *by removing* y *(or* z*) and adding* x *is also an optimal solution.*

Proof. Let O' be the set obtained from O removing y ($|O'| = |O| - 1$). Since β is the only bubble incident with y, only β is unbroken with solution O'. But then adding x to O' breaks β and $|O'| = |O|$. □

Breaking a bubble $\Delta = (u, v, w)$ *through* vertex v ($Break(\Delta, v)$ in short) means that edge (u, w) is deleted from $B(I)$. In fact, by Lemma 4.3, Δ is the only bubble incident with both w and u. v is removed only after breaking all bubbles incident with it.

Algorithm 4.1 Pruning

```
1: procedure PRUNING(B(I): graph, Bubbles: list of bubbles)
2:     Δ ← FIRST(Bubbles)
3:     if Δ = (u, v, w) is a 0−bubble then
4:         BREAK(Δ, v) ; BACKINSERT(v, O); DELETE(Δ, Bubbles)
5:     end if
6:     if Δ = (u, v, w) is a 1−bubble, v being the non-simplicial node then
7:         BACKINSERT(v, O)
8:         for all Δ incident with v do
9:             BREAK(Δ, v) ; DELETE(Δ, Bubbles);
10:        end for
11:    end if
12:    Delete v and any isolated node of B(I) and updates weights.
13:    for all connected component B(I') do
14:        PRUNING(B(I'), Bubbles)
15:    end for
16: end procedure
```

Pruning works as follows. If there is a $0-$bubble Δ, one of its vertices v is added to O and Δ is broken through v. If there are no $0-$bubbles, the algorithm consider $1-$bubbles. Let

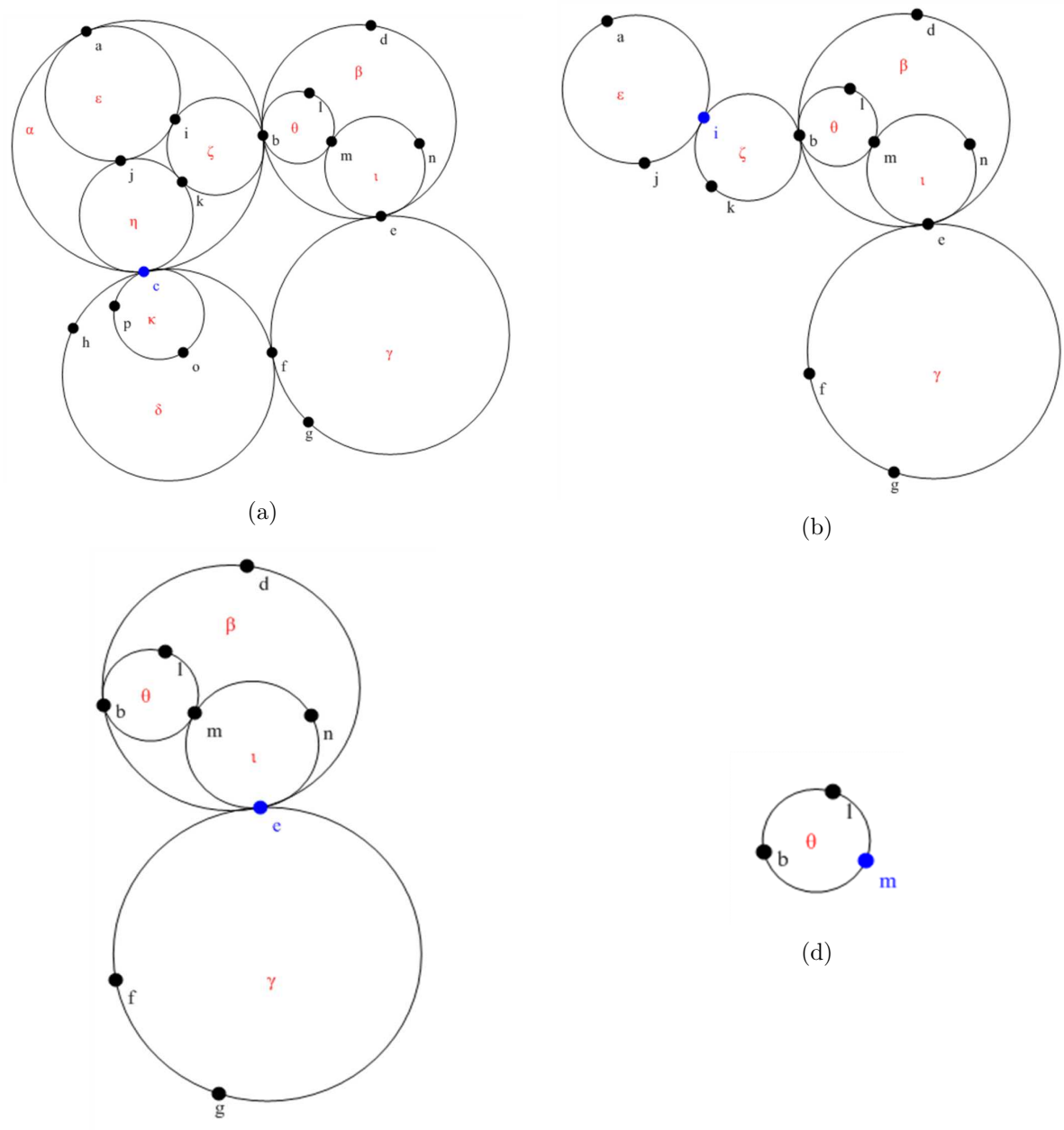


Figure 4.15: Example of an island broken by pruning. At each step, the blue vertex is added to the solution.

$\Delta = (x, y, z)$ be a $1-$bubble and x be its non-simplicial vertex. x is added to O and all bubbles incident with x are broken through x. Then, x is removed from B. As a result, the degree (and thus the weight) of the neighbours of x must be updated; any $0-$weighted node is removed. After breaking Δ, $B(I)$ may have more than one connected component, each corresponding to a different island. Thus, the procedure is iterated on each island, until all bubbles are broken or only $2-$ and $3-$bubbles remain. In the first case, by Lemma 4.6 and 4.7, O is an optimal solution to OSTB.

4.5.2 MaxWeight

MaxWeight is a greedy algorithm. At each step, in fact, adds a maximum-weighted vertex v to O, breaks through v all bubbles incident with v and removes it. Then, it updates the weights of the vertices adjacent to v and iterates the procedure until all bubbles are broken (Fig. 4.17).

This heuristic has two major issues. Firstly, at each step there may be more than one maximum-weighted vertex and some may not belong to an optimal solution (Fig. 4.16 (a)). Unfortunately, there is not a general criterion to state whether a vertex belongs to an optimal solution or not. Therefore, MaxWeight chooses one randomly. Secondly, even a unique maximum-weighted vertex may not belong to an optimal solution (Fig. 4.16 (b)).

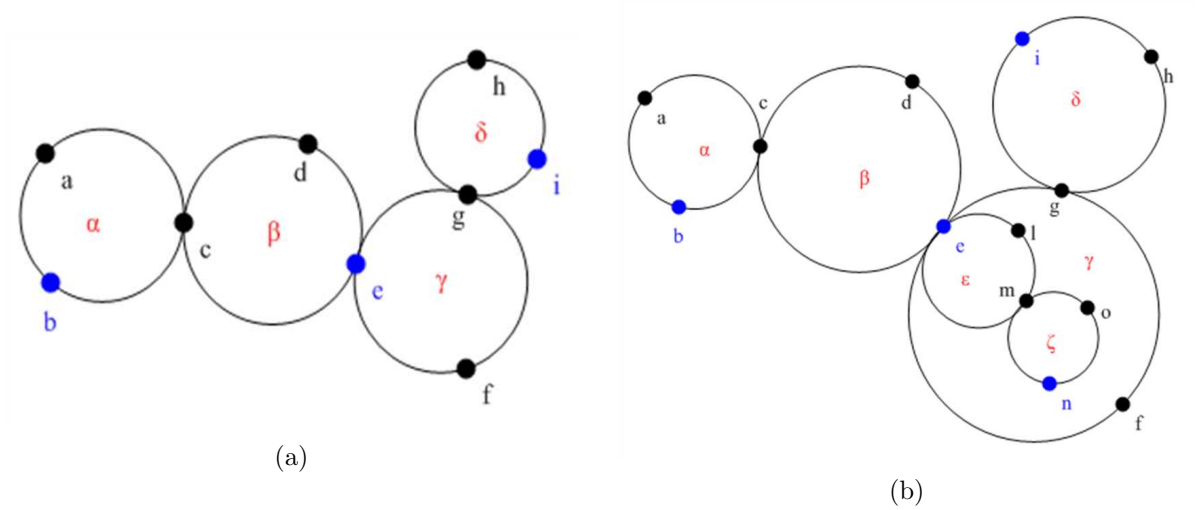


Figure 4.16: (a) The blue vertices belong to the solution found by MaxWeight, starting with vertex e. The optimal solution is $\{c, g\}$. (b) The solution found by MaxWeight starting with vertex c (which is the only one with weight 3). However, an optimal solution only needs three vertices.

In order to improve the efficiency of MaxWeight, we couple it with pruning, obtaining a new heuristic called Prune MaxWeight. First, $B(I)$ is reduced using pruning; if some

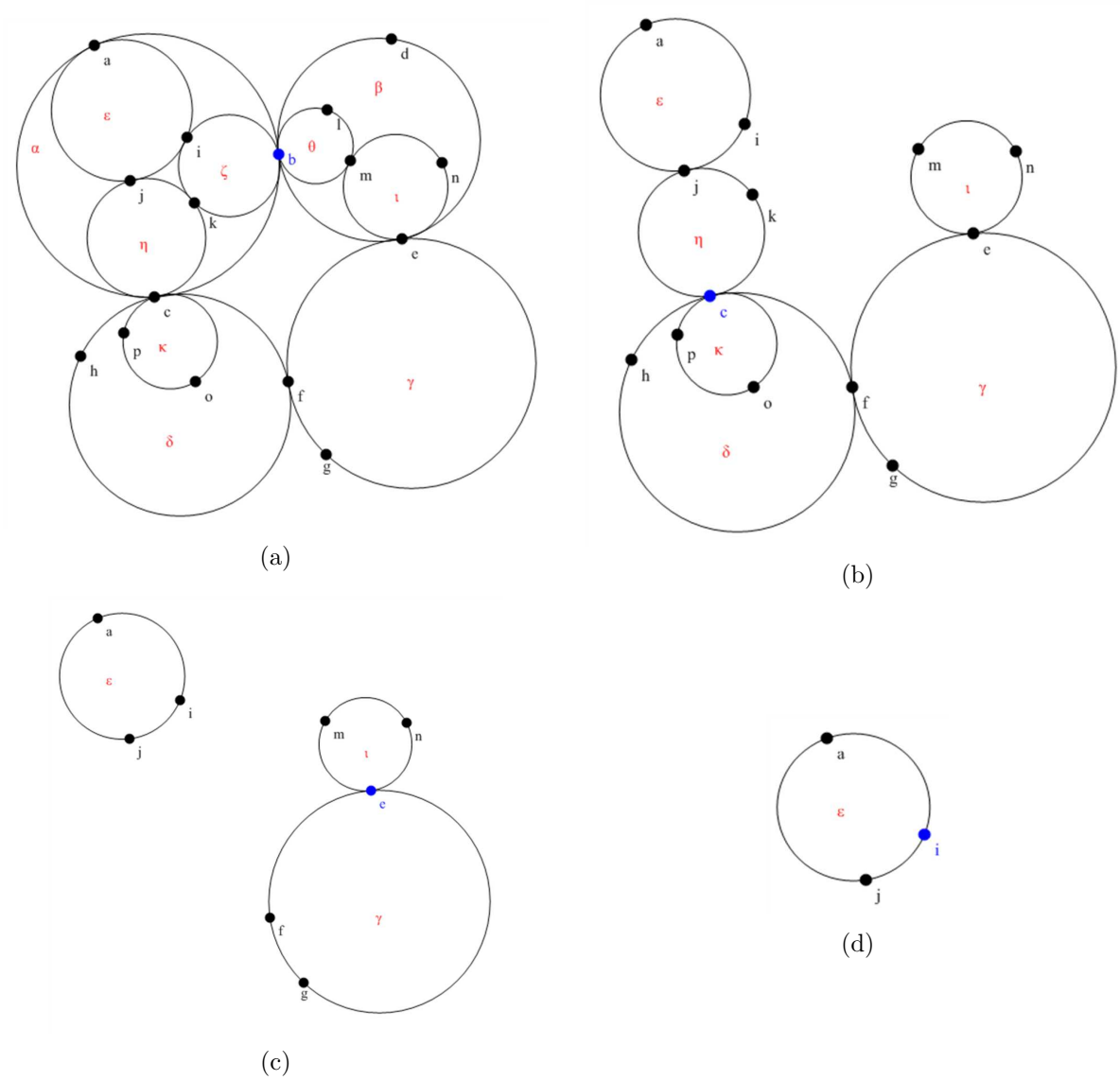


Figure 4.17: Example of an island broken by MaxWeight.

bubbles are still unbroken, a step of MaxWeight is performed. Next, pruning is used again to prune any new $0-$ or $1-$bubble.

Finally, a further improvement is obtained running twice MaxWeight (2-MaxWeight) and Prune MaxWeight (*2-Prune MaxWeight*). Since at each step vertices are chosen randomly among those with maximum weight, the obtained results are likely to differ. In this case, the better solution is kept.

4.5.3 MiniMax and SubCubic

As seen in the previous section, MaxWeight may fail also in simple occurrences of OSTB. Clearly, this is disappointing, as the aim of an heuristic is to provide an efficient, solution in at least all simple cases (which are those occurring the most). In particular, the example in Fig. 4.16 (a) is extremely simple, as it is a $0-$level containment island. MiniMax bases on the observation that in a $0-$level containment island, the weight of a $k-$bubble[2] is less than the weight of a $(k+1)-$bubble, $0 \leq k \leq 2$. In fact, $0-$bubbles weigh 3, $1-$bubbles weigh 4, $2-$bubbles weigh 5 and $2-$bubbles weigh 6. Lemma 4.6 and 4.7 say that it is better to create an optimal solution breaking first $0-$ and $1-$bubbles. In a $0-$level containment island, this corresponds to breaking first the minimum-weighted bubbles through their maximum-weighted vertex (Fig. 4.18). When two bubbles have the same weight, MiniMax chooses one randomly. Surprisingly enough, our test results (Section 4.6) prove that MiniMax scales well also to $k-$level containment islands, $k > 0$. As done for MaxWeight, we also implemented Prune MiniMax, 2-MiniMax and *2-Prune MiniMax*. SubCubic is a further refinement of MiniMax, as it explicitly breaks all $k-$bubbles before any $(k+1)-$bubble. Thus, it incorporates pruning.

4.6 Experimental Results

In order to evaluate the efficiency of the heuristics described in the previous section, we tested them on randomly generated graphs. The crux is how to generate plane graphs, containing a significant number of STs. In other words, we would like to generate the "worse" plane graphs, which are likely to contain more STs than other plane graphs. In a general random graph $G(n,p)$, the expected number of triangles is p^3n^3, p being the probability that an edge occurs between any two vertices. The greater is p, the more dense is G. Thus, the number of triangles increases with the density of the graph. For this reason, we decided to test our heuristic algorithms on PTGs, which are the most dense among plane graphs. Our algorithm for generating a PTG G works as follows (Fig. 4.19).

[2]The weight of a bubble is the sum of the weights of its vertices.

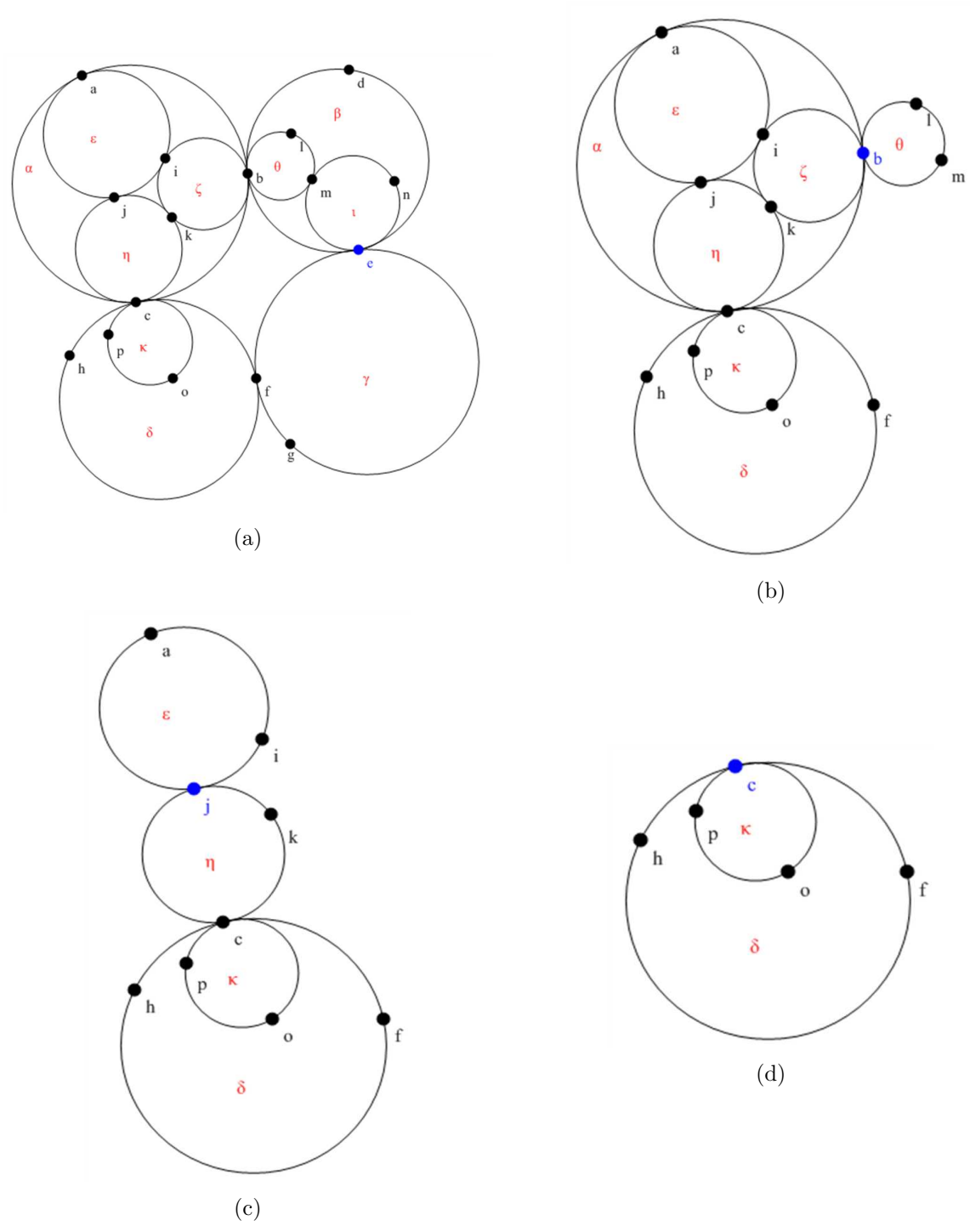


Figure 4.18: Example of an island broken by MiniMax.

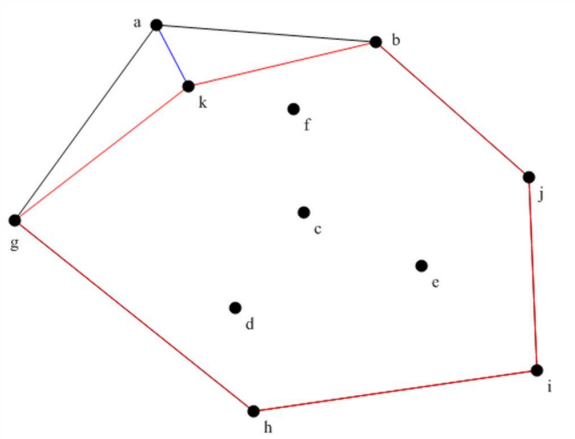


Figure 4.19: One iteration of the OcORD random graph generator. First a convex hull of the whole point set is created. Then vertex a is removed and a new convex hull (in red) is computed. Finally, all vertices belonging to the second convex hull, but not to the first, (in this case k) are linked to the removed vertex a.

First, it generates n points in the plane. These points form the vertex set $V(G)$ of G. Next, it creates the convex hull[3] $CH(V(G))$ of $V(G)$. Among the vertices of the convex hull, one (say v) is chosen and the convex hull $CH(V(G)-\{v\})$ is created. For each vertex w, such that $w \in CH(V-\{v\})$ and $w \notin CH(V)$, an edge is added between w and v. The algorithm iterates this procedure until no further convex hull can be created.

One could object that the graphs generated by this algorithm are not properly random. Indeed, we do not need a random graph, but a graph in which STs are randomly distributed. We experimentally verified that graphs with the same order significantly differ in the number of STs, islands and STs per island.

The test set is composed of 10000 graphs, with order ranging from 100 to 1000 nodes. For each heuristic, we evaluated four parameters: success (S), bad failure (BF), bad failure within an island (BF island) and optimality deviation (OD). An heuristic scores a *success* when it finds the optimal solution. A *bad failure* occurs when the size of the computed solution exceeds $|O|+2$, O being the optimal solution. A bad failure may be the sum of non-bad failures on different islands; worse, a bad failure can also occur within an island. Finally, the *optimality deviation* (OD) is the difference between the size of a bad failure and the size of the optimal solution. Each heuristic is run 100 times on every graph of our test set; each parameter is assigned the average of the scores obtained after each run.

[3]The convex hull of a point set X is the smallest convex set containing X.

Table 4.1: Test Set

Average Graph Order	501
Average ST number	89
Max ST number	223
Ratio ST/Order	18%
Average Island Size	1.55
Max Island Size	21
Average Pruning Effect	96.80%
Solutions found by Pruning	4583/10000

4.6.1 Discussion

Table 4.1 sums up the main characteristics of the graphs in the test set. As stated in [YS91], the number of STs is on average 18% of the number of nodes. The largest island has 21 STs, but on average islands are small, as they hardly contain 2 STs. In Section 4.4.2 we have seen that the smallest island I, for which its TEIG I' is $r-$outerplanar, contains $8(r-1)$ STs. Thus, the TEIGs of the islands in the graph of the test set are at most $4-$outerplanar graphs.

The performance of *Pruning* is particularly impressive. Alone, it manages to break all STs in almost half (4583/10000) of the graphs. In other words, in **46%** of the graphs, the optimal solution is found only using *Pruning* and no heuristic. In the remaining 54%, the effect of pruning is still remarkable, as it breaks almost **97%** of the STs. Thus, either *Pruning* finds an optimal solution or it significantly reduces the number of bubbles. In

Table 4.2: Heuristic Test Results

Heuristic	**S (%)**	**BF (%)**	**BF Island(%)**	**OD (%)**
MaxWeight	77.57	4.97	17.24	5
Prune MaxWeight	99.97	0	0	1
MiniMax	97.58	0.02	0.08	2
Prune MiniMax	99.73	0	0	1
2-MaxWeight	87.70	1.75	14.10	4
Prune 2-MaxWeight	99.99	0	0	1
2-MiniMax	98.66	0.004	0.06	2
Prune 2-MiniMax	99.73	0	0	1
SubCubic	99.95	0	0	1

the specific case of our test set, the average number of STs is 89; thus the islands which the heuristics using pruning have to deal with contain no more than 3/4 STs on average. To make more evident the impact of *Pruning*, we selected from the test set the graphs which *Pruning* reduced without eliminating all STs and we run all heuristic algorithms on

them. Table 4.3 shows the results: all heuristics have almost a 100% success rate. This explain why the heuristics, when coupled with *Pruning*, outperform the heuristics not using *Pruning* (Table 4.2). Moreover, the optimality lag of the heuristics using *Pruning* never exceed 1. The success rate of SubCubic is particularly impressive; it depends on that 98% of the bubbles SubCubic deals with are 0− or 1−bubbles.

Table 4.3: Heuristic Test Results on Pruned Graphs.

Heuristic	**S (%)**
MaxWeight	99.69
Prune MaxWeight	99.96
MiniMax	99.81
Prune MiniMax	99.82
2-MaxWeight	99.89
Prune 2-MaxWeight	99.99
2-MiniMax	99.87
Prune 2-MiniMax	99.82
SubCubic	99.92

As for the heuristics which do not use *Pruning*, we remark the gap between MaxWeight and MiniMax. This is something we predicted, when we noticed that MaxWeight may fail also on small 0−level containment islands. On the other hand, MiniMax is equivalent to *Pruning* on 0−level containment islands and, as it comes out from the results, it mimics well the behaviour of *Pruning* in $k-$level containment islands. To strike a blow for MaxWeight, we notice that the best result (99.9% of successes) is obtained with Prune 2-MaxWeight. It is indeed interesting how MaxWeight benefits from *Pruning*. Prune MaxWeight, in fact, performs slightly better than Prune MiniMax, in spite of the gap between the two heuristics without pruning. This may be a further evidence of the similarity of MiniMax with *Pruning*; with a quip, we would say that two similar procedures get in each other's way! A possible (serious) explanation of the fact is that after the removal of 0−bubbles and 1−bubbles, the remaining bubbles have almost the same weight, thus making hard the choice of MiniMax. Finally, the behaviour of MiniMax proved to be more stable and predictable than MaxWeight; the performances of 2-MiniMax and MiniMax are comparable, whereas between MaxWeight and 2-MaxWeight there is a considerable gap.

Chapter 5

Graph Visualization and Rectangular Dualization

In this chapter we describe the application of rectangular dualization to graph visualization. We give particular emphasis to the visualization of clustered graphs, introducing *c-rectangular dualization*. We also describe the role of a rectangular dual graph in visualizing 3D Electronic Institutions.

5.1 Drawing Planar Graphs

In this section, we describe three algorithms for drawing plane graphs from their rectangular dual. In our knowledge, rectangular dualization has never been considered as a graph drawing method. This is partly due to the lack of efficient (linear time) algorithms as well as to the admissibility conditions, which restrict the scope of rectangular dualization to a small class of graphs. The algorithms presented in this thesis should have contributed to overcome such obstacle.

Given a plane connected graph G, the three algorithms first create a rectangular dual $\mathscr{R}(G) = (\Gamma, f)$ and then they create a drawing from it with a specific drawing style. The drawing styles used are orthogonal box drawing, orthogonal drawing and bus-mode drawing. The latter generalizes orthogonal drawing to graphs with maximum degree greater than 4. With only minor changes, the same algorithms create, with the aforementioned drawing styles, drawings for clustered graphs (Section 5.3). Fig. 5.1 shows the sample graph which was used as input to the three drawing algorithms. It contains a separating triangle $\Delta = (a, g, d)$, broken by crossover vertex c_1, and has some non-triangular faces. We intentionally chose a non-rectangular graph, to show that the obtained graph drawings

are not affected by any edge or crossover vertex added by the rectangular dualization algorithm.

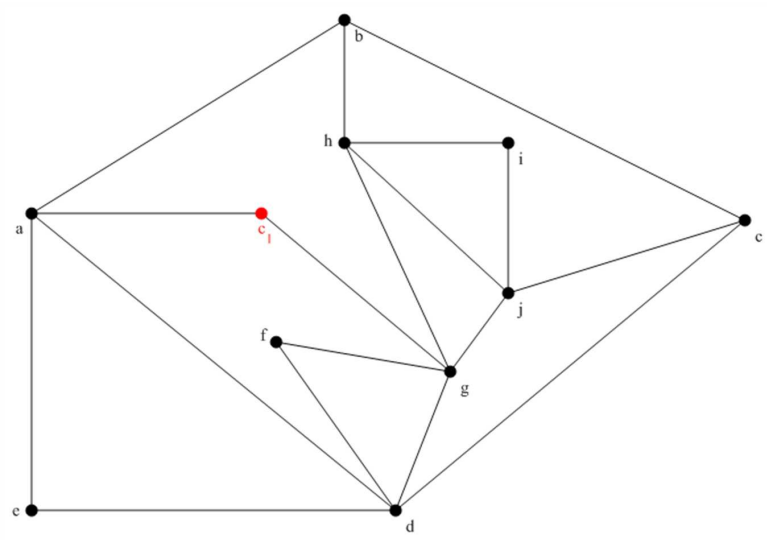


Figure 5.1: Sample graph.

5.1.1 Orthogonal Box Drawing

The orthogonal box drawing is the easiest type of drawing that can be created from a rectangular dual. In fact, the vertices are already represented as "boxes" in $\mathscr{R}(G)$. The algorithm draws a box (typically a rectangle) inside each rectangle of $\mathscr{R}(G)$ (Fig. 5.2). The size of each box is bounded by the size of the enclosing rectangle and must be such that two adjacent boxes can be linked by a straight line. To obtain this, we introduce the notion of *window*.

Definition 5.1. *Let x and y be two adjacent rectangles in $\mathscr{R}(G)$. We call* window *between x and y the portion of edge they have in common. If x and y are adjacent through a gate g, the window between x and y is the segment that x and g and the segment that y and g share.*

The size of the box inside a rectangle x is set up in such a way that for each window w, the line perpendicular to w and passing through a point of the box intersects w. If $\mathscr{R}(G)$ has no gate, the drawing has no bends. It is not difficult to see that in the worst case an edge has at most two bends.

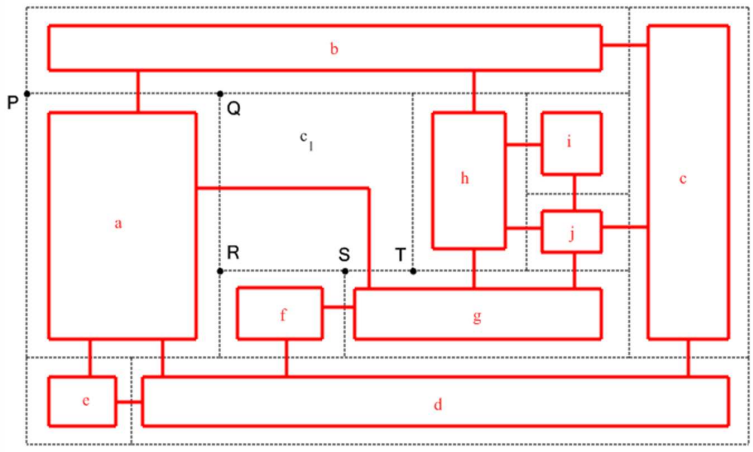


Figure 5.2: A rectangular box drawing. The dashed edges belong to the rectangular dual. Segment PQ is the window between a and b. QR and ST is the window between a and g, which are linked through a gate.

From a strictly aesthetic point of view, the fact that each vertex is represented as a box of different size might not be desirable. However, the size of a rectangle in $\mathscr{R}(G)$ is generally proportional to the degree of the corresponding vertex in G. Consequently, high-degree vertices are represented as larger boxes than the others. This improves the angular resolution between the edges incident with high-degree vertices. Moreover, we remark that such a drawing may also be obtained from a rectangular cartogram; in this case, the different sizes of the boxes, rather than being unpleasant to see, enhance the semantic of the graph.

5.1.2 Orthogonal Drawing

As anticipated in Chapter 1, only graphs with maximum degree 4 admit an orthogonal drawing. This limitation makes easier the creation of such a drawing with rectangular dualization. Each rectangle, in fact, has exactly four sides. We assume that nodes are represented as small circles and edges as polygonal chains of segments, each being parallel to either the $x-$axis or to the $y-$axis (Fig. 5.3). The first issue is deciding the correct position for each vertex v inside the corresponding rectangle $f(v)$. Placing v at the centre of $f(v)$ is not much of a choice, as it would create too many bends. An edge between u and v, in fact, would have no bends only if the centres of $f(u)$ and $f(v)$ had the same $x-$ or $y-$coordinate. The second issue concerns the edges. Adding one edge at a time, in fact, also may produce too many bends (Fig. 5.4).

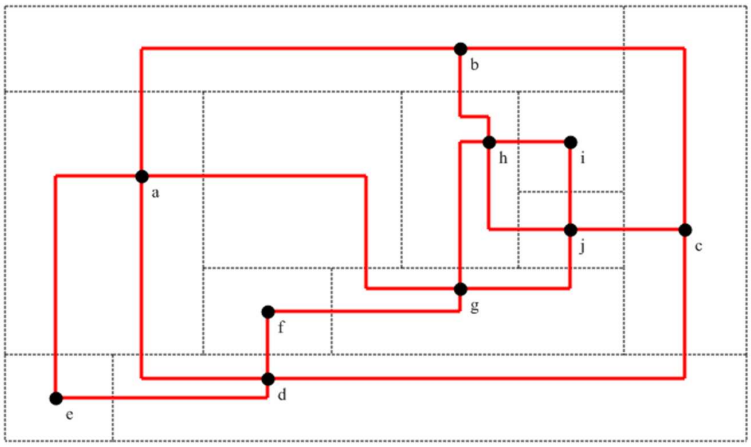


Figure 5.3: Orthogonal Drawing.

The orthogonalization algorithm is based on the notion of *orthogonal transform* (Fig. 5.5) of a rectangle [ACDQ06].

Definition 5.2. *The* orthogonal transform *of a rectangle R of $\mathscr{R}$ is a pair $T_R = (p, \mathscr{H})$ such that:*

1. *p is a point in the plane enclosed in R.*
2. *$\mathscr{H}$ is a set of as many polygonal chains as the adjacencies of R. Let S be a rectangle adjacent to R and s be the polygonal chain corresponding to this adjacency. One endpoint of s is p and the other is a point laying on the window between R and S.*

Creating an orthogonal drawing for G reduces to computing an orthogonal transform for each rectangle in $\mathscr{R}(G)$. Since the degree of a vertex can not exceed four, there is a finite (and reasonably small) set of cases (Fig. 5.6 to 5.9), which an algorithm can check to decide the correct position of a vertex, as well as a nice drawing of the incident edges. As before, any gate is considered as a mere interconnection area. We now briefly describe the implementation of this algorithm (Algorithm 5.1). An array OT stores the orthogonal transforms of each rectangle. Each entry of OT is associated with the following information:

- The coordinate of the vertex inside the rectangle.
- The coordinates of the points laying on the windows of the rectangle (at most 4).

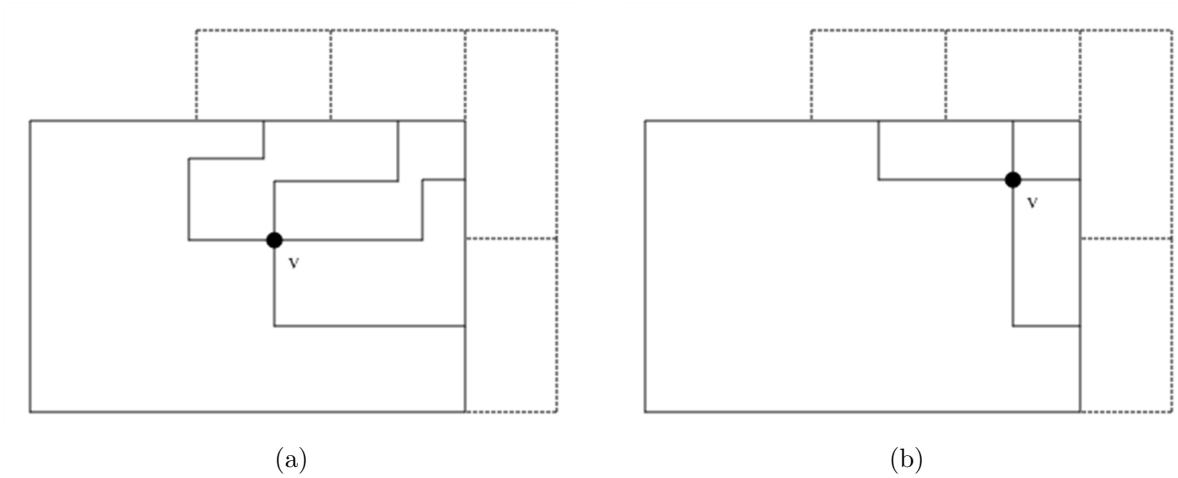


Figure 5.4: (a) Drawing obtained placing vertex v at the centre of rectangle $f(v)$ and adding one edge at a time (b) Drawing obtained placing v so as to minimize the number of bends. The position of v depends on the position of the adjacent rectangles.

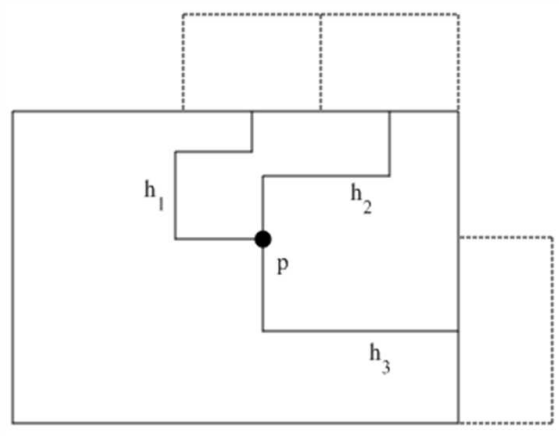


Figure 5.5: An orthogonal transform of a rectangle.

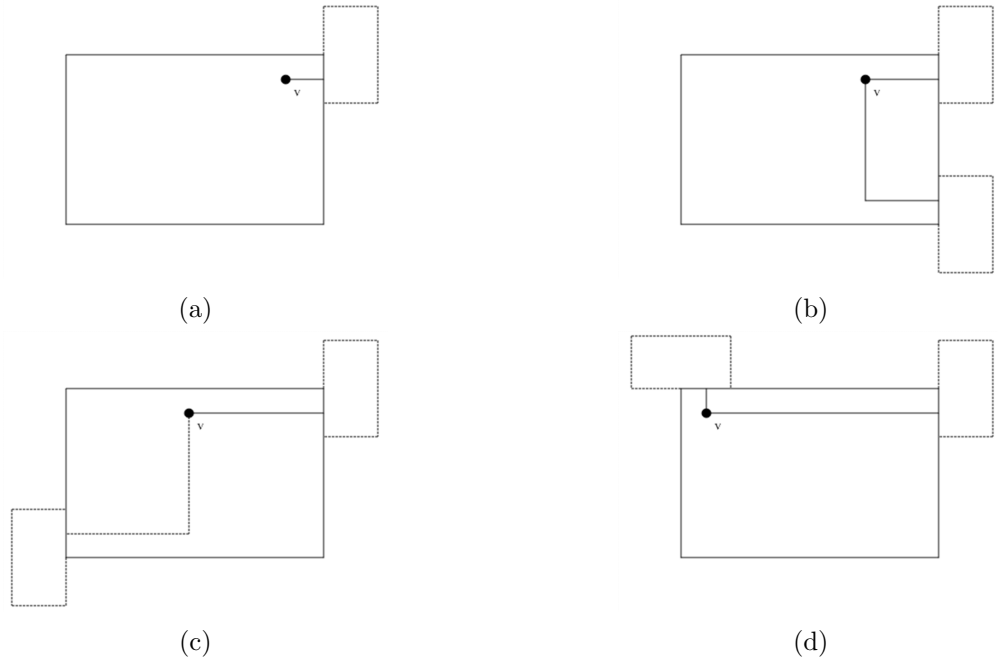


Figure 5.6: (a) v has degree 1. (b)-(d) v has degree 2.

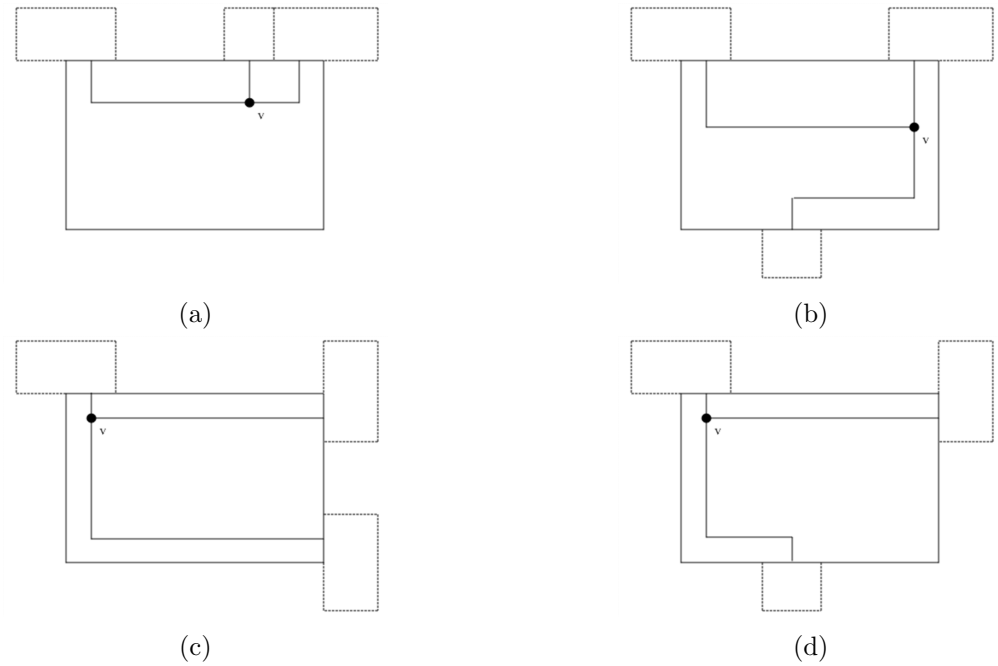
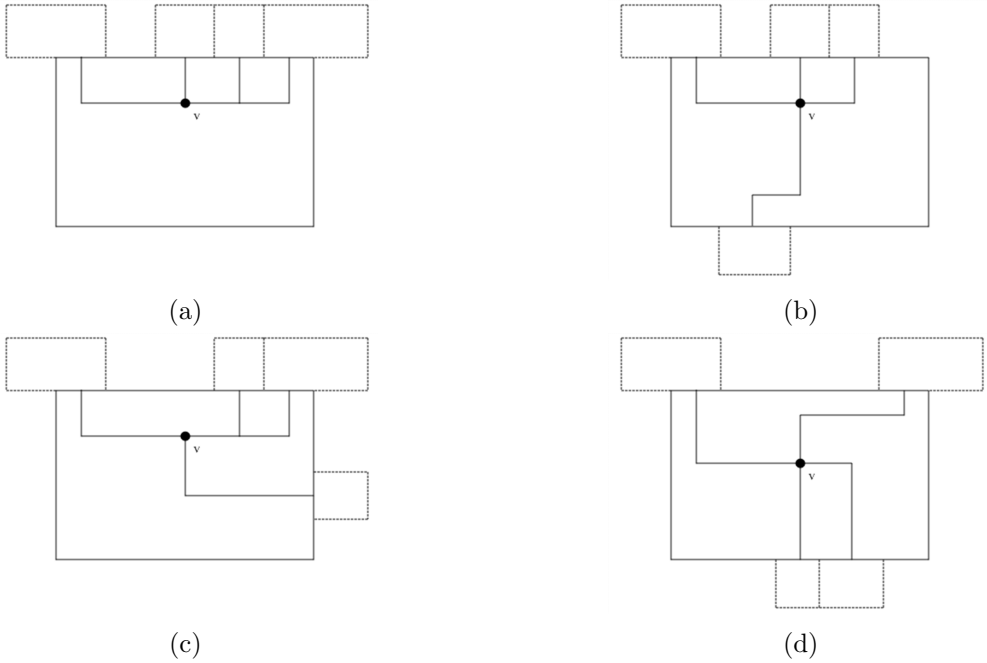


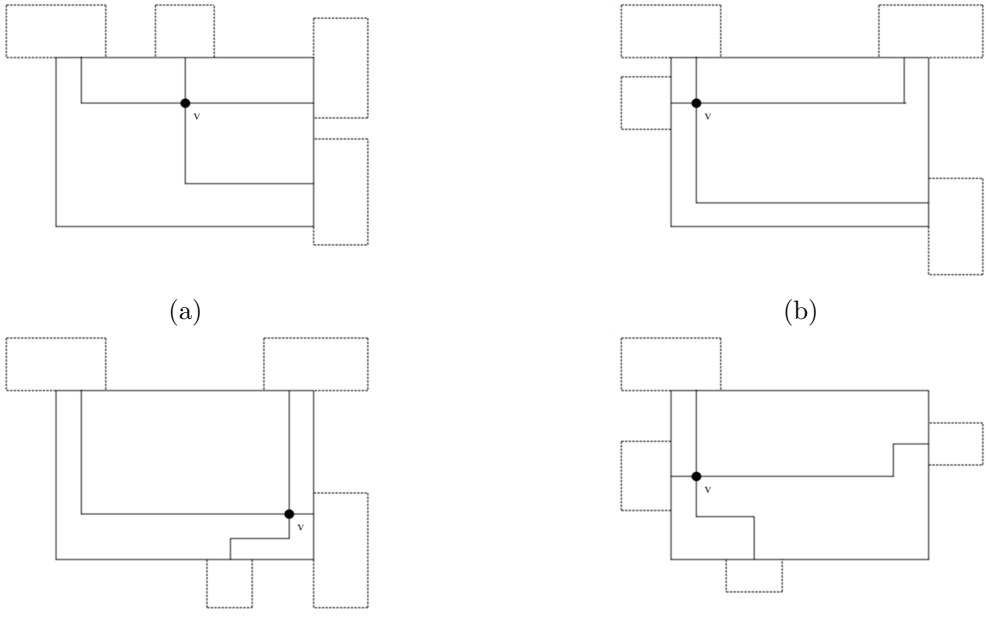
Figure 5.7: v has degree 3.



(a) (b)

(c) (d)

Figure 5.8: v has degree 4.



(a) (b)

(c) (d)

Figure 5.9: v has degree 4.

Algorithm 5.1 Orthogonalization

```
1: procedure ORTHO(R(G) = (Γ, f))
2:     for all non-gate rectangle f(v) ∈ Γ do
3:         Determine the exact configuration among those listed in Fig. 5.6 - 5.9
4:         Create a point p_v with appropriate coordinates and add it to OT[f(v)]
5:         for all rectangle f(u), u ∈ N(v) do
6:             if the orthogonal transform of f(u) has not been created yet then
7:                 Create a point p_v^u on the window between f(u) and f(v)
8:                 Add p_v^u to OT[f(v)] and OT[f(u)]
9:             end if
10:        end for
11:        Draw the polygonal chains from p_v to the points on the windows
12:    end for
13:    for all gate g do
14:        Let f(v) and f(u) be the rectangles adjacent through g
15:        p_v^g ← point on the window between f(v) and g
16:        p_u^g ← point on the window between f(u) and g
17:        Draw a polygonal chain between p_v^g and p_v^g
18:    end for
19: end procedure
```

For each rectangle $f(v)$, the algorithm applies the following procedure:

1. The correct orthogonal transform among those illustrated in Fig. 5.6 - 5.9 is determined, based on the degree of v and the position of the rectangles adjacent to $f(v)$.
2. A point p_v is placed inside $f(v)$ according to the configuration found in the first step. A point p_v^u is added on the window between $f(v)$ and each adjacent rectangle $f(u)$ such that $(u, v) \in E(G)$, if the orthogonal transform of $f(u)$ has not been created yet. Finally, p and p_v^u are added to the entry of OT corresponding to $f(v)$.
3. The polygonal chains between p_v and p_v^u are drawn, for each $u \in N(v)$

Since the degree of v is at most 4, the orthogonal transform of a rectangle is created in constant time. Thus, the cost of the algorithm is $O(n)$.

It is not difficult to verify that each polygonal chain connecting two vertices has at most 4 bends; moreover, each edge linking two vertices through a gate have at most 2 bends. In the worst case, there may be $O(n)$ gates; therefore, the total number of bends is $O(6n)$. Clearly, this algorithm adds more bends than needed. Some heuristic approaches to decrease them have been described in [Cos04].

5.1.3 Bus-Mode Drawing

In this section we propose a new drawing style (called *bus-mode drawing*), which extends orthogonal drawing to graphs with degree greater than 4 [ADQB07]. The current approaches (box orthogonal drawing, Kandinski and quasi-orthogonal drawing, described in Chapter 1) are good solutions, but hardly deal high degree graphs. Typical issues, in fact, are poor angular resolution and a high number of bends. The effectiveness of an orthog-

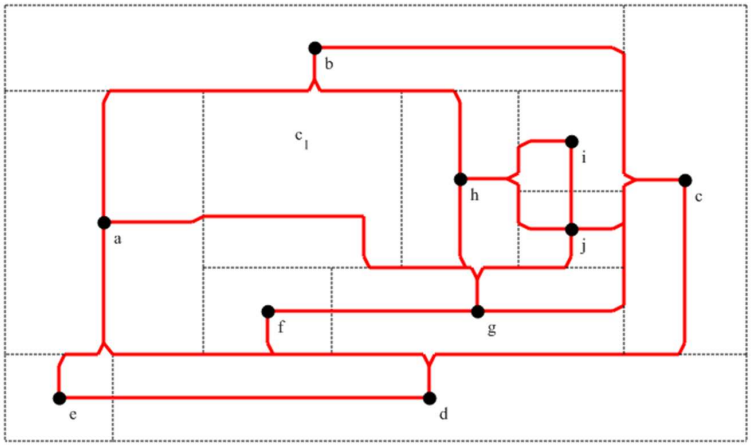


Figure 5.10: A bus-mode drawing.

onal drawing rely on the fact that the angular resolution is $\frac{\pi}{2}$, which makes it readable and pleasant to see. It would be desirable to maintain such an angular resolution also for vertices with degree higher than 4. A bus-mode drawing achieves this goal by grouping the edges incident with a vertex into up to 4 *bundles* (Fig. 5.10). Since each bundle may contain several edges, it may not be clear whether two nodes are adjacent or not. To eliminate the ambiguity, the bundles are provided with curved corners which indicate the path of each edge. Thus, an edge is either an horizontal segment or a vertical segment or a polygonal chain such as the one depicted in Fig. 5.11. Each bundle is like a bus which conveys a group of wires between two electronic devices; hence, the name of this representation. Creating a bus-mode representation from $\mathscr{R}(G)$ is easier than creating an orthogonal drawing. In fact, the only bends which can occur are the curved corners on the bundles; however, unlike bends, they do not negatively impact on the drawing and thus minimizing them is of little concern. Therefore, a point is placed at the center of each rectangle; if two points have the same $x-$ or $y-$coordinate they are joined with a straight edge. For each point p_v, representing vertex v, at most four bundles, leaving p_v and running parallel to the $x-$ and $y-$axis, are created, based on the following rules:

- An upward bundle if and only if $north(v) \neq \emptyset$.

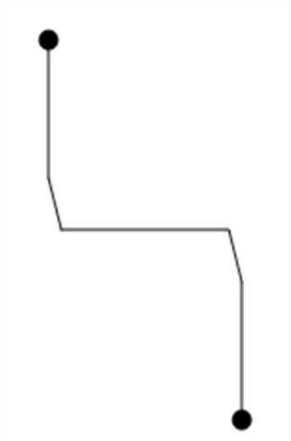


Figure 5.11: A polygonal chain in bus-mode drawing.

- A leftward bundle if and only if $west(v) \neq \emptyset$.
- A rightward bundle if and only if $east(v) \neq \emptyset$.
- A downward bundle if and only if $south(v) \neq \emptyset$.

As said before, each bundle is terminated by at most two curved corners, which indicate the direction that the bundle follows when leaving $f(v)$. If the rectangles $f(u)$, such that $u \in N(v)$, can be reached following only one direction, the bundle has only one curved corner. After leaving $f(v)$, a bundle runs along the sides of the rectangles and enters a rectangle (through a curved corner) only to reach an endpoint. As for gates, they can be treated as normal vertices; once the drawing has been created, the vertices corresponding to the gates can be replaced by a curved corner.

5.2 Drawing of Clustered Graphs

The visualization of large hierarchical clustered graphs has been extensively studied in graph drawing. The issues arising in this context are pretty much the same as in the case of plain graphs. However, clusters are a third item to draw, besides nodes and edges; their representation should not confuse the drawing and make clear how the graph is structured. Usually, a cluster is visualized as a closed region enclosing the subgraph induced by its vertices. It is desirable, from an aesthetic point of view, to use same shaped regions. Common drawing styles are *NC-Drawings* (which use non-convex regions), *C-Drawings* (using convex regions) and *R-Drawings* (which use rectangles).

In the next subsections we give a rapid round-up of the most important approaches in clustered graph visualization. Both two-dimensional and three-dimensional drawings have been widely investigated. Which one is preferable depends on the application using the visualization. As a general rule, applications using large and highly structured graphs (with a complex hierarchy among clusters) would be rather in favour of a 3D drawing, whereas 2D representations would be more suitable for those dealing with smaller graphs.

5.2.1 Two-Dimensional Drawing

In a two-dimensional representation, the underlying graph is drawn as usual and a cluster as a closed region enclosing the drawing of the subgraph induced by the vertices of the cluster. Important results were obtained by Eades et al. [Fen96, EFL96, EF97, EFN99, EFH00, FCE95a], extending the use of straight-line drawing and orthogonal drawing to HCGs. Feng's PhD thesis contains a remarkable collection of all these results [Fen97].

Fig. 5.12 is an example of *orthogonal grid rectangular cluster drawing*

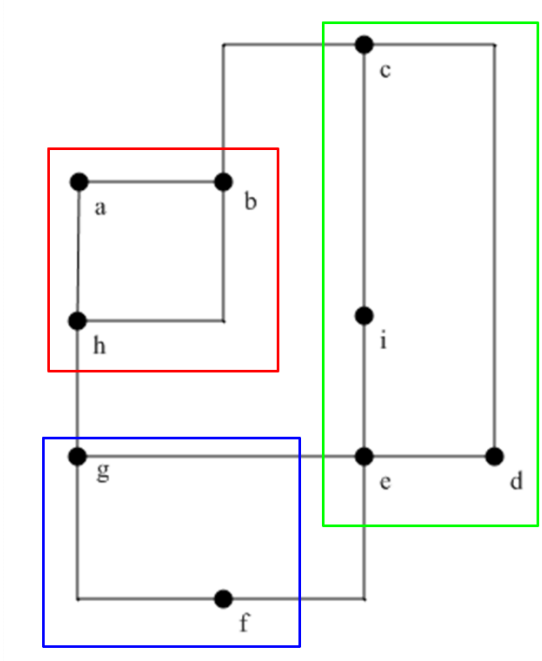


Figure 5.12: Orthogonal R-Drawing of a Clustered Graph.

The algorithm presented in [EF97, EFN99] creates a R-Drawing of a HCG in $O(n)$ time. The drawing has area $O(n^2)$ and each edge has at most three bends. The algorithm consists of three steps. First, a visibility representation of the underlying graph is created. This

is done by computing a special st-numbering of the graph called *c-st numbering*. Given a biconnected graph with n vertices and an edge (s, t), the *st-numbering* of the graph is defined as follows. The nodes are numbered from 1 to n, assigning number 1 to node s (called *source*) and number n to node t (denoted as *sink*); the other nodes are numbered in such a way that every node (other than s and t) has a neighbour with smaller st-number and a neighbour with larger st-number. A c-st numbering is obtained by imposing that vertices belonging to the same cluster are numbered consecutively. On the second stage, the visibility representation is turned into an orthogonal drawing, transforming each segment representing a vertex into a point in the plane. As a result, all edges have at most 3 bends, except of edges linking a source of degree 4 with a sink of degree 4. Applying local changes to the drawing, the number of bends of these edges is reduced to 3.

Another remarkable result regards straight line drawings of HCGs. Eades et al., in fact, proved that every c-planar HCG admits a planar straight-line drawing such that clusters are drawn as convex polygons [EFL96]. Moreover, they gave a linear-time algorithm, which turns the input HCG into a hierarchical graph[1] (using a c-st numbering) and computes a straight-line drawing of it using the algorithm described in [ELT96].

An interesting strategy that combines visualization and navigation is presented in [EFH00]. At any time, only an *abridgement* (that is a small portion) of the HCG is shown. Each cluster is represented as a macro-vertex, which hides the vertices belonging to that cluster. The user can change the abridgement by expanding/collapsing the macro-vertex. The abridgement is drawn with a particular force-directed method, which consists of three forces. An *internal spring* acts between pairs of adjacent nodes belonging to the same cluster, keeping them as close as possible. An *external spring* acts between pairs of adjacent nodes belonging to adjacent clusters. Finally, a *virtual spring* exerts a force along *virtual edges*, which connect each node with the macro-node representing the parent cluster. When a macro-vertex is collapsed, the virtual and the internal spring are strengthened; to help the user to preserve the *mental map*, clever animations are applied when updating the drawing.

Another method which aims at potentiating both visualization and interaction is described in [JAY05]. The HCG is represented by a compound fisheye view of the graph and a treemap, describing the hierarchy of the clusters. The compound fisheye view visualizes only the vertices of the underlying graph belonging to a selected focus area. The other vertices are hidden in the macro-vertices representing their enclosing clusters; the farther is a node from the focus, the closer is the visualized macro-vertex to the root of the hierarchy tree. Thus, the graph is visualized with successively less details, as the distance from the focus increases. This is the philosophy behind the traditional fisheye view, but is realized without using any distortion. A *treemap* (Fig. 5.14) is a powerful "space-filling

[1] A *hierarchical graph* $H = (V, E, \lambda, k)$ is a directed graph (V, E) such that each node v is assigned an integer $\lambda(v) \in \{1, ..., k\}$ with the property that if $(u, v) \in E$ then $\lambda(u) > \lambda(v)$.

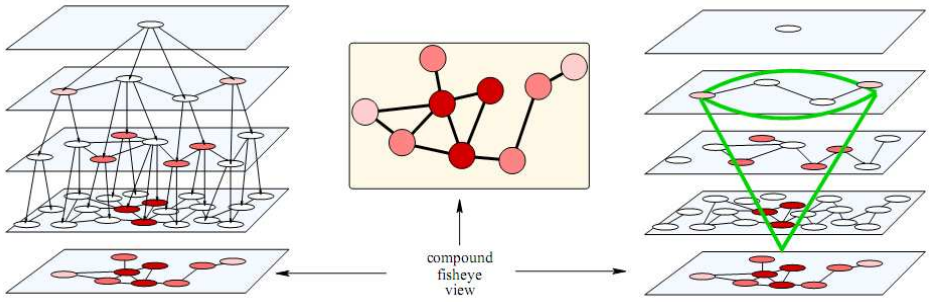


Figure 5.13: The second image from the left is a compound fish-eye representation. It is obtained from the intersection of the multilevel view of the clustered graph with an inverted cone centred at the focus.

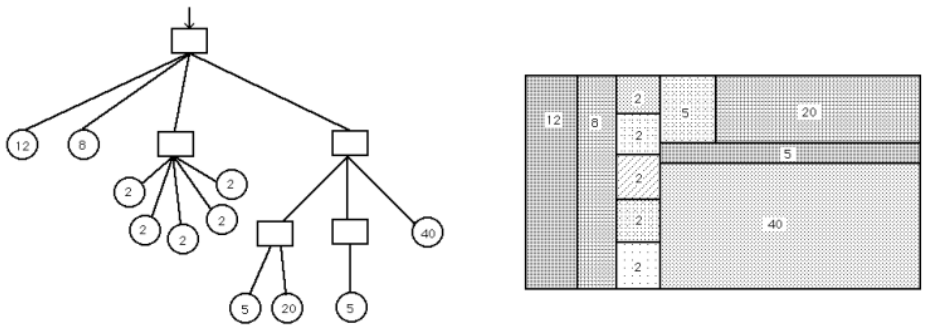


Figure 5.14: Visualization of a tree with a treemap. Image taken from [Shn92].

visualization method" [Shn92] for representing hierarchical data sets. In our opinion, at the moment treemaps are the best way to draw a tree. Traditional node-link representations, in fact, tend to be uneffective with large trees, which have a high-level hierarchy. A treemap, on the other hand, is a compact 2D representation, more pleasant to see, and provides a good navigability. The interested reader is warmly invited to take a look at http://www.cs.umd.edu/hcil/treemap/, which details the history of treemaps and its possible applications. With the compound fisheye view, a treemap helps the user to navigate the HCG. In fact, the cluster being currently visited is highlighted on the treemap, which prevents the user to loose sight of the context.

5.2.2 Three-Dimensional Drawing

The effectiveness of a 2D representation of a HCG decreases while the cluster hierarchy becomes deeper. For this reason many researchers concentrated on three-dimensional visualization, with all the upsides and downsides described in Chapter 1. A *multilevel representation* combines the visualization of the hierarchy tree and the underlying graph of a HCG [Fen96]. Each level of the tree is a plane containing the drawing of the nodes and macro-nodes at that level (Fig. 5.15). This representation makes immediately clear

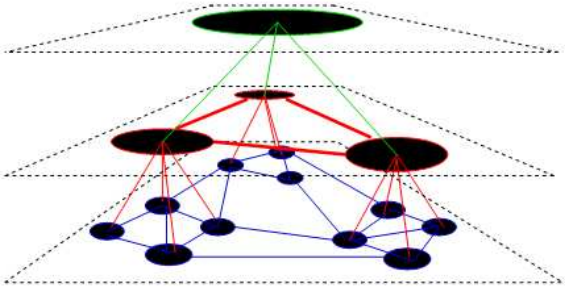


Figure 5.15: A multilevel representation of a HCG. Image taken from [Fen96].

the distribution of the clusters in the hierarchy tree and, at the same time, allows to have a global vision of the underlying graph. A similar approach is described in [FT04]. In this case, the edges (and not the vertices) are drawn on different planes (Fig. 5.16): intracluster edges lay on the lower plane and intercluster edges in the upper plane. To make the content of the cluster always visible, each cluster is represented as a semi-transparent pyramid.

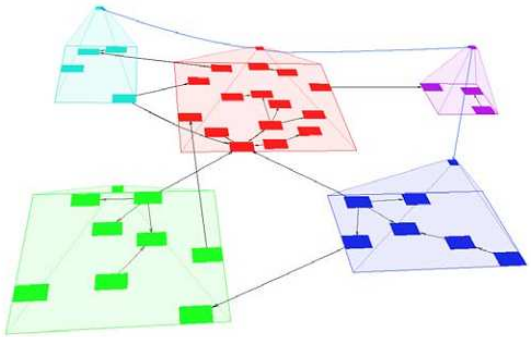


Figure 5.16: An alternative multilevel view. Image taken from [FT04].

Finally, Balzer and Deussen presented an innovative approach based on a concept, borrowed from computer graphics: *level-of-detail* [BD07]. When rendering a 3D scene, the

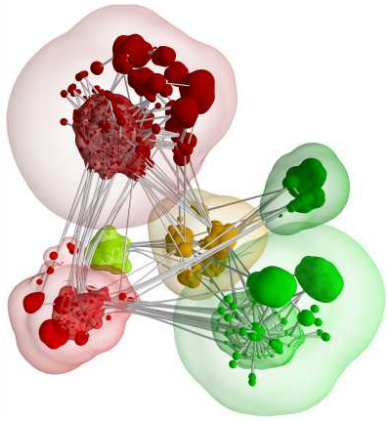


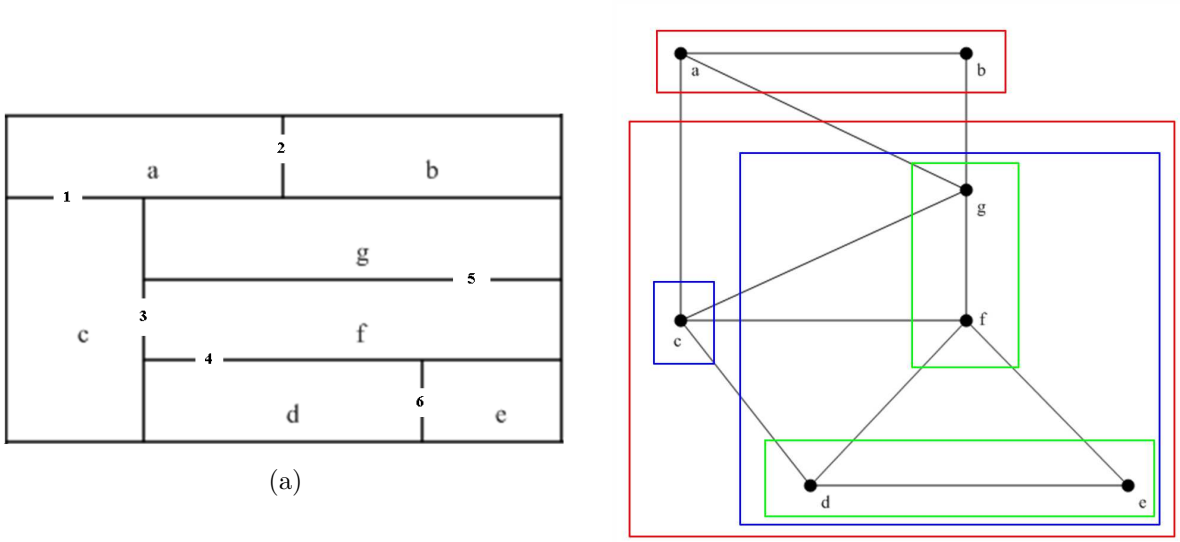
Figure 5.17: Level-of-detail representation.

complexity of some objects is decreased according to metrics such as the distance from the viewer, perspective or eye-space speed or position. This unburdens computations needed to visualize and update the scene. In the context of graph visualization, level-of-detail aims at representing a cluster as a single object, but not necessarily as a macro-vertex which completely hides its structure. Rather, the vertices in a cluster are drawn with a level of detail that is proportional to their distance from the focus (Fig. 5.17). Moreover, the shape of each cluster adapts to the distribution of its interior vertices; this gives an idea of the internal structure of a cluster, even when its vertices are hidden.

5.3 C-Rectangular Dualization

A sliceable rectangular dual (Chapter 3) can be seen as a hierarchically clustered rectangular dual. In fact, it can be hierarchically decomposed into smaller sliceable rectangular duals, the hierarchy being described by a BSP tree. Moreover, a sliceable rectangular dual seems to "suggest" a partition of the vertex set of the primal graph (Fig. 5.18). Alternatively, a sliceable rectangular dual is the rectangular dual of a HCG, such that the rectangles corresponding to the vertices belonging to the same cluster form a rectangular dual of that cluster.

Looking at several rectangular duals, it is not uncommon to find groups of subrectangles which are themselves rectangular duals of some subgraph of the primal graph. We call such groups *clusters* in the rectangular dual. Thus, even non-sliceable rectangular duals may suggest a clustering of the primal graph. We remark that rectangular dualization is not meant to be a clustering method. Often graphs are clustered based on semantic constraints, which rectangular dualization could not enforce. However, the partition of the



(a) (b)

Figure 5.18: (a) A sliceable rectangular dual. (b) The corresponding primal graph, drawn as a HCG, where clusters are suggested by the rectangular dual.

vertex set proposed by a rectangular dual may be useful when the goal of clustering is only to reduce the number of nodes to visualize, regardless of the semantic of the graph.

Throughout the thesis, we have extensively stressed the importance of rectangular dualization in several applications. In particular, nice drawings of plane graphs can be easily created from a rectangular dual. In order to extend these benefits to CGs and HCGs, we define a new concept called *hierarchically clustered rectangular dual*, or, in short, *c-rectangular dual*, on the analogy of c-planarity and c-connectivity. A *c-rectangular dual* of a HCG $\mathscr{G} = (G, T)$ is a rectangular dual of the underlying graph G such that for each cluster ν the rectangles corresponding to the vertices in $V(\nu)$ form a rectangular dual of $G(\nu)$ (Fig. 5.19 and 5.20). C-rectangular dualization is a visualization method for HCGs. It represents the hierarchy tree as a treemap, while displaying the nodes of the underlying graph, along with their adjacencies. Therefore, it is not inappropriate to think of a c-rectangular dual as a "graphmap", that is a generalization of treemaps to graphs. As treemaps proved to be effective in representing structured data, a treemap-like visualization may improve the readability of a large HCG, containing a complex hierarchy of clusters. Fig. 5.19 shows just an example. A different colour is used to enhance the clusters at a given level in the hierarchy tree. Instead of colours (which may confuse the drawing), 3D effects can be used, to enhance the position of clusters in the hierarchy tree. Finally, collapsing and expanding clusters is trivial in a c-rectangular dual. To collapse a cluster, in fact, it suffices to hide the rectangles forming the rectangular dual of that cluster, while maintaining its enclosure rectangle. The adjacent rectangles are not minimally concerned.

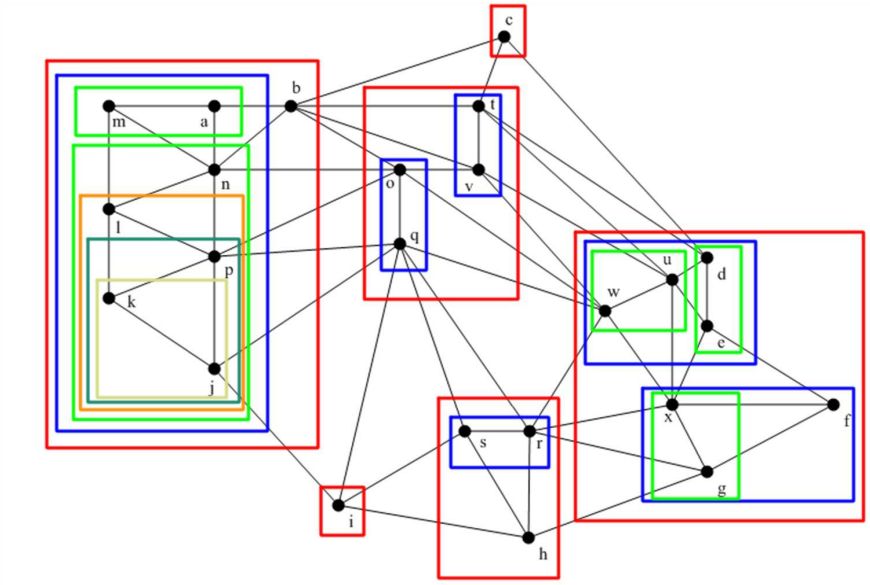


Figure 5.19: A HCG.

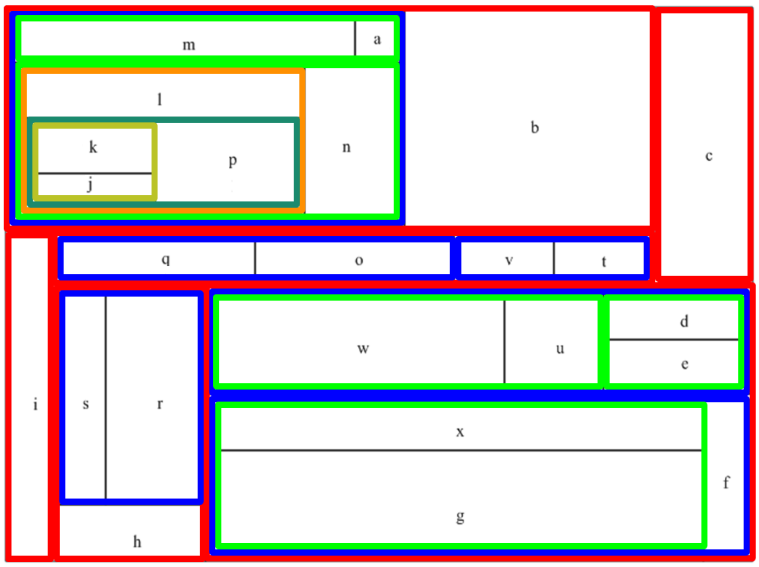


Figure 5.20: A c-rectangular dual of the HCG depicted in Figure 5.19. Clusters having the same nesting level in the hierarchy tree have the same colour.

Conversely, expanding a cluster means showing the rectangles inside the enclosure rectangle. Moreover these changes preserve the "mental map", which undoubtedly results in a better navigability of the graph.

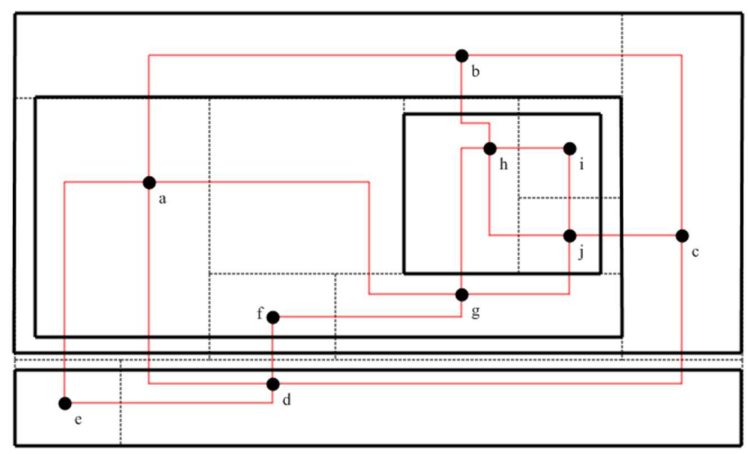


Figure 5.21: An orthogonal drawing of a HCG taken from its c-rectangular dual.

Despite all these advantages, most people still prefer a more traditional node-link visualization. This is not a pure aesthetic whim. A node-link visualization may turn out to be more suitable for some applications. We remark that no drawing-style is universally recognized as superior to the others; the use of one instead of another only depends on the requirements of the application. In this sense, rectangular dualization gives another important contribution in graph visualization. It does not impose a drawing style and is general enough to allow an easy switch between different drawing styles. Thus, a node-link representation of a HCG can be created from its c-rectangular dual, in the same way as described in Section 5.1 (Fig. 5.21). Exactly the same algorithms are used for drawing both a plain graph and a HCG. After the creation of the drawing of the underlying graph, in fact, only one more step is required to draw the clusters. However, this step is trivial, as each cluster in the c-rectangular dual is itself a rectangular dual; thus, the enclosure rectangle bounds the region containing the vertices of the cluster in the primal graph.

5.4 3D Electronic Institution Visualization

People keen on computer games are certainly aware of the evolution 3D Virtual Worlds have been undergone in the last decades. Recently, virtual reality is making a comeback because of the success of Second Life. In Second Life, people "live" in a virtual environment,

sharing their experiences and seeking new friends and places to discover. 3D modeling allows the imitation of real-life objects, which makes the experience of the user closer to reality. One drawback of Virtual Worlds is the lack of methodological support for their design and implementation. In particular, there is little or no support to structure the interactions among the participants. To this extent the concept of *3D Electronic Institution* was developed by the eMarkets research group[2] [BD06, BES$^+$07, Bog07].

In real world, an institution is something upon which a society is built up and organized. An institution helps the cooperation among participants of a society and enforces some rules, which each participant must comply with. 3D Electronic Institutions (EI) bring this concept to 3D Virtual Worlds. EIs provide mechanisms for creating and enforcing the rules in a Virtual World, as well as visualizing it. More precisely, an EI can be defined as a software system composed of autonomous entities (human or software agents) which interact according to predefined conventions and rules that must be enforced [Bog07].

Three steps are required to generate a 3D Virtual World based on a EI: *specification*, *annotation* and *generation*. On first step, the interactions among participants are specified. In most Multi-Agent Systems, this is achieved by assuming that each participant has a predefined behaviour. In other words, each participant is specifically "programmed" to behave in a predefined fashion. In 3D EIs, on the contrary, participants are supposed to be heterogeneous and self-interested [BES$^+$07]; it is the Institution that is entitled to care about their behaviour. Thus, each participant is free to do whatever he wants up to the limitations imposed by the Institution. The limitations on user behaviour are determined by three types of conventions: conventions on language (*Dialogic Framework*), conventions on activities (*Performative Structure*) and conventions on behaviour (*Norms*). The dialogic framework defines the language agents should use while interacting. Moreover, as it would be impossible to tailor rules for each individual participant, a set of *roles* is defined. Each participant has a role and each role must comply with a set of norms. The Performative Structure defines the activities which the agents can do. Each activity is performed by an agent interacting with the other agents. These interactions within an activity are called *scenes*. Finally, the norms are the rules which each role has to comply with. The specification alone does not provide any information for visualizing the institution. Therefore, on the annotation stage, the specification is enriched with additional graphical elements, such as textures and 3D objects. In general, a EI is usually represented as a virtual building, composed of as many rooms as the number of scenes in the performative structure. Finally, the Virtual World is generated, based on both specification and annotation.

The specification of a EI is supported by a set of tools called EIDE (Electronic Institutions Development Environment) [BES$^+$07]. EIDE includes five utilities. ISLANDER provides a graphical interface to specify the Institution [BES$^+$07, EdlCS02]; SIMDEI is a simulation

[2] http://research.it.uts.edu.au/emarkets/

environment in which a specification can be quickly tested, without the need of generating the whole virtual world, aBUILDER allows to easily program software agents participating to the EI; finally, AMELI uses the specification created by ISLANDER and mediates the interactions among participants while enforcing the institutional norms.

5.4.1 An Example

Before describing what rectangular dualization has to do with an EI, we present a small example (the Trading Institution) taken from [BES+07]. The Trading Institution is a virtual building in which trades take place. Each participant has one of the following roles: receptionist, room manager, guest, seller and buyer. The receptionist is an employee whose duty is to greet guests and help them in registering to the institution. The room manager is another employee which monitors the interactions between the participants. A guest is a participant who has not filled his registration form yet. After doing this, a guest may become either seller or buyer.

This building is composed of three rooms (scenes):

Registration Room. Guests are welcomed by a receptionist and register themselves as either buyers or sellers.

Meeting Room. It is used for social interactions among buyers.

Trade Room. It is the place where different kind of auctions are conducted.

Fig. 5.22 shows the Performative Structure graph of the Trade Institution. The rectangular-shaped nodes represent the rooms of the institution. The blue ones are *functional* rooms, which have a precise role within the institution. Nodes *Root* and *Exit* are just used as entry and exit points and have no particular meaning. For this reason, often they are not even visualized in the virtual building. The triangular-shaped nodes are *transitions* between rooms. The labels above each edge define the role flow of the participants. For instance, the label "rm:RoomManager" on the edge leaving the Root node means that only the RoomManager is allowed to go to the transition node on the top, without having to register himself. The other participants have to first enter the registration room.

5.4.2 Generating the Virtual World

At this point it should be clear that, from our point of view, the most important part of the specification is the Performative Structure. The generation of a virtual building is

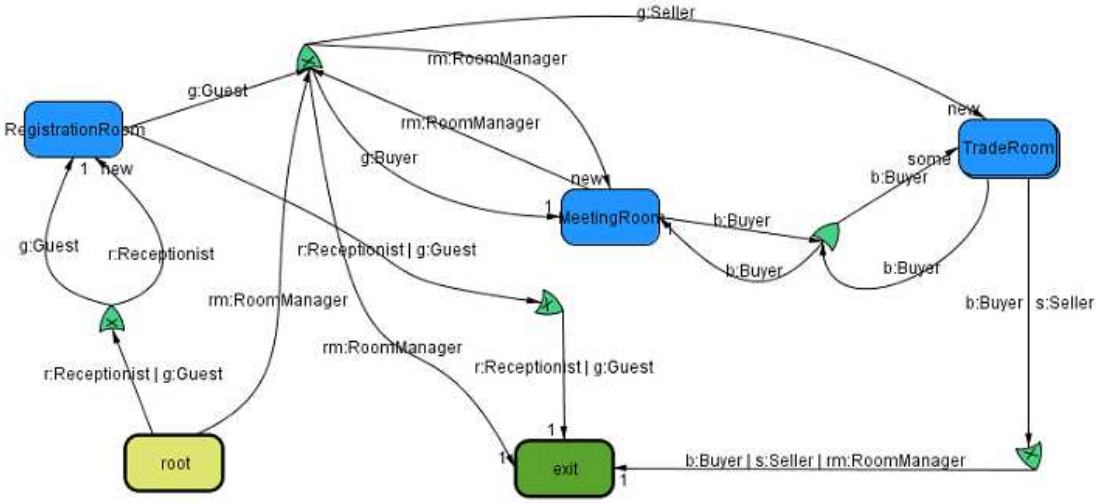


Figure 5.22: Performative Structure of the Trading Institution.

done upon a floorplan of the building, which is the rectangular dual of the Performative Structure Graph.

Creating a rectangular dual of the performative structure graph is not trivial. First, the graph may contain multiedges, which have to be removed. Second, often transitions are not visualized in the virtual building. Thus, before creating the rectangular dual, transition nodes are removed. This changes the performative structure as follows. Let r be a node connected to a transition node t and let t be connected to nodes $r_1, ..., r_k$: after removing t, edges $(r, r_1), ..., (r, r_k)$ are created[3]. Finally, Root and Exit are usually removed.

Despite of all these simplifications, the input graph may not be planar. A planarization algorithm is thus needed. An easy solution consists in removing edges until a planar graph is obtained. In this context, it is not desirable to loose adjacencies between any two rooms. A better approach is to create an embedding of the graph and add a crossover vertex at each point where two edges cross. In this way, two rooms are adjacent through a gate (which can be thought of as a corridor in the virtual building). After creating a plane embedding, the algorithm described in Section 3.7.1 is used to turn the Performative Structure into a PTP graph and create a rectangular dual (Fig. 5.23).

[3]The Performative Structure if a directed graph, thus, a vertex v is connected to w only if there is an edge directed from v to w. The direction of the edges is not considered when creating the rectangular dual.

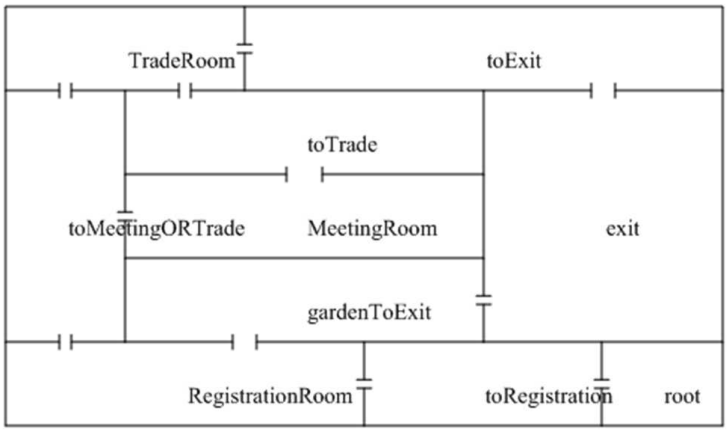


Figure 5.23: Floorplan of the Trading Institution created with OcORD.

We conclude with some important remarks. First, the virtual building can be seen, with a bit of imagination, as a different visualization of the Performative Structure. Indeed, when talking of graphs, usually people think about the traditional node-link representation. However, there are potentially infinite ways to visualize a graph, based on the context. In this case, the relationships between rooms is better visualized (and understood) as a virtual building. Second, the floorplan may be more flexible if each room is allowed to have a shape different than rectangles. In this way, there would not be the need of representing gates as corridors. Finally, a rectangular dual does not have the control over the size of the rooms. It would be interesting to use a rectangular (or rectilinear) cartogram, to tailor a more a precise floorplan.

Chapter 6

C-Rectangular Dualization

This chapter describes a generalization of rectangular dualization to hierarchically clustered graphs (HCGs), called *c-rectangular dualization* on the analogy of *c-planarity* and *c-connectivity*. Not every HCG admits a c-rectangular dual. Thus, in the first part of the chapter we discuss the admissibility conditions. In the second part we describe an algorithm which, given a c-planar c-connected HCG, creates a c-rectangular dual, after enforcing the admissibility conditions.

6.1 Preliminaries

Let H be a plane graph, $S \subseteq V(H)$ and $\mathscr{R}(H) = (\Gamma, f)$ a rectangular dual of H. The following:

$$\mathcal{R}(\Gamma, f, S) = \{R \in \Gamma | f^{-1}(R) = u \wedge u \in S\}$$

is the set of rectangles of $\mathscr{R}(H)$ corresponding to the vertices in S. Henceforth, notation $f|_S$ refers to a function f whose domain is restricted to a given subset S.

Definition 6.1. *A* c-rectangular dual $\mathscr{R}(\mathscr{G}) = (\Gamma, f)$ *of a c-planar c-connected clustered graph* $\mathscr{G} = (G, \mathscr{C})$ *is a rectangular dual of* G *such that for each cluster* $C_i \in \mathscr{C}$, $1 \leq i \leq k$, $(\mathcal{R}(\Gamma, f, C_i), f|_{C_i})$ *is a rectangular dual of* $G[C_i]$.

Definition 6.2. *A* c-rectangular dual *(HCRD)* $\mathscr{R}(\mathscr{G}) = (\Gamma, f)$ *of a c-planar c-connected HCG* $\mathscr{G} = (G, T)$ *is a rectangular dual of* G *such that for each cluster* ν*:*

1. $(\mathcal{R}(\Gamma, f, V(\nu)), f|_{V(\nu)})$ *is a rectangular dual of* $G(\nu)$.

2. *If μ is a subcluster of ν, $\mathcal{R}(\Gamma, f, V(\mu)) \subset \mathcal{R}(\Gamma, f, V(\nu))$*

Clearly, a HCG admits a c-rectangular dual only if its underlying graph admits a rectangular dual. Let G be a PTP graph. In Section 5.3 we pointed out that even non-sliceable rectangular duals of G contain subsets of rectangles (called *clusters*) which are rectangular duals of subgraphs of G. Thus, a first step to define the admissibility conditions and an algorithm for c-rectangular dualization is to understand how clusters in rectangular duals are created.

Kant and He proved that there is a one-to-one correspondence between a REL (T_1, T_2) in G and its rectangular dual (Theorem 3.6). Therefore, we should expect that the creation of clusters in $\mathscr{R}(G)$ depends on some property of the REL. Intuitively, if a cluster of $\mathscr{R}(G)$ is a rectangular dual of a subgraph $G[S]$ (induced by $S \subset V$) of G, the REL on G used to compute $\mathscr{R}(G)$ "induces" a REL on $G[S]$. Let R be a simple closed region that encloses the drawing of $G[S]$ (Fig. 6.1).

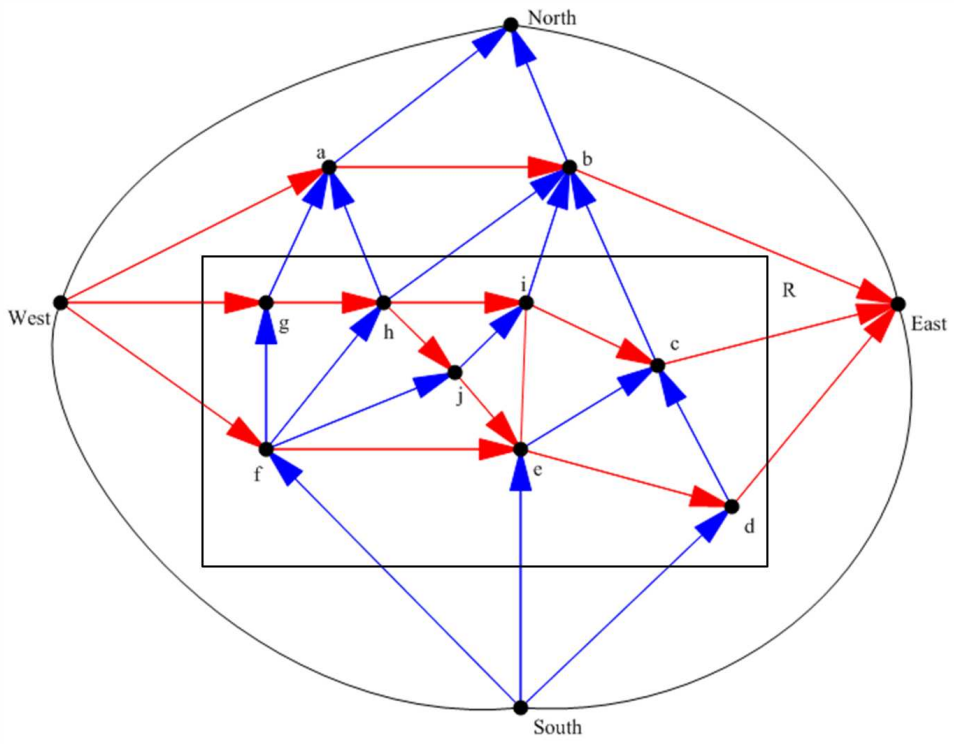


Figure 6.1: An REL induced on the subgraph $G[S]$ enclosed in rectangle R.

The edges connecting a vertex of S to a vertex of $V(G)\backslash S$ can be sorted around R in counterclockwise order, provided that they cross the boundary of R only once. Thus, (T_1, T_2)

induces a REL on $G[S]$ if the edges crossing the boundary of R appear in counterclockwise order as follows: a set of edges in T_1 leaving R, a set of edges in T_2 entering R, a set of edges in T_1 entering R and a set of edges in T_2 leaving R.

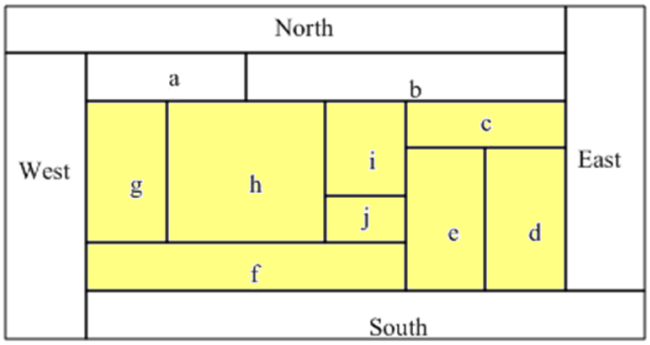


Figure 6.2: Rectangular dual obtained from the REL shown in Figure 6.1. The rectangles corresponding to the vertices of $G[S]$ (in yellow) form a rectangular dual of $G[S]$.

The question naturally arises whether there is any correlation between the existence of an induced REL in G and the presence of a cluster in $\mathscr{R}(G)$ (Fig. 6.2). Theorem 6.1 answers positively to this question.

Theorem 6.1. *Given a PTP graph G, a rectangular dual $\mathscr{R}(G) = (\Gamma, f)$ obtained from (T_1, T_2) and a set $S \subset V$ of inner vertices of G, the rectangles in $\mathcal{R}(\Gamma, f, S)$ form a rectangular dual of $G[S]$ if and only if $G[S]$ is connected and (T_1, T_2) induces a REL on $G[S]$.*

Proof. Let R be the enclosure rectangle of $(\mathcal{R}(\Gamma, f, S), f|_S)$ and let the embedding of G be created from $\mathscr{R}(G)$, as seen in the previous chapter. As $(\mathcal{R}(\Gamma, f, S), f|_S)$ is a rectangular dual of $G[S]$, $G[S]$ is connected. Moreover, R encloses the drawing of $G[S]$. All edges connecting a vertex in S with a vertex in $V(G)\backslash S$ intersect one of the four sides of R only once. If (v_1, v_2) intersects the top side of R then $v_2 \in north(v_1)$. By Lemma 3.5, $(v_1, v_2) \in (T_1, \rightarrow)$. Similarly, $(v_1, v_2) \in (T_2, \leftarrow)$ if it intersects the left side, $(v_1, v_2) \in (T_1, \leftarrow)$ if it crosses the bottom side of and $(v_1, v_2) \in (T_2, \rightarrow)$ if it intersects the right side. By definition, (T_1, T_2) induces a REL on $G[S]$.

To establish sufficiency, let M be the simple closed region which encloses the drawing of $G[S]$. The last edge $e_1 \in T_1$ leaving M and the first edge $e_2 \in T_2$ entering M in counterclockwise order must be incident with the same vertex (say $v_{top,left}$) in $G[S]$. Suppose not. Let $e1 = (x, u)$ and $e_2 = (y, v)$ such that $x, y \in V(G)\backslash S$ and $u, v \in S$; two cases are possible:

1. $x \neq y$. Since no edge between e_1 and e_2 enters or leaves R, e_1 and e_2 are incident with the same face f. However, f is not a 3−face, which contradicts that G is a PTP.

2. $x = y$. e_1 follows e_2 in the incident list of x in counterclockwise direction. Since $e_1 \in (T_1, \leftarrow)$ and $e_2 \in (T_2, \rightarrow)$, (T_1, T_2) is not a REL. Again we have a contradiction.

Similarly, the last edge in T_2 entering M and the first edge in T_1 entering M both meet at vertex $v_{bottom,left}$; the last edge in T_1 entering M and the first edge in T_2 leaving M are incident with $v_{bottom,right}$; and finally the last edge in T_2 leaving M and the first edge in T_1 leaving M meet at vertex $v_{top,right}$. We call such vertices *corners*. There can be up to four corner vertices. Without losing generality, we assume that the corner vertices are exactly four.

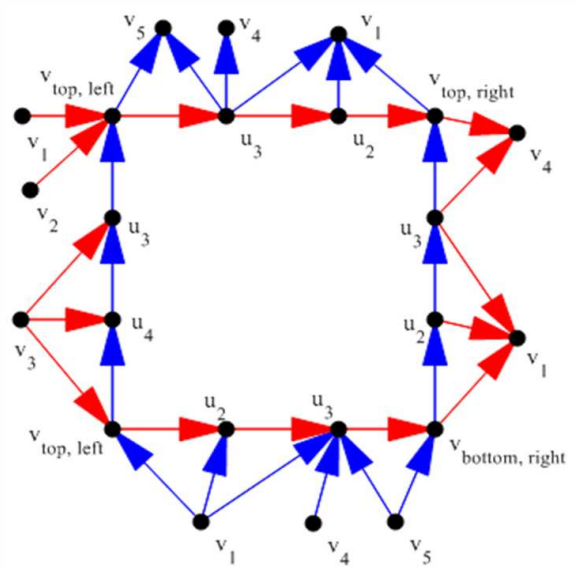


Figure 6.3: Sequence of the vertices incident with at least one edge crossing the boundary of M. The blue edges belong to T_1, the red ones to T_2.

We define

$$M[l] = ((u_i, v_i)|1 \leq i \leq k, k > 1, u_i \in S \wedge v_i \in V(G) \backslash S)$$

as the sequence of the edges crossing the boundary of M in counterclockwise order and having label l in (T_1, T_2). Since (T_1, T_2) induces a REL on $G[S]$, the edges incident with M appear in counterclockwise order as follows (Fig. 6.3):

1. Edges of $M[(T_1, \rightarrow)]$, $u_1 = v_{top,right}$ and $u_k = v_{top,left}$;
2. Edges of $M[(T_2, \leftarrow)]$, $u_1 = v_{top,left}$ and $u_k = v_{bottom,left}$;
3. Edges of $M[(T_1, \leftarrow)]$, $u_1 = v_{bottom,left}$ and $u_k = v_{bottom,right}$;
4. Edges of $M[(T_2, \rightarrow)]$, $u_1 = v_{bottom,right}$ and $u_k = v_{top,right}$.

Given a label l, either $u_i = u_{i+1}$ or $u_i \neq u_{i+1}$ in $M[l]$. In the first case, $v_i \neq v_{i+1}$ (as G is a simple graph) and $(v_i, v_{i+1}) \in E(G)$ by Lemma 2.8. In the second case $(u_i, u_{i+1}) \in E(G)$ and $v_i = v_{i+1}$, otherwise (u_i, v_i) and (u_{i+1}, v_{i+1}) would be incident with a non-triangular face. We have that (Fig. 6.3):

1. Any edge (u_i, u_{i+1}), $1 \leq i < k$, such that u_i, u_{i+1} are incident with edges in $M[(T_1, \rightarrow)]$, belongs to T_2 and is directed from u_{i+1} to u_i.
2. Any edge (u_i, u_{i+1}), $1 \leq i < k$, such that u_i, u_{i+1} are incident with edges in $M[(T_2, \leftarrow)]$, belongs to T_1 and is directed from u_{i+1} to u_i.
3. Any edge (u_i, u_{i+1}), $1 \leq i < k$, such that u_i, u_{i+1} are incident with edges in $M[(T_1, \leftarrow)]$, belongs to T_2 and is directed from u_i to u_{i+1};
4. Any edge (u_i, u_{i+1}), $1 \leq i < k$, such that u_i, u_{i+1} are incident with edges in $M[(T_2, \rightarrow)]$ belongs to T_1 and is directed from u_i to u_{i+1}.

Consequently, referring to the Kant-He algorithm, all vertices in $M[T_1, \rightarrow]$ are incident with the same *above face*; all vertices in $M[T_2, \leftarrow]$ are incident with the same *left face*; all vertices in $M[T_1, \leftarrow]$ are incident with the same *bottom face*; and all vertices in $M[T_2, \rightarrow]$ are incident with the same *right face*. By Theorem 3.6, the union of the rectangles corresponding to the vertices in S is a rectangle in $\mathscr{R}(G)$.

□

The following theorem states the necessary and sufficient conditions for a HCG to have a c-rectangular dual.

Theorem 6.2. *A c-planar c-connected HCG $\mathscr{G} = (G, T)$ admits a c-rectangular dual if and only if there exists a REL in G inducing a REL on $G(\nu)$, for each cluster ν.*

Proof. Immediate consequence of Theorem 6.1. □

Theorem 6.2 also suggests an algorithm for checking whether a HCG admits a c-rectangular dual and, if this is the case, computing it. RELS are generated until one complying with the hypothesis of Theorem 6.2 or no suitable REL is found. This approach is clearly unsuitable, as G may have an arbitrary high number of RELs.

We now concentrate on studying c-rectangular dualization on a c-planar c-connected clustered graph (CG) $\mathscr{G} = (G, \mathscr{C})$, such that G is a PTP graph. The four exterior vertices of G ($North$, $West$, $South$ and $East$) are contraction vertices belonging to no cluster. In Section 6.4, we describe a linear-time algorithm which creates a c-rectangular dual $\mathscr{R}(\mathscr{G}) = (\Gamma, f)$ of a CG, after enforcing the admissibility conditions. In Section 6.5 we generalize the algorithm to HCGs.

6.2 Role of the Quotient Graph

An effective representation of $\mathscr{G}$ is its quotient graph Q. Each node of Q corresponds to a cluster of $\mathscr{G}$ and is referred to as *macro* (Section 2.7.2). Thus, in $\mathscr{R}(Q)$ every rectangle corresponds to a cluster in $\mathscr{G}$. Therefore, $\mathscr{R}(Q)$ can be considered a sort of "skeleton" of a c-rectangular dual of $\mathscr{R}(\mathscr{G})$. The algorithm for creating $\mathscr{R}(\mathscr{G})$ starts from this skeleton and "fills" each rectangle with the rectangular dual of the corresponding cluster.

In Chapter 2 we studied the properties of PTGs and the quotient graph of CGs. Now, we exploit that knowledge to deduce the properties of the quotient graph of a CG whose underlying graph is a PTP. Recall that, given clusters $\mathscr{C} = \{C_1, ..., C_k\}$, $1 \leq k \leq |V(G)|$, we created the sequence of graphs Q^i, $0 \leq i \leq k$, where $Q^0 = G$ and Q^i is obtained from Q^{i-1} contracting cluster C^i and removing all loops.

Lemma 6.1. *Let G be a degenerate PTG such that no loop bounds a complex $1-$cycle. Removing all loops results in a multi-PTG G' with the same vertex-connectivity as G.*

Proof. Clearly, removing a loop does not change the vertex-connectivity of G. Any loop is incident with two faces, one $1-$face and one degenerate $3-$face, that is a face whose boundary has 3 vertices and 4 edges. Removing the loop deletes the $1-$face and turns the degenerate $3-$face into a $3-$face. Since the faces incident with no loop are $3-$faces, the thesis is proven. □

Lemma 6.2. *Q^i, $0 \leq i \leq k$, is a connected multi-PTG with no loops and with an exterior $4-$cycle.*

Proof. The proof proceeds by induction on i. As the case $i = 0$ is trivial, we assume $i \geq 1$. By induction, Q^{i-1} is a connected multi-PTG with no loops and with an exterior $4-$cycle. By Lemma 2.10, the contraction of C_i creates a degenerate PTG which, by Lemma 2.13,

has no complex 1-cycle. The exterior cycle and the inner edges incident with it are not concerned by these contractions. By Lemma 6.1, removing all loops results in a connected multi-PTG. □

Corollary 6.1. *The quotient graph of $\mathscr{G}$ is a connected multi-PTG with no loops and with an exterior $4-$cycle.*

Therefore, in general we should not expect that Q is a PTP graph.

6.2.1 Well-Clustered Graphs

Whether Q is PTP does not depend on the particular embedding of $\mathscr{G}$. In fact, since G is $3-$connected, it admits just one embedding in the plane. Consequently, also Q has only one embedding in the plane. We say that $\mathscr{G}$ is *well-clustered* (*badly clustered*) if Q is (not) a PTP graph. Since clusters are given as an input, either we find a way to deal with *bad clustering* or we restrict the scope of c-rectangular dualization to just well-clustered graphs. Since *bad clustering* is not as uncommon as we would like to be, it is worth correctly addressing the problem. Two questions naturally arise: how *bad clustering* can be recognized? can it be corrected and how? It is time for more notation. Let C be a connected cluster of $\mathscr{G}$. $G\backslash C$ is the graph obtained from G by contracting C. C_G^{adj} denotes the *c-adjacency set* of C in G, that is the set of the neighbours of all vertices of C in G. $G[C \cup C_G^{adj}]$ (G_C^{adj} in short) is the subgraph of G induced by the vertices in C and C_G^{adj}. Finally, C is a *good cluster* in G if its contraction creates no separating triangle or multiedge; otherwise, it is a *bad cluster*.

Lemma 6.3. *Let G (the underlying graph of $\mathscr{G}$) be a multigraph with no loops. G_C^{adj} is connected and has an exterior cycle Ext_G^C, whose vertices belong to C_G^{adj}.*

Proof. Since C is connected, G_C^{adj} would be disconnected only if some vertex of C_G^{adj} were adjacent to no vertex of C. Suppose by contradiction that at least one vertex $v \in C$ is exterior in G_C^{adj}. The vertices adjacent to v in G either belong to C or to C_G^{adj}. Therefore, no edge incident with v is removed when creating G_C^{adj} from G. Let v_1 and v_2 consecutive in $Adj(v)$. By Lemma 2.8, $(v_1, v_2) \in E(G)$. v_1 and v_2 belong to either C or C^{adj}; thus $(v_1, v_2) \in E(G_C^{adj})$. Since G has no loops nor 2-faces, $v \notin \{v_1, v_2\}$ and $v_1 \neq v_2$. Thus, the neighbours of v induce a cycle enclosing v (Fig. 6.4). But then v can not be an exterior node in G_C^{adj}. Let Π be the cycle induced by the vertices in C_G^{adj} incident with the exterior face of G_C^{adj}. Π is the exterior cycle of G_C^{adj}. Suppose, in fact, that an exterior node w does not belong to Π. Since all vertices of C are inner vertices of G_C^{adj}, $w \notin C \cup C_G^{adj}$. Consequently, w is not a vertex of G_C^{adj}.

□

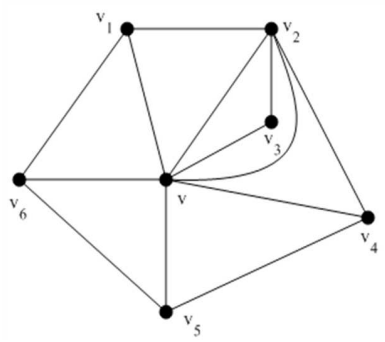


Figure 6.4: All neighbours of v belong to either C or C^{adj} and are pairwise adjacent. They induce a cycle enclosing v. Thus, v can not be an exterior node.

Henceforth, Ext_G^C will denote the exterior cycle of G_C^{adj}.

Lemma 6.4. *Let G be a multigraph PTP with no loops. If Ext_G^C contains all nodes of C_G^{adj}, then Ext_G^C is chordless.*

Proof. Let $Ext_G^C = (v_1, ..., v_k)$, $k \geq 2$, in counterclockwise order. Suppose that a chord links v_i to v_j, with $1 \leq i < j - 1 \leq k$ (Fig. 6.5). If (v_i, v_j) is outside Ext_G^C, either vertices

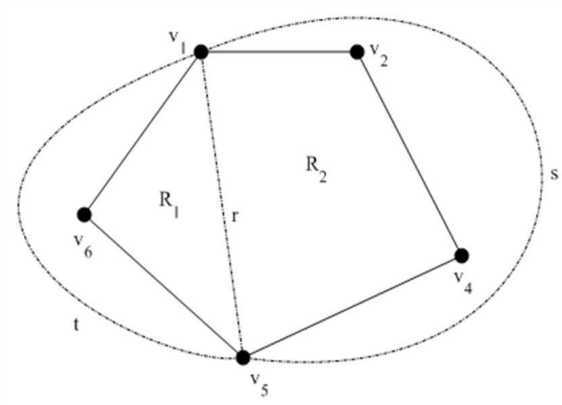


Figure 6.5: The chord joining v_1 and v_5 can be either r, or s or t. r splits the region enclosed in Ext_C^G into two regions R_1 and R_2. As any vertex of Ext_C^G is adjacent to at least one node of C and all nodes of C are enclosed in Ext_C^G, some vertices of C belong to R_1 and some to R_2 and thus C is disconnected. In the other two cases either v_2, v_4 or v_6 is not an exterior node.

$v_{j+1}, v_{j+2}, ..., v_1, ..., v_{i-1}$ or $v_{i+1}, ..., v_{j-1}$ are not on Ext_G^C, which is a contradiction. If (v_i, v_j) is enclosed in Ext_G^C, it splits the region enclosed in Ext_G^C into two regions. Consequently, C is disconnected, which again is a contradiction. □

Lemma 6.5. *Let G be a simple PTP graph. If Ext_G^C contains all vertices of C_G^{adj}, then all vertices enclosed in Ext_G^C belong to C.*

Proof. Let $Ext_G^C = u_0, ..., u_{k-1}$, $k > 3$ in counterclockwise order. Since C is connected, all vertices of C are enclosed in Ext_G^C. Suppose that the vertices of a non-empty set D, $D \cap C = \emptyset$, are enclosed in Ext_G^C. None of these vertices belongs to C_G^{adj}, as all vertices in C_G^{adj} are on Ext_G^C. Consequently, the vertices of D are adjacent to either vertices of D or vertices of C_G^{adj}. Since G is connected, at least a node $v \in D$ is adjacent to one or more vertices of C_G^{adj}. If v is adjacent to $u_i, u_j \in C_G^{adj}$, $i < j - 1$ (all subscripts are modulo k), part of the vertices of C are enclosed in cycle $(v, u_i, u_{i+1}, ..., u_{j-1}, u_j, v)$ and part in cycle $(v, u_j, u_{j+1}, ..., u_{i+k-1}, u_{i+k}, v)$ (Fig. 6.6(a)). Thus, C is disconnected, which contradicts the hypothesis. Therefore, v can be adjacent to at most two vertices u_i and

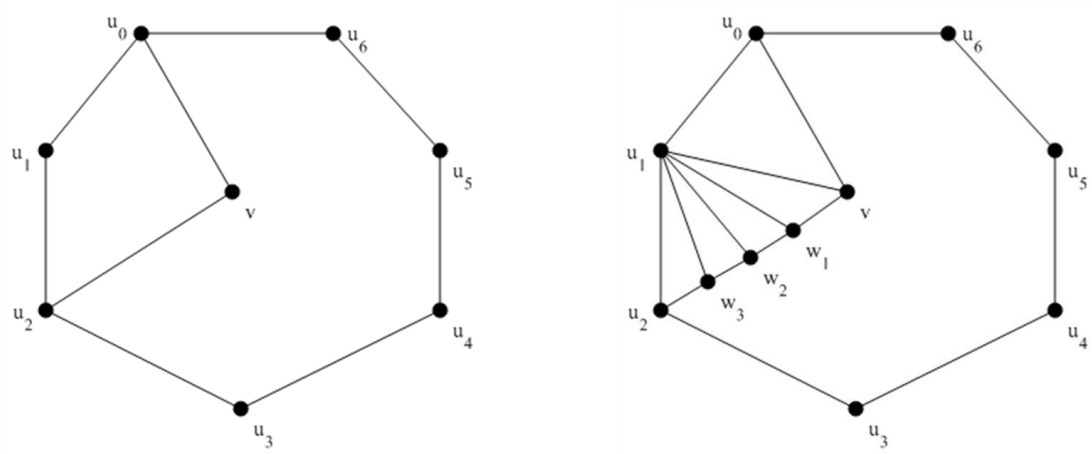


Figure 6.6: (a) If $v \in D$ is adjacent to two vertices u_0 and u_2 non consecutive on Ext_G^C, then part of the vertices of C are enclosed in cycle (u_0, u_1, u_2, v) and part in cycle $(u_2, u_3, u_4, u_5, u_6, v)$ and thus C is disconnected. (b) If $v \in D$ is adjacent to two consecutive vertices u_0 and u_1, vertices w_1, w_2 and w_3 belong to D and consequently u_1 is adjacent to no vertex of C.

u_{i+1} of C_G^{adj}, consecutive on Ext_G^C. v and u_{i+2} can not be consecutive in $Adj(u_{i+1})$, as v would be adjacent to u_{i+2}, by Lemma 2.8. Let $w_1, ..., w_l$, $l \geq 1$, be the vertices that are between v and u_{i+2} in clockwise order in $Adj(u_{i+1})$. Vertices w_j can not belong to C_G^{adj}, as G is simple and Ext_G^C, by Lemma 6.4, is chordless. By Lemma 2.8 w_1 and v are adjacent; since w_1 is an inner vertex, $w_1 \in D$. Similarly, w_1 is adjacent to w_2 and $w_2 \in D$. In general, $w_i \in D$, $1 \leq i \leq l$. Also w_l and u_{i+2} are adjacent, implying that u_{i+1} is adjacent to no vertex in C, which contradicts the hypothesis (Fig. 6.6(b)). We obtain the same contradiction, supposing that v is adjacent only to u_i.

□

A weaker result holds for G_C^{adj} when G is not a PTP.

Lemma 6.6. *Let G be a multigraph PTP with no loops. If Ext_G^C contains all vertices in C_G^{adj}, any vertex $v \notin C$ enclosed in Ext_G^C is enclosed in a complex 2-cycle induced by two vertices consecutive on Ext_G^C.*

Proof. Let D be the set of vertices enclosed in Ext_G^C such that $D \cap C = \emptyset$. Since G is connected, at least one node $v \in D$ is adjacent to one or more vertices of C_G^{adj}. Repeating the same reasoning done in the proof of Lemma 6.5, v can be adjacent to at most two vertices u_i and u_{u+1} consecutive on Ext_G^C. Since Ext_G^C is chordless, u_i can not be adjacent to any vertex of G_C^{adj} different than u_{i+1}. Suppose that there is an occurrence of edge (u_i, u_{i+1}) enclosed in Ext_G^C; if the vertices of D are enclosed in the complex $2-$cycle induced by the two occurrences of (u_i, u_{i+1}) we get no contradiction. □

The following lemma gives the necessary and sufficient conditions for a cluster to be good in $\mathscr{G}$.

Lemma 6.7. *Let G be a multigraph PTP with no loops. C is a good cluster in G if and only if all vertices enclosed in Ext_G^C belong to C.*

Proof. Let C be a good cluster and $Ext_G^C = (u_1, ..., u_k)$, $k > 3$, in counterclockwise order. To prove necessity, we suppose that D, $D \cap C = \emptyset$, contains vertices enclosed in Ext_G^C. Without losing generality, we suppose that $G[D]$ is connected. If no neighbour of the vertices of D belongs to Ext_G^C, then all vertices of D are inner vertices of subgraph $G[C \cup D]$, otherwise some faces incident with vertices of D would not be triangular. But then in G there is a complex cycle bounded by vertices of C enclosing all vertices in D; C is an invalid cluster (Lemma 2.12). Let $v \in D$ be adjacent to a vertex on Ext_G^C (say u_i) and let c be the vertex in $G \backslash C$ resulting from the contraction of cluster C. c is adjacent to all vertices of Ext_G^C. Thus, all vertices of D are enclosed either in cycle $\Delta_1 = (u_i, c, u_{i-1})$ or in cycle $\Delta_2 = (u_i, c, u_{i+1})$ or in cycle $\Delta_3 = (u_i, c)$. Δ_1, Δ_2 are separating triangles and Δ_3 is a complex 2-cycle; in any case, C would not be a good cluster.

The sufficiency is proved by induction on $|C|$. If $|C| = 1$, $G = G \backslash C$ and the lemma is trivially true. By induction, cluster C is good if $|C| \leq i$. Let $e = (v_1, v_2)$ be an intracluster edge of C, $|C| = i + 1$. If e is an edge of a complex $2-$cycle Σ, the contraction of e creates a loop bounding a complex $1-$cycle Π, enclosing the same vertices as Σ. Thus, G is a degenerate PTP (Lemma 2.10). Since all vertices enclosed in Ext_G^C belong to C, the vertices in $I(\Pi)$ are all vertices of C. Contracting the edges enclosed in Π turns Π into a non-complex $1-$cycle. By Lemma 6.1, the removal of the loop turns G into a non-degenerate PTP, where $|C| < i$. By induction the lemma is proved. Analogous considerations can be replicated if e is an edge of a separating triangle or a complex $4-$cycle. □

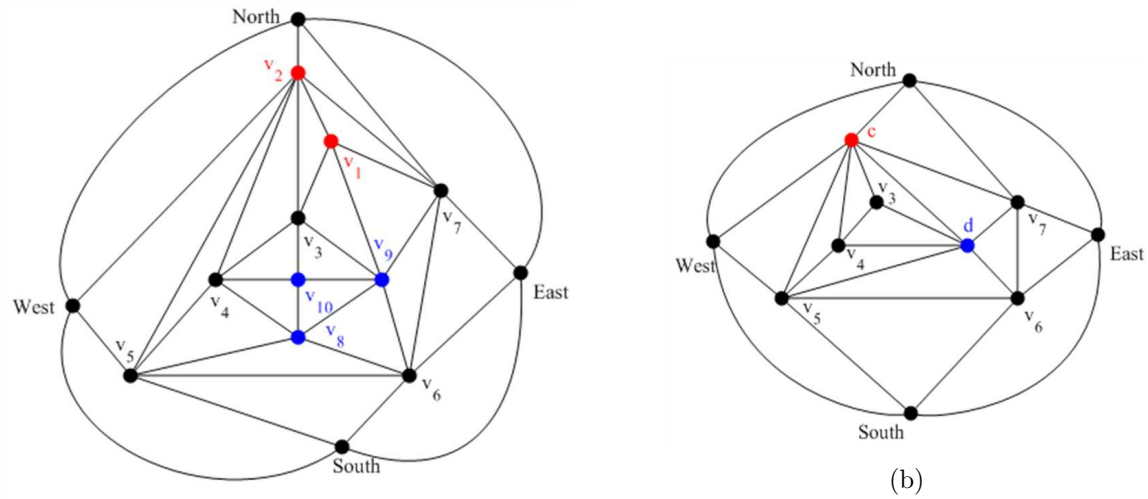


Figure 6.7: (a) A graph $\mathscr{G}_1$ with two good clusters (red and blue). (b) The quotient graph of $\mathscr{G}_1$. (c, d, v_5) and (c, d, v_4) are separating triangles. $\mathscr{G}_1$ is badly clustered although its clusters are good.

We would like to conclude that $\mathscr{G}$ is well-clustered if and only if all clusters are good in G. Unfortunately, both implications of this statement are false (Fig. 6.7 and 6.8). This depends on that the contraction of a cluster changes the goodness of the adjacent clusters. Therefore, good clusters in G may be bad in Q^1 and bad clusters in G may be good in Q^1. Clearly, the goodness of a cluster C may change only because of the contraction of an adjacent cluster. In other words, the goodness of C must be evaluated only after contracting all its adjacent clusters. Let $Q(C)$ be the graph obtained from Q by expanding only cluster C. $Q(C)$ is a connected multi-PTG with no loops and with an exterior $4-$cycle (Lemma 6.2). Moreover, $Q(C)^{adj}_C$ is connected and has an exterior cycle $Ext^C_{Q(C)}$ whose vertices belong to $C^{adj}_{Q(C)}$ (Lemma 6.3).

Lemma 6.8. *C is good in $Q(C)$ if and only if:*

1. $|Ext^C_{Q(C)}| > 3$.
2. $Ext^C_{Q(C)}$ *contains all nodes in* $C^{adj}_{Q(C)}$. *Alternatively,* $Ext^C_{Q(C)}$ *is chordless (Lemma 6.4).*
3. $Ext^C_{Q(C)}$ *encloses no separating* 2*-cycle induced by two consecutive vertices on* $Ext^C_{Q(C)}$.

Proof. By Lemma 6.6 and 6.7. □

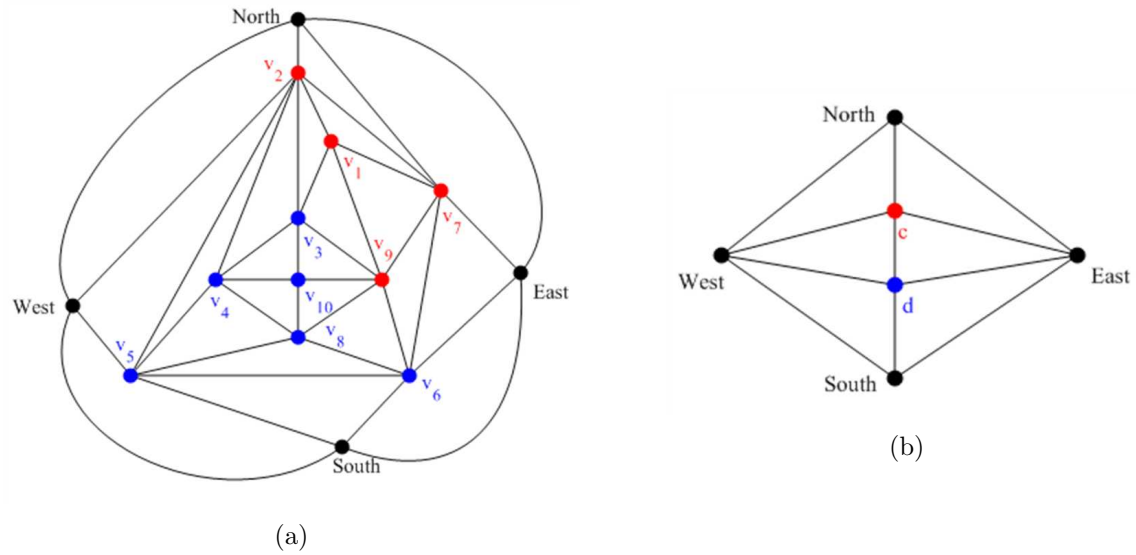


Figure 6.8: (a) A graph $\mathscr{G}_2$ with two bad clusters (red and blue). (b) The quotient graph of $\mathscr{G}_2$. It is a PTP. $\mathscr{G}_2$ is well-clustered although its clusters are bad.

The following theorem states the necessary and sufficient conditions for a CG to be well-clustered.

Theorem 6.3. *A CG $\mathscr{G} = (G, \mathscr{C})$, G being a PTP graph, is a well-clustered graph if and only if each cluster C is good in $Q(C)$.*

Proof. If $\mathscr{G}$ is well-clustered, its quotient graph Q is a PTP. Thus, the contraction of C in $Q(C)$, for every cluster C, never creates a separating triangle. This establishes necessity.

The sufficiency is proved by contradiction. Suppose that Q is not a PTP graph. Since Q is a multi-PTG with no loops and an exterior 4−cycle (Corollary 6.1), either Q contains a separating triangle or a complex 2-cycle. Let $\Delta = (c_1, c_2, c_3)$ be a separating triangle in Q, where c_i is the macro associated to cluster C_i, for $i = 1, 2, 3$. The exterior cycle of $Q(C_1)^{adj}_{C_1}$ does not contain any vertex which is enclosed in Δ. Since at least one of them belongs to $(C_1)^{adj}_{Q(C_1)}$, C_1 is not good, which contradicts the hypothesis. Same contradiction is obtained when supposing that Δ is a complex 2−cycle. □

Theorem 6.4. *If $\mathscr{G}$ admits a c-rectangular dual it is well-clustered.*

Proof. Let $\mathscr{R}(\mathscr{G})$ be a c-rectangular dual of $\mathscr{G}$, including the rectangles corresponding to the four exterior vertices of G. From $\mathscr{R}(\mathscr{G})$, the rectangular dual of Q can be obtained by deleting all rectangles, except for the enclosure rectangles of the rectangular duals of each cluster. The thesis follows from Theorem 3.5. □

6.3 Rectangular dual of a Cluster

Let us assume that $\mathscr{G}$ is a well-clustered graph. After creating the skeleton of a c-

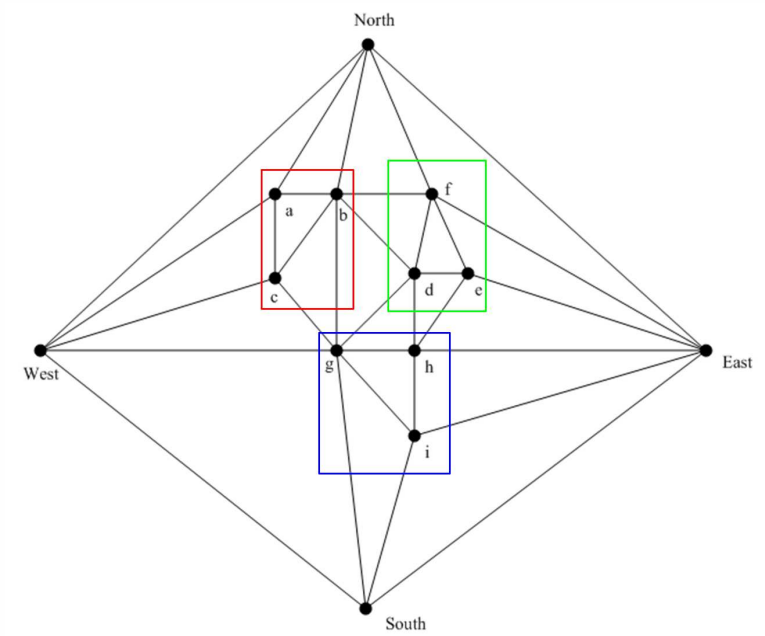


Figure 6.9: A clustered graph with three clusters: $A = \{a, b, c\}$, $B = \{d, e, f\}$, $C = \{g, h, i\}$.

rectangular dual, the rectangular dual of each cluster C must be inserted into the corresponding rectangle in $\mathscr{R}(Q)$. Since a c-rectangular dual of $\mathscr{G}$ is also a rectangular dual of the underlying graph, the rectangular dual of C can not be an arbitrary one. In fact, if $v \in C$ is incident with edge (v, w), $w \in D$, then $f(v)$ must be adjacent to $f(w)$ in $\mathscr{R}(\mathscr{G})$ (Fig. 6.9 and 6.10).

In order to take into account the adjacent clusters, the rectangular dual of C is computed from G_C^{adj}.

Lemma 6.9. *G_C^{adj} is a simple connected PTG with no separating triangles.*

Proof. G_C^{adj} is connected and has an exterior cycle Ext_G^C composed of vertices of C_G^{adj} (Lemma 6.3). Ext_G^C is a complex cycle in G. G_C^{adj} is a PTG only if all vertices enclosed by Ext_G^C belong to either C or C_G^{adj} (Lemma 2.11). Suppose that a vertex $v \notin C \cup C_G^{adj}$ is enclosed in Ext_G^C. v is adjacent to at least one vertex belonging to C_G^{adj} (say w), otherwise G would be disconnected. But v is consecutive in $Adj(w)$ to at least one node $z \in C$. By Lemma 2.8, v is adjacent to z, contradicting the fact that $v \notin C_G^{adj}$. □

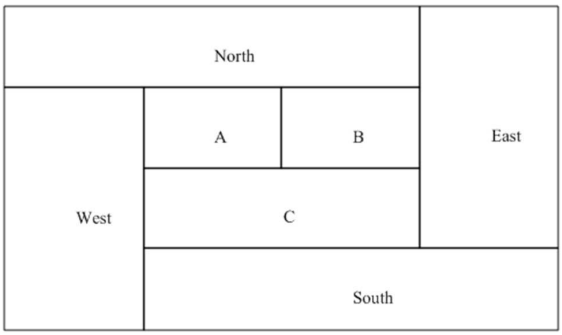


Figure 6.10: Rectangular dual of the quotient graph of the graph in Figure 6.9. Since vertex b (which belongs to cluster A) is adjacent to vertex f (belonging to B), the rectangular dual of the subgraph induced by A must be such that the rectangle corresponding to b is adjacent to the right side of rectangle A. In fact, rectangles A and B share respectively the right and the left edge.

Lemma 6.10. *Let C, D be two clusters of $\mathscr{G}$. The following hold:*

1. *At least one vertex of D is on Ext_G^C.*
2. *If $d_1, ..., d_k$, $d_i \in D$, $1 \leq i \leq k$, are on Ext_G^C, then they induce a path.*
3. *If a vertex $d_1 \in D$ is enclosed in Ext_G^C, there is a path composed of vertices of D joining d with some vertex $d_2 \in D$ on Ext_G^C.*

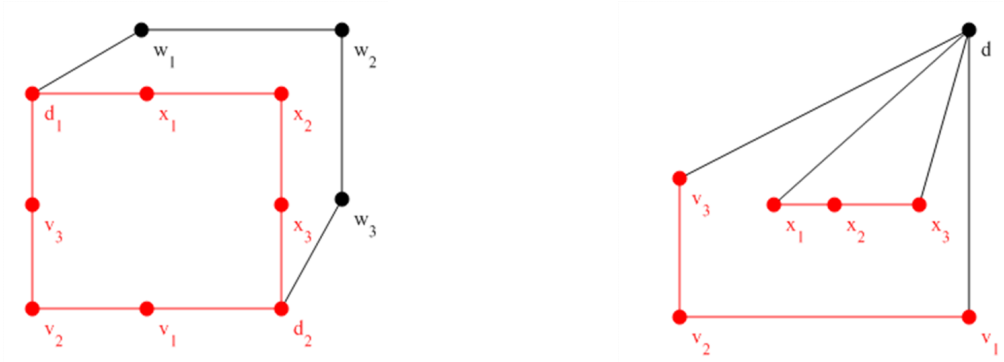
Proof. 1.

2. Since $\mathscr{G}$ is well-clustered, the macro corresponding to D is on the exterior cycle of $Q(C)_C^{adj}$. Consequently, at least one vertex of D is on Ext_G^C.

3. By contradiction. Suppose that there is a path $(d_1, x_1, ..., x_k, d_2)$ on Ext_G^C such that $x_i \notin D$, $1 \leq i \leq k$. Since $\mathscr{G}$ is c-connected, there must be a path Π, composed of vertices of D, joining d_1 and d_2. Π is either enclosed in Ext_G^C or outside. In the first case, Ext_G^C is subdivided by Π into two complex cycles; as a result, C has two connected components, which is not possible in a c-connected graph. In the second case, after the contraction of cluster D, vertices $x_1, ..., x_k$ are enclosed in a cycle composed of vertices adjacent to C (Fig. 6.11). This contradicts the goodness of C.

4. Since all vertices enclosed in Ext_G^C are either vertices of C or vertices of C_G^{adj}, $d_1 \in C_G^{adj}$. By the first property, at least one node d_2 of D belongs to Ext_G^C and a path of vertices of D join d_1 to d_2, as D is connected.

□



(a) (b)

Figure 6.11: (a) The path from $(d_1, w_1, w_2, w_3, d_2)$, such that $w_i \notin D$, lays outside C^{adj} (in red). (b) After the contraction of D, x_1, x_2 and x_3 are enclosed in a cycle composed of vertices adjacent to C.

Lemma 6.11. *$Q(C)^{adj}_C$ can be obtained from G^{adj}_C by contracting each edge linking two vertices belonging to the same cluster adjacent to C.*

Proof. We prove the lemma by construction. Let $d_1, ..., d_k$ be the vertices of D, D being adjacent to C, on Ext^C_G. By Lemma 6.10, these vertices induce a path Π on Ext^C_G. Contracting the edges on Π creates just one vertex (say d) belonging to Ext^C_G. By Lemma 6.10, there is a path between each vertex enclosed in Ext^C_G and d. After contracting the edges of all these paths, Ext^C_G contains only the macro associated to D. The thesis is obtained, by applying these considerations to all clusters adjacent to C. □

Lemma 6.12. *$Q(C)^{adj}_C$ is a PTG with no loops.*

Proof. By Lemma 6.2, $Q(C)$ is a PTG with no loops. Since C is good, all nodes of $Ext^C_{Q(C)}$ belong to $C^{adj}_{Q(C)}$ and all nodes enclosed in $Ext^C_{Q(C)}$ belong to C. Since $Ext^C_{Q(C)}$ is complex chordless cycle in $Q(C)$, the thesis follows from Lemma 2.11. □

Unfortunately, $Q(C)^{adj}_C$ may contain separating triangles and multiple edges (Fig. 6.12 and 6.13). However, a rectangular dual of $Q(C)^{adj}_C$ can not be computed with Kant-He algorithm even if it is simple and has no separating triangles. In fact, the algorithm requires $Q(C)^{adj}_C$ to have an exterior 4−cycle. The vertices on $Ext^C_{Q(C)}$ are the macros of the clusters adjacent to C. Each macro corresponds to a distinct rectangle in $\mathscr{R}(Q)$, with a precise orientation (north, west, south and east) with respect to $f(c)$, c being the macro corresponding to C (Fig. 6.10). The notion of *orientation* applies to rectangles as well as to their corresponding vertices. Thus, if rectangle $f(r)$ is north of $f(s)$, then r is north of

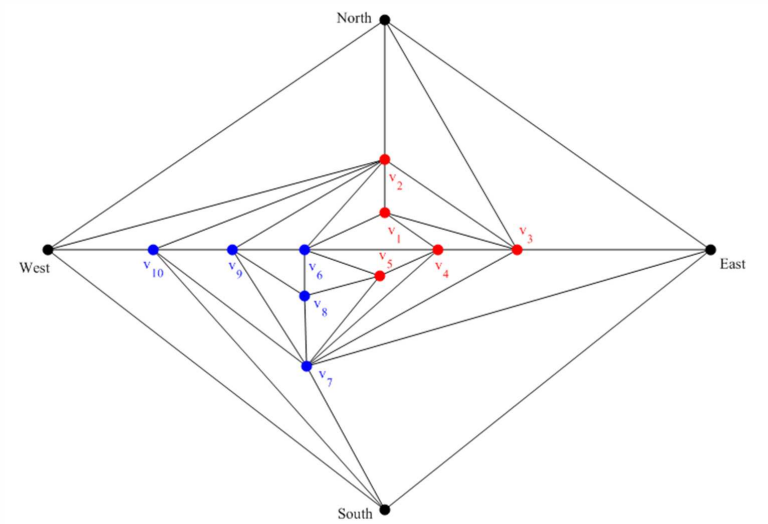


Figure 6.12: A clustered graph with two clusters: A (red vertices) and B (blue vertices).

s. As a result, the vertices in $C^{adj}_{Q(C)}$ can be partitioned into four disjoint sets (denoted as N^C, W^C, S^C, E^C), one for each orientation (Fig. 6.14).

Lemma 6.13. *The exterior cycle of graph $Q(C)^{adj}_C$ is composed in counterclockwise direction of:*

1. *A path $(v_{top,1}, ..., v_{top,t})$, $t \geq 1$, such that $v_{top,i} \in N^C$, $1 \leq i \leq t$;*
2. *An edge $(v_{top,t}, v_{left,1})$;*
3. *A path $(v_{left,1}, ..., v_{left,l})$, $l \geq 1$, such that $v_{left,i} \in W^C$, $1 \leq i \leq l$;*
4. *An edge $(v_{left,l}, v_{bottom,1})$;*
5. *A path $(v_{bottom,1}, ..., v_{bottom,b})$, $b \geq 1$, such that $v_{bottom,i} \in S^C$, $1 \leq i \leq b$.*
6. *An edge $(v_{bottom,b}, v_{right,1})$;*
7. *A path $(v_{right,1}, ..., v_{right,r})$, $r \geq 1$, such that $v_{right,i} \in E^C$, $1 \leq i \leq r$;*
8. *An edge $(v_{right,r}, v_{top,1})$.*

Proof. The exterior cycle of $Q(C)^{adj}_C$ is the same as the one in Q^{adj}_C. In $\mathscr{R}$(Q), the rectangles adjacent to $f(c)$, with the same orientation with respect to $f(c)$, are pairwise adjacent; by duality, the corresponding vertices induce a path. Rectangles $f(v_{top,t})$ and $f(v_{left,1})$

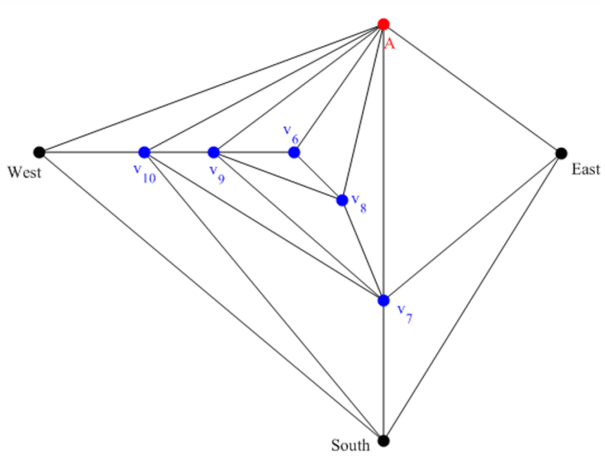


Figure 6.13: Graph $Q(B)_B^{adj}$ of cluster B shown in Fig. 6.12

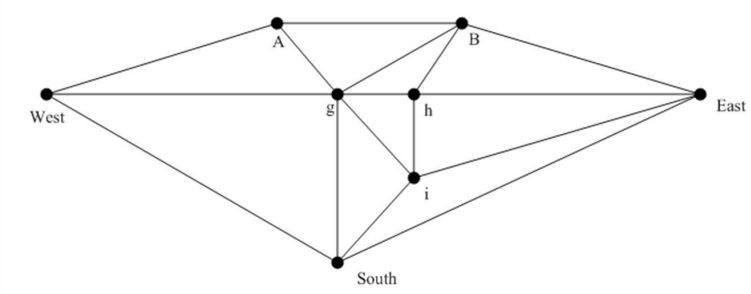


Figure 6.14: The graph $Q(C)_C^{adj}$ of the graph in Figure 6.9. From Fig. 6.10, we get $N^C = \{A, B\}$, $W^C = \{West\}$, $S^C = \{South\}$ and $E^C = \{East\}$.

are adjacent too, as otherwise there would be four rectangles meeting at a point. Thus, $v_{top,t}$ and $v_{left,1}$ are adjacent. The same holds for $(v_{left,l}, v_{bottom,1})$, $(v_{bottom,b}, v_{right,1})$ and $(v_{right,r}, v_{top,1})$. The proof is complete by remarking that in counterclockwise direction the northern rectangles precedes the western rectangles which precede the southern rectangles which in turn precede the eastern ones. □

By Lemma 6.13, contracting the vertices of each set N^C, W^C, S^C and E^C results in a graph with an exterior 4−cycle. The exterior vertices are denoted as $North^C$, $West^C$, $South^C$ and $East^C$. We call this new graph the *rectangle graph* of C and we denote it as Q_4^C (Fig. 6.15).

Lemma 6.14. *Q_4^C is multi-PTG with no loops, with an exterior chordless 4−cycle composed of simple edges.*

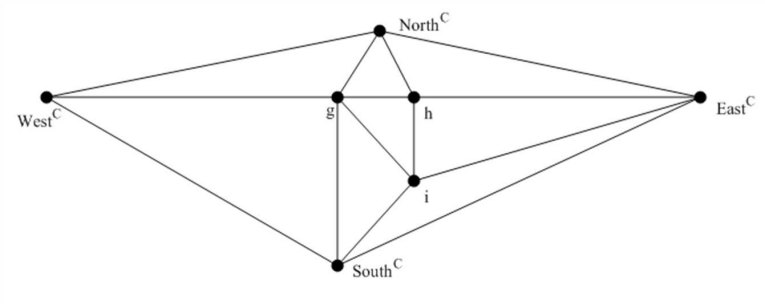


Figure 6.15: The graph Q_4^C associated to cluster C of graph G shown in Fig. 6.9. There is a one-to-one correspondence between the intracluster edges of C in G and the edges of Q_4^C not incident with exterior vertices. The inner edges incident with the exterior vertices may correspond to more than one intercluster edge of C in G. Edge $(g, North^C)$, for instance, corresponds to (g, c), (g, b) and (g, d). In fact, while contracting edges (c, b) and (b, d), these edges are merged to $(g, North^C)$.

Proof. Since C is good, the exterior cycle of $Q(C)_C^{adj}$ only contains simple edges. By Lemma 6.12, $Q(C)_C^{adj}$ is a multi-PTG with no loops. Since Q_4^C is obtained from $Q(C)_C^{adj}$ by only contracting the exterior edges, Q_4^C is a multi-PTG with no loops (Lemma 2.9). The exterior cycle of Q_4^C has length four and is chordless by construction. Finally, all exterior edges are simple, as a multiedge would be created only contracting a non-exterior edge of $Q(C)_C^{adj}$. □

Q_4^C may contain separating triangles and/or multiedges. We will describe in Section 6.4.3 how to cope with this situation. Now we assume that Q_4^C is a simple graph with no separating triangles (thus a PTP, by Lemma 6.14). Using the Kant-He algorithm, we create a rectangular dual of Q_4^C, which is a rectangular dual of cluster C such that each rectangle has the right position with respect to the rectangles of the adjacent clusters. However, this is not enough yet. In fact, also the size of each rectangle must be properly set up, to guarantee that its adjacencies are correct (Fig. 6.16). Manually adjusting to right the wrong adjacencies does not sound a feasible solution, especially when the graph is large. A better solutions consists of computing a REL of Q_4^C (without creating the rectangular dual) for each cluster C and merging these RELs to obtain a REL for the underlying graph.

6.3.1 A REL for the Underlying Graph

The edges of Q_4^C are related to the edges of the underlying graph G (Fig. 6.15). There is a one-to-one correspondence between the intracluster edges of C in G and the edges of Q_4^C not incident with the exterior vertices. However, the inner edges incident with the

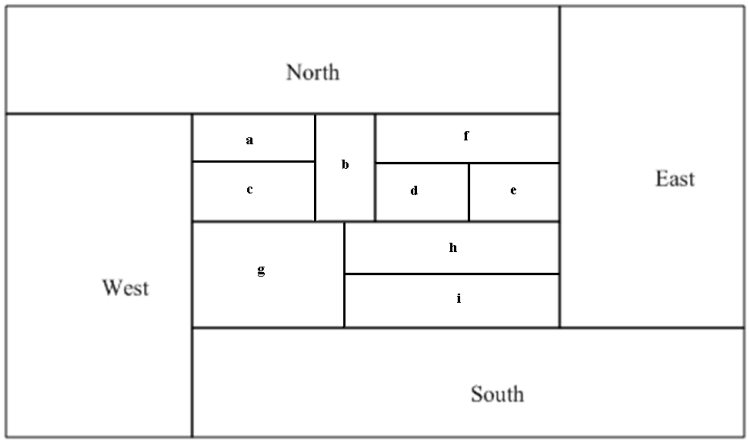


Figure 6.16: A wrong c-rectangular dual of the graph shown in Fig. 6.9. This is created filling each rectangle of the skeleton of Fig. 6.10. Rectangle g should be adjacent to rectangle d, but it is too small.

exterior vertices may correspond to more than one intercluster edge of C in G. This is due to the fact that, while contracting the exterior edges of C_G^{adj}, these edges may be merged together. We define the following function:

$$corr_C : E(Q_4^C) \to \wp(E(G))$$

which associates each edge e of Q_4^C to the edges of G which are merged to e while creating Q_4^C. After creating a REL of Q_4^C, the label of each inner edge e is assigned to the edges in $corr_C(e)$. Applying this for each cluster C, we create a REL ρ for G. We now explain why.

As said before, each inner edge e of Q_4^C connecting two inner vertices corresponds to only one intracluster edge of C in G. Moreover, as clusters do not overlap, e receives a label only once, when the REL of Q_4^C is created. Thus, no ambiguity arises when assigning the label of e to edge $corr_C(e)$. When it comes to the intercluster edges of C the situation is worse. In fact, if e is an inner edge of Q_4^C incident with an exterior vertex, $corr_C(e)$ may contain more than one edge. What does guarantee that assigning the label of e to all edges in $corr_C(e)$ is the right choice? Furthermore, each intercluster edge of C in G will receive two labels, as it connects exactly two clusters. Which one is the correct label? The following results answer both questions.

Lemma 6.15. *Let e be an edge of Q_4^C incident with an exterior vertex. The edges in $corr_C(e)$ are incident with clusters having the same orientation with respect to C.*

Proof. Q_4^C is obtained from G_C^{adj} by contracting edges linking two vertices with the same

orientation with respect to C. By Lemma 6.9 G_C^{adj} is a simple connected PTG with no separating triangles. Let $e = (u, v)$ be an exterior edge of G_C^{adj}, such that u and v have the same orientation with respect to C. The contraction of e merges e_1 and e_2 if and only if (e, e_1, e_2) is a face, that is e_1 is incident with u and e_2 is incident with v. □

Lemma 6.16. *Let C and D be two clusters and let $e = (c, d)$, $c \in C$ and $d \in D$ be an intercluster edge of C in G. e receives the same label from the REL of Q_4^C and Q_4^D.*

Proof. Without loss of generality we assume that D is north of C. While creating Q_4^C, e is merged into edge $(c, North^C)$, which is assigned label $(T_1, \rightarrow)$ (Lemma 3.5); similarly, while creating Q_4^D, e is merged into $(d, South^D)$, which is given label $(T_1, \leftarrow)$. Thus, in both cases, (c, d) belongs to T_1 and is directed from c to d. □

Lemma 6.17. *ρ is a REL on G.*

Proof. By construction, the edges incident with the four exterior vertices of G are properly labelled, according to Definition 3.2. Let v be an inner vertex of G belonging to a cluster C. In Q_4^C, v is an inner vertex and the edges incident with v are labelled as stated in Definition 3.2. If, for each edge e incident with v, $corr_C(e)$ only contains one edge, the lemma is trivially true. Let e_1 be such that $|corr_C(e_1)| > 1$. By the properties of edge contraction in PTGs, the edges in $corr_C(e_1)$ are pairwise consecutive in the incidence list of v in G. Thus, the incidence list of v in G is exactly that in Q_4^C with the exception that e_1 is replaced by the edges in $corr_C(e_1)$. Since they all receive the same label as e (as they have the same orientation with respect to C, Lemma 6.15), the edges incident with v are properly labelled in ρ. □

The following theorem states that ρ is not a generic REL on G.

Theorem 6.5. *The rectangular dual of G computed from ρ is a c-rectangular dual of $\mathscr{G}$.*

Proof. Let C be a cluster of $\mathscr{G}$ and (T_1, T_2) be a REL on it. The edges incident with the four exterior vertices in Q_4^C appear in counterclockwise order as a set of edges of T_1 directed towards $North^C$, a set of edges of T_2 coming from $West^C$, a set of edges of T_1 coming from $South^C$ and a set of edges of T_2 directed towards $East^C$. Each such edge $e = (v, w)$, w being one of the exterior vertices, is replaced in G by edges (v, w_1), ..., (v, w_k), $k \geq 1$, having the same label as e. Consequently, ρ induces a REL on $G[C]$. The thesis follows from Lemma 6.2. □

The necessary and sufficient conditions for c-rectangular dualization immediately follow:

Theorem 6.6. *A c-connected c-planar CG $\mathscr{G} = (G, \mathscr{C})$, G being a PTP, admits a c-rectangular dual if and only if $\mathscr{G}$ is well-clustered and, for every cluster $C \in \mathscr{C}$, Q_4^C is a PTP graph.*

Proof. From Theorem 6.4 and Theorem 6.5. □

6.4 The algorithm

The input to the algorithm is a c-planar c-connected clustered graph $\mathscr{G} = (G, \mathscr{C})$, with G being a simple connected graph. The following steps are executed:

1. The algorithm described in Section 3.8 is used to turn G into a PTP graph. If a crossover vertex c is added on edge (v, w), c is indifferently assigned to the cluster v or w belongs to.
2. The quotient graph Q of $\mathscr{G}$ is created using the procedure described in Section 2.8.1.
3. Q is turned into a PTP graph, if not yet. $\mathscr{G}$ is changed accordingly to make it a well-clustered graph (Section 6.4.1). Then, a REL on Q is created.
4. For each cluster $C \in \mathscr{C}$
   (a) The orientation of the neighbours of C with respect to C is obtained from the REL on Q.
   (b) The rectangle graph Q_4^C is created and turned into a PTP graph. Then, a REL on Q_4^C is computed.
5. REL ρ is computed, as described in Section 6.3.1.
6. The Kant-He algorithm is used to create a rectangular dual of G from ρ.

6.4.1 Enforcing Good Clustering

If $\mathscr{G}$ is badly clustered, its quotient graph Q is not a PTP. By Lemma 6.1, this implies that Q contains either separating triangles or multiple edges (or both). Thus, testing whether $\mathscr{G}$ is well-clustered reduces to testing the existence of separating triangles or multiple edges in G ($O(n)$ time).

To get rid of multiple edges, we use the *break edge* approach, as done for the separating triangles. However, a crossover vertex must be added on each occurrence of a multiple edge, as otherwise a separating triangle would be created (Fig. 6.17). That is why multiple edges are so undesirable and why the absence of loops is such a good news. Thus, unlike separating triangles, breaking 2-separating cycles is not an optimization problem. For this reason, it is better to first eliminate the multiple edges and then the separating triangles

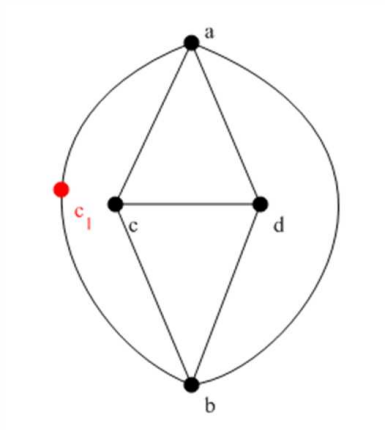


Figure 6.17: Adding a crossover vertex c_1 turns the 2-separating cycle (a, b) into a separating triangle (a, b, c_1).

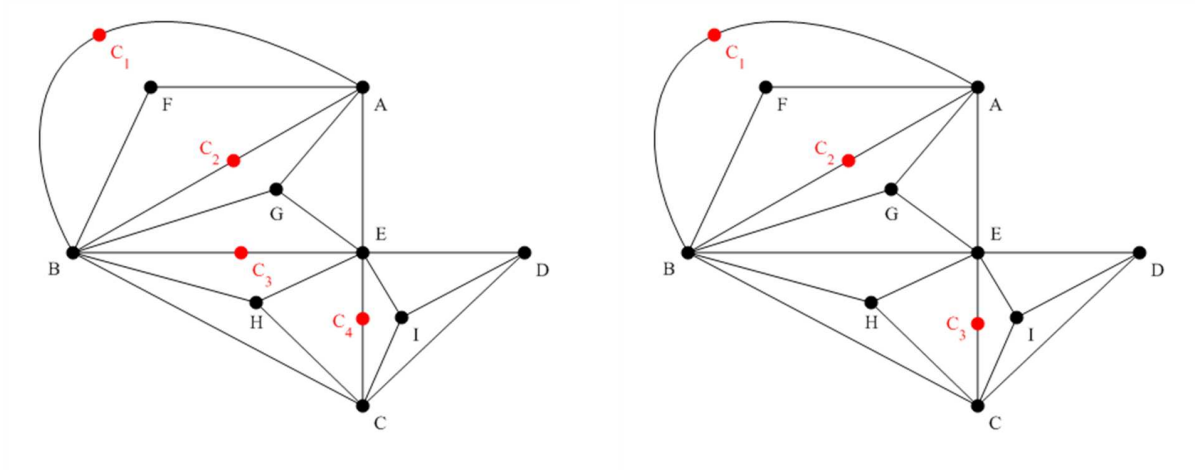
(Fig. 6.18); adding a crossover vertex on an occurrence of a multiedge also deletes the separating triangles incident with it. Each new crossover vertex requires a change in the underlying graph. As done in Section 6.3.1, we define a function $corr : E(Q) \to \wp(E(G))$, such that $corr(e)$ is the set of edges which are merged to e while creating Q from G. Let C_1 be a crossover vertex added on an inner edge (C, D) of Q (Fig. 6.19). Since Q is a PTG, e is incident with two faces (A, C, D) and (B, C, D); when adding C_1 on (C, D), C_1 is linked to A and B so that the inner faces are still 3−faces.

Conceptually, C_1 is the macro-vertex of a new cluster C_1 in $\mathscr{G}$, whose vertices are adjacent to vertices of clusters A, B, C and D. Since $\Sigma = (A, B, C, D)$ is a cycle in Q, there is one cycle Π in G with the characteristics stated by the following theorem.

Lemma 6.18. *There is one cycle* Π *in* G *such that:*

1. Π *encloses all the edges in* $corr((C, D))$.
2. Π *does not enclose any vertex of* A *and* B.
3. *Its vertices appear in counterclockwise order as follows: one or more vertices of* C, *one vertex* a *of* A, *one or more vertices of* D *and one vertex* b *of* B.
4. *There is exactly one face* (v, w, a), *enclosed by* Π, *such that* $v \in C$ *and* $w \in D$.
5. *There is exactly one face* (x, y, b), *enclosed by* Π, *such that* $x \in C$ *and* $y \in D$.

Proof. G can be obtained from Q by applying a sequence of vertex splits. The order in which the vertices are split does not matter, as well as the order in which edges are



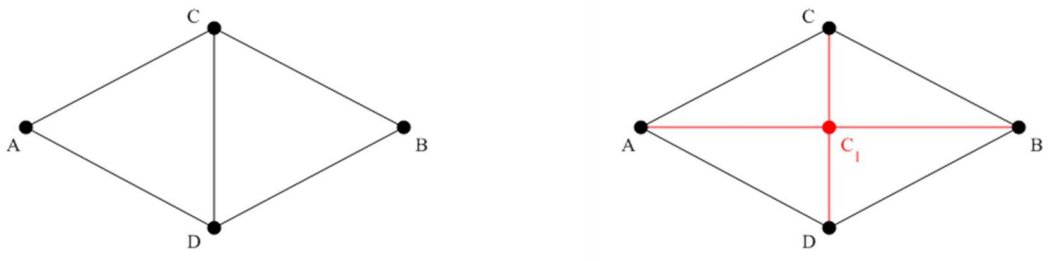
(a) (b)

Figure 6.18: (a) Breaking separating triangles before eliminating multiedges results in four crossover vertices. (b) Breaking separating triangles after eliminating multiedges results in three crossover vertices.

contracted. Therefore, we assume that all vertices of A and B are split before those of C and D. We prove the lemma by induction on the number of vertex splits. Henceforth, Π denotes a cycle complying with the conditions of this lemma.

If A contains only one node, the lemma is trivially true, as $\Pi = \Sigma$. Let z be the $k-$th vertex to split. If $z \neq a$, Π does not change. If $z = a$, then a new vertex a' is created and three cases are possible (Fig. 6.20): a' is enclosed in Π, a' is outside Π and a' is on Π. Π becomes (a', D, B, C) in the first case, (a, D, B, C) in the second and third case. Similarly, we prove the validity of the lemma after splitting the vertices of B.

Let c be the $k-$th vertex to split in C, $k > 0$ and $c \neq v, x$. If c is outside Π or enclosed in



(a) (b)

Figure 6.19: (a) An interior edge of Q ((C, D)) with the two incident faces. (b) Operations performed upon addition of vertex C_1 on (C, D).

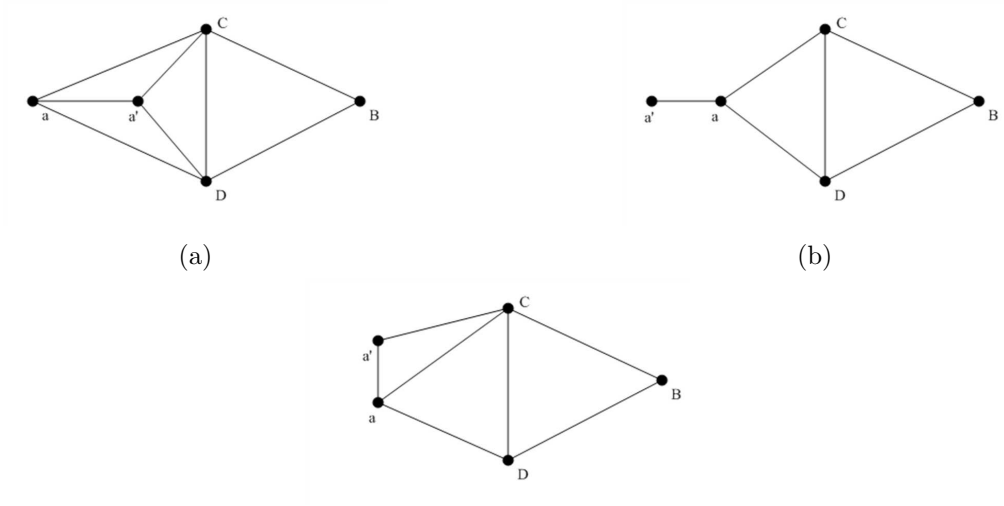


Figure 6.20: (a) The new vertex a' falls inside $\Pi = (a, D, B, C)$. (b) The new vertex a' is outside Π. (c) The new vertex a' is on Π.

Π, Π does not change. If c is on Π, Π only changes if c' is also on Π (and in this case, the conditions of the lemma still hold). If $c = v$ the following cases are possible (Fig. 6.21).

□

First, c' is not adjacent to a or precedes c in clockwise direction in $Inc(a)$; Π does not change. Second, c' follows c in clockwise direction in $Inc(a)$; Π does not change and (a, w, c') is the only face incident with a, complying with condition 4. Same considerations apply if $c = x$. Finally, the validity of the lemma is proved similarly when splitting the vertices of D.

Lemma 6.19. *If* Π *is a separating cycle, the vertices enclosed in* Π *belong to either* C *or* D*.*

Proof. By Lemma 6.18 Π does not enclose any vertex of A and B. If Π enclosed a vertex of a cluster E, $E \neq C$ and $E \neq D$, Σ would be a separating cycle too. □

Since vertex C_1 is added on edge (C, D) in Q, a crossover vertex $c_{1,i}$ is added on each edge $e_i \in E(G)$, such that $e_i \in corr(e)$. When adding $c_{1,i}$ on $e_i = (v, w)$, e_i is deleted and substituted with $(v, c_{1,i})$ and $(c_{1,i}, w)$ and two new edges are added, $(c_{1,i}, x)$ and $(c_{1,i}, y)$, where (v, w, x) and (v, w, y) are the faces incident with e_i. Vertices $c_{1,i}$ form a new cluster C_1 in G. As $\mathscr{G}$ is c-connected, we need that C_1 is connected too.

Lemma 6.20. C_1 *is connected.*

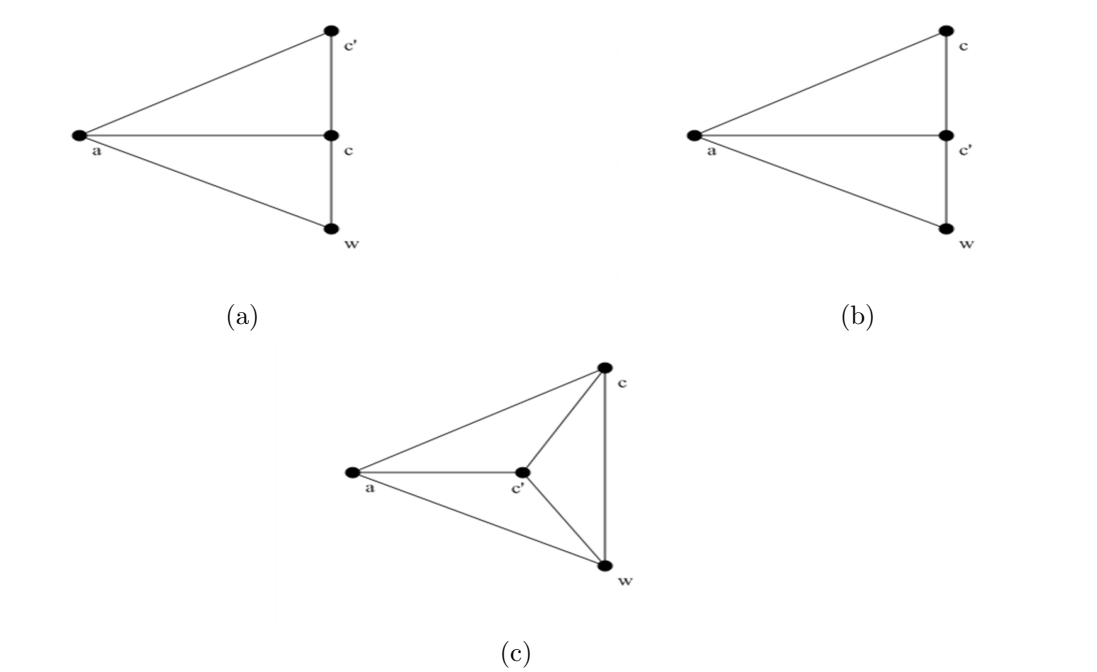


Figure 6.21: (a) c' precedes c in the adjacency list of a. (b) c' follows c in the adjacency list of a. (c) c' follows c in the adjacency list of a and is enclosed in the separating triangle (a, c, w).

Proof. Suppose not. There is at least one edge $e = (v, w) \in corr((C, D))$, $v \in C$ and $w \in D$ which is incident with two faces (v, w, x) and (v, w, y) such that $x \notin \{A, B, C, D\}$ or $y \notin \{A, B, C, D\}$. Since e is enclosed in Π (Lemma 6.18), this implies that x and y are enclosed in Π. This contradicts Lemma 6.19. □

The macro of Q corresponding to C_1 is enclosed in Σ, as vertices $c_{1,i}$ are added only on edges enclosed in Π. Moreover, by Lemma 6.19 C_1 is adjacent to only A, B, C and D. This means that adding $c_{1,i}$ actually corresponds to adding a crossover vertex on (C, D) in Q.

Algorithm 6.1 reports the pseudocode of the procedure EnforceGoodCluster and Fig. 6.22 shows an example of a CG is turned into a well-clustered graph.

Lemma 6.21. *Procedure* EnforceGoodCluster *adds at most $O(n)$ crossover to G.*

Proof. In fact, each edge receives at most one crossover vertex. □

Theorem 6.7. *Procedure* EnforceGoodCluster *runs in $O(n)$ time.*

Proof. Lines 2 to 7 require $O(n)$ time. If fact, each separating triangle is incident with at most three edges. Procedure BreakST breaks all separating triangles with one of the

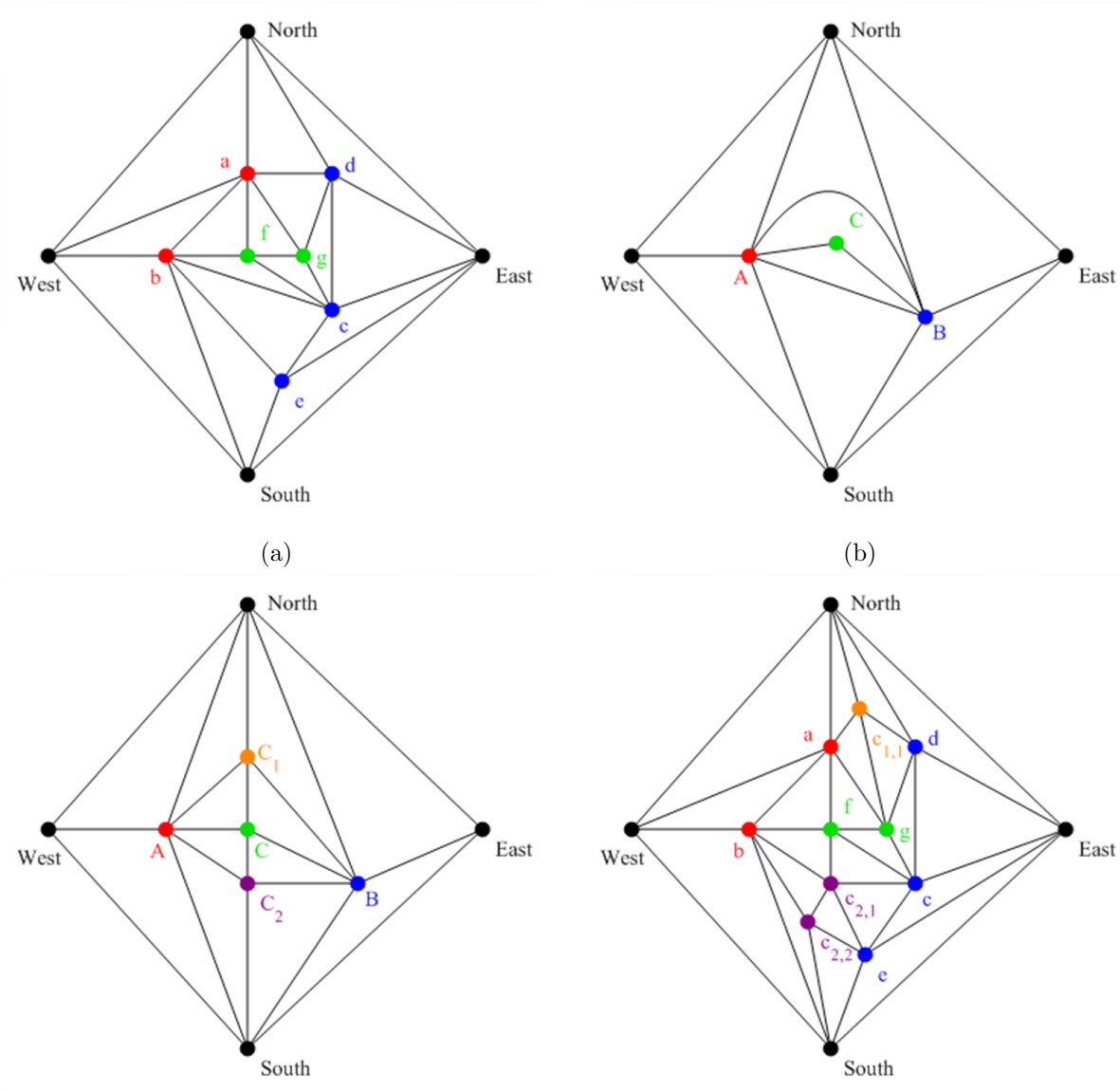


Figure 6.22: (a) A clustered graph G with three clusters $A = \{a, b\}$, $B = \{c, d, e\}$ and $C = \{f, g\}$. (b) The quotient graph Q, containing a separating triangle $(A, B, North)$ and a separating 2-cycle $((A, B))$. (c) Two crossover vertices C_1 and C_2 are required to turn Q into a PTP graph. (d) Since the contraction list of edge (A, B) is composed of edges (a, d), (b, e) and (b, c), three crossover vertices $c_{1,1}$, $c_{2,1}$ and $c_{2,2}$ are added to G.

Algorithm 6.1 Enforcing Good Clustering

```
1: procedure EnforceGoodCluster(Q: Graph, ST: array of separating triangles in
   Q, M: array of multiedges in Q)
2:     for all e ∈ M do
3:         AddCrossoverVertex(e)
4:         delete(e, M)
5:         for all Δ ∈ ST incident with e do
6:             mark Δ as deleted
7:         end for
8:     end for
9:     BreakST()
10:    for all broken edge e ∈ E(Q) do
11:        for all e_1 ∈ corr(e) do
12:            AddCrossoverVertex(e_1)
13:        end for
14:    end for
15: end procedure
```

heuristic algorithms studied in Chapter 4. Finally, a crossover vertex is added on each edge of G corresponding to the edges broken in Q (Lines 10 to Lines 14); this costs $O(m)$ time. □

6.4.2 Orientation of a Cluster

After breaking all multiedges and separating triangles in Q, a REL (T_1, T_2) is created on Q. We recall the following properties from Chapter 3:

- If $(C, D) \in (T_1, \rightarrow)$ then D is north of C.
- If $(C, D) \in (T_2, \leftarrow)$ then D is west of C.
- If $(C, D) \in (T_1, \leftarrow)$ then D is south of C.
- If $(C, D) \in (T_2, \rightarrow)$ then D is east of C.

For each cluster C, we introduce an array $Orientation_C$ such that $Orientation_C[D]$, D being a cluster adjacent to C, assumes value N, W, S and E if D is north, west, south or east of C respectively. Filling array $Orientation_C$ only requires the visit of $Adj(c)$, c being the macro associated to C

Algorithm 6.2 Determining Cluster Orientation

```
1: procedure EnforceGoodCluster(Q: Graph, C: cluster)
2:     for all e = (C, D) ∈ Inc(C) do
3:         switch(e.label)
4:         case: (T_1, →) Orientation[D] = N
5:         case: (T_2, ←) Orientation[D] = W
6:         case: (T_1, ←) Orientation[D] = S
7:         case: (T_2, →) Orientation[D] = E
8:     end for
9: end procedure
```

6.4.3 Creating the Rectangle Graph of a Cluster

In Section 6.3 we introduced the rectangle graph associated to a cluster C as the graph Q_4^C obtained from G_C^{adj} by contracting all exterior edges $e = (v, w)$ such that v and w have the same orientation with respect to C. This approach requires $O(|C| + |C_G^{adj}|)$ time; since a vertex of G may belong to the adjacency set of as may clusters as its degree, computing the rectangle graph of every cluster takes more than linear time. In this section, we describe an algorithm for creating Q_4^C in $O(|C|)$ time.

First, four nodes $North^C$, $West^C$, $South^C$ and $East^C$ are added to Q_4^C as well as edges $(North^C, West^C)$, $(West^C, South^C)$, $(South^C, East^C)$ and $(East^C, North^C)$ (Fig. 6.23).

Each node $v \in C$ is associated a node v_{new} in Q_4^C. $Adj(v_{new})$ is then obtained from $Adj(v)$ in the following way. Let w be a neighbour of v in G. Since each intracluster edge of C in G corresponds to exactly one edge in Q_4^C, if $w \in C$, then w_{new} is added to $Adj(v)$. If $w \notin C$, edge $(v, w) \in E(G)$ corresponds to an inner edge of Q_4^C incident with one of the four exterior vertices. Without losing generality, we assume that w is north of C; $North^C$ is added to $Adj(v_{new})$ if the last vertex added to $Adj(v_{new})$ is $North^C$. If so, in fact, the vertex x preceding w in $Adj(v)$ is also north of C. $Inc(v_{new})$ is obtained from $Inc(v)$ similarly.

Finally, the adjacency and incidence lists of the four exterior vertices are created. To this extent, we first find the corner vertices $v_{top,right}$, $v_{top,left}$, $v_{bottom,left}$ and $v_{bottom,right}$ (Fig. 6.23). $v_{top,right}$ is the vertex such that $North^C$ precedes $East^C$ in its adjacency list; similarly, $West^C$ precedes $North^C$ in $Adj(v_{top,left})$, $West^C$ precedes $South^C$ in $Adj(v_{bottom,left})$ and $South^C$ precedes $East^C$ in $Adj(v_{bottom,right})$. To create $Adj(North^C)$, $v_{top,right}$ is added as the second element, immediately after $East^C$. Then, the vertices $v_{top,1}$, v_{top_2}..., $v_{top,left}$ are added one after the other. $v_{top,i}$ is retrieved in constant time as the item preceding $North^C$ in $Adj(v_{top,i-1})$. The adjacency and incidence lists of the other exterior nodes are computed in the same way. Fig. 6.24 shows an example. By Lemma 6.14, if Q_4^C is not

Algorithm 6.3 Create Q_4^C

```
1: procedure CreateRD(𝒢: Graph, C: cluster, orientation_C: array)
2:     Create nodes North^C, West^C, South^C and East^C
3:     Adj(North^C) ← [West^C, East^C]; Adj(West^C) ← [South^C, North^C];
4:     Adj(South^C) ← [East^C, West^C]; Adj(East^C) ← [North^C, South^C];
5:     Create edges e_1 = (North^C, West^C), e_2 = (West^C, South^C), e_3 = (South^C, East^C),
       e_4 = (East^C, North^C)
6:     Inc(North^C) ← [(e_1, e_4)]; Inc(West^C) ← [(e_2, e_1)];
7:     Inc(South^C) ← [(e_3, e_2)]; Inc(East^C) ← [(e_1, e_3)];
8:     for all v ∈ C do
9:         InsertNode(Q_4^C, v_new, 0)
10:    end for
11:    for all v_New do
12:        CreateNode(Q_4^C, v_new, v)
13:    end for
14:    find the corner vertices v_top,right, v_top,left, v_bottom,left, v_bottom,right
15:    Create adjacency and incidence lists of the four exterior vertices
16:    Eliminate multiedges and separating triangles
17: end procedure
```

Algorithm 6.4 Create the node v_{new} in Q_4^C corresponding to v in G

```
1: procedure CreateNode(Q_4^C: graph, v_new: node, v: node)
2:
3:     for all w ∈ Adj(v) do
4:         x ← Prev(w, Adj(v))
5:         if w ∈ C then
6:             BackInsert(w_new, Adj(v_new))
7:         else if w ∉ C and orientation[w] ≠ orientation[x] then
8:             z ← one of the exterior nodes of Q_4^C according to orientation[w]
9:             BackInsert(z, Adj(v_new))
10:        end if
11:    end for
12:    create Inc(v_new) similarly
13: end procedure
```

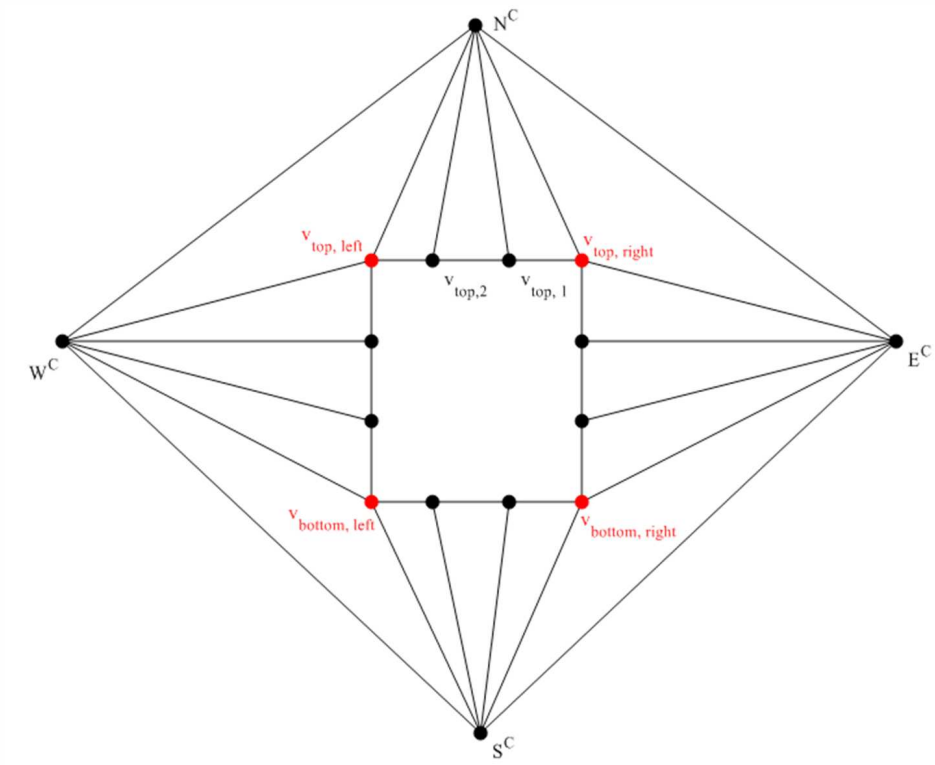


Figure 6.23: Adjacent vertices of the four exterior vertices. The adjacency lists of the four exterior vertices are initialized as follows: $Adj(North^C) = [West^C; East^C]$, $Adj(West^C) = [South^C; North^C]$, $Adj(South^C) = [East^C; West^C]$ and $Adj(East^C) = [North^C; South^C]$.

a PTP, then either it contains separating triangles or multiedges. To break them, an approach similar to that described in Section 6.4.1 is used. A first issue is that no crossover vertex can be added on an exterior edge, as it would increase its length. While Lemma 6.14 that all exterior edges are simple, nothing prevents them to be incident with a separating triangle. Thus, we assume that crossover vertices are added only on inner edges of Q_4^C. Let c be a crossover vertex to add on edge e. Since e is an inner edge, c is an inner vertex and can be considered as a vertex of cluster C. Since c is adjacent to at least one vertex of C, C remains connected.

Any crossover vertex added to Q_4^C must be also added to G. If $|corr_C(e)| = 1$, a crossover vertex is added on the only edge in $corr_C(e)$. Suppose that $corr_C(e) = \{e_1, ..., e_k\}$, $k \geq 2$, implying that e is incident with an exterior vertex. Without losing generality, let $e = (v, North^C)$. We consider separately the cases that e is incident with a separating triangle or is a multiedge. In the first case (Fig. 6.25), $e_1, ..., e_k$ are pairwise consecutive in $Inc(v)$, as already pointed out. Since c breaks the adjacency between v and $North^C$, in G a vertex

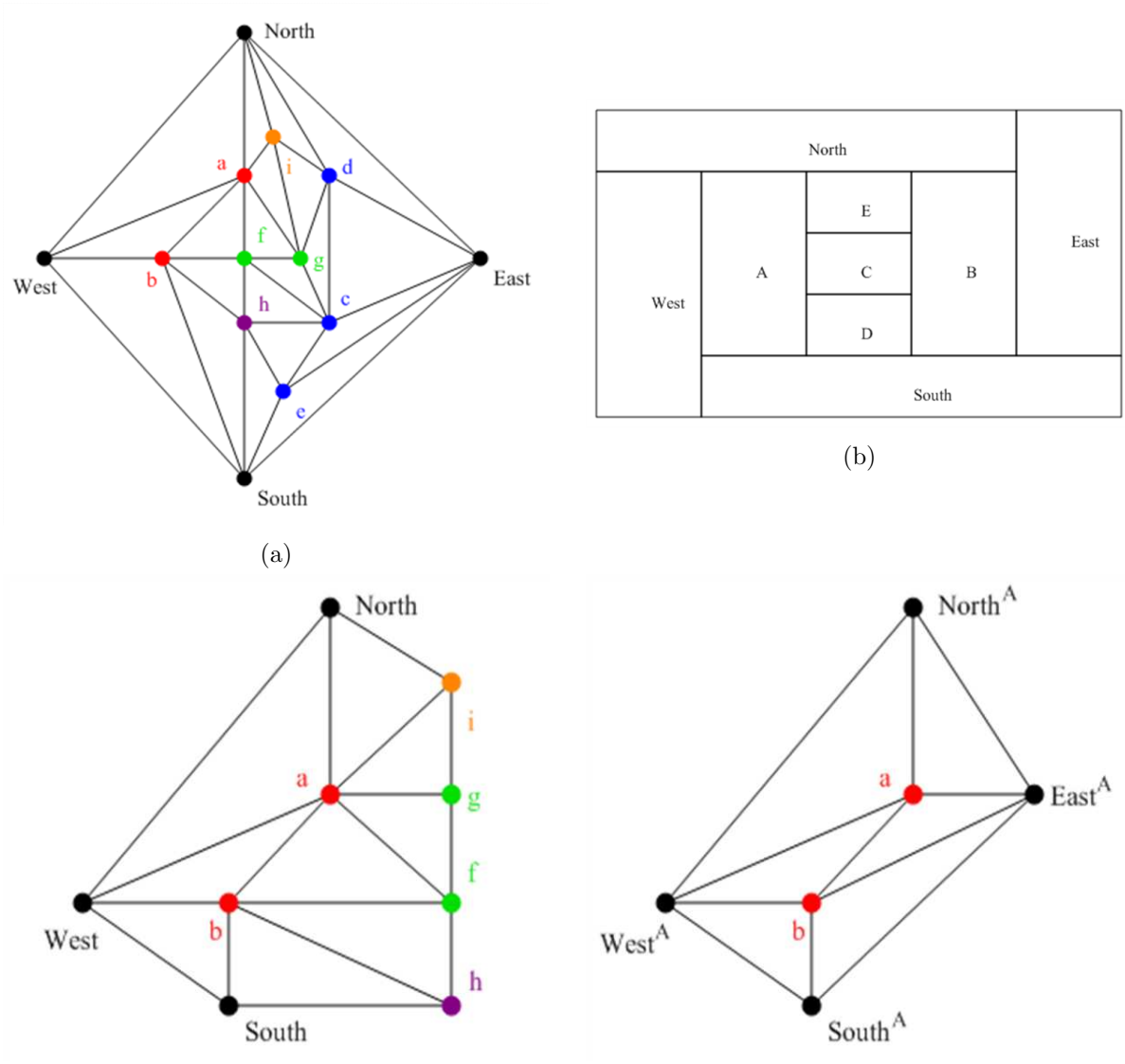


Figure 6.24: (a) A well-clustered graph $\mathscr{G} = (G, \mathscr{C})$, with five clusters: $A = \{a, b\}$, $B = \{c, d, e\}$, $C = \{f, g\}$, $D = \{h\}$ and $E = \{i\}$. (b) A rectangular dual of the quotient graph Q of $\mathscr{G}$. (c) G_A^{adj}. The adjacency list of node a is $[North, West, b, f, g, i]$. (d) Q_4^A. Since f, g and i are east of A, they are replaced by vertex $East^A$ in the new adjacency list.

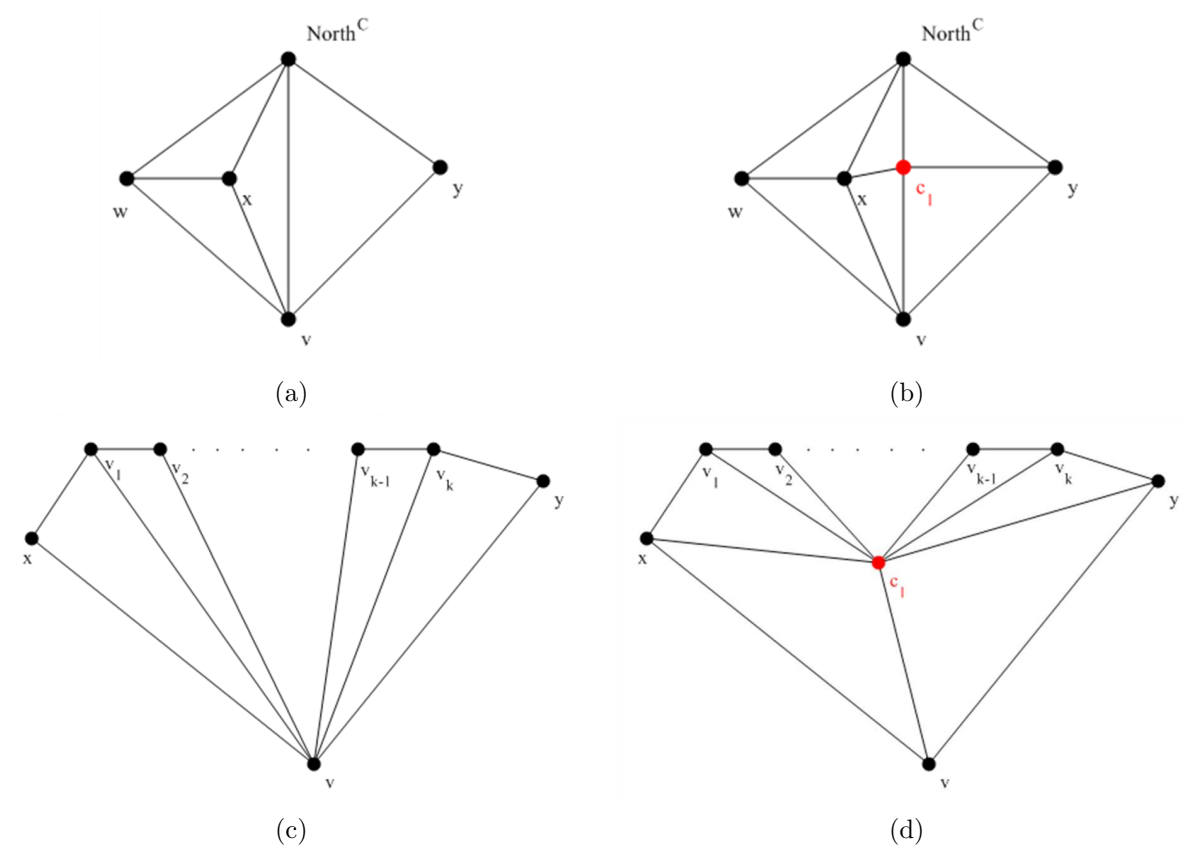


Figure 6.25: (a) $(North^C, v)$ is an edge to break in Q_4^C (b) Crossover vertex c_1 is added on $(North^C, v)$. (c) Situation in graph G: edges $(v, v_1), ..., (v, v_k)$ are in the contraction list of $(North^C, v)$. (d) c_1 is added in G.

has to be added to break the adjacency between v and $v_1, ..., v_k$. Let $G = G + \{c_1\}$ be the graph obtained by adding c_1 as done in Fig. 6.25.

Lemma 6.22. *If G is a PTP, $G + \{c_1\}$ is a PTP.*

Proof. Let $(North^C, v)$ be incident with a separating triangle $\Delta = (North^C, v, w)$. Since $(North^C, v)$ is an inner edge, it is incident with two 3−faces $(North^C, v, x)$ and $(North^C, v, y)$. Let x be enclosed in Δ; consequently, x is an inner vertex of Q_4^C and $x \in C$. y can be an inner or an exterior node, but we do not lose generality if we assume it as an inner vertex. When deleting edges $e_1, ..., e_k$, a new face f is created bounded by a cycle Π which contains v, x, y and all vertices v_i, $1 \leq i \leq k > 1$. Since Π has more than three edges, adding c_1 inside it does not create any separating triangle. f is then triangulated linking c_1 with all vertices of Π. This introduces a separating triangle if and only if Π ha a chord. However, x and y are not adjacent in Q_4^C (as x is enclosed in Δ and y is not) and, consequently, no edge connects x and y in G. If v_i and v_j, $1 \leq i < j - 1 < k$, were adjacent in G, then (v, v_i, v_j) would be a separating triangle, which is a contradiction. □

Lemma 6.23. *$G + \{c\}$ is a well-clustered graph.*

Proof. c is an inner vertex of Q_4^C; as a result, $Ext^C_{Q(C)}$ is not altered and C is still good (Lemma 6.8). The goodness of each cluster D adjacent to C is evaluated in $Q(D)$, where all vertices of C are contracted to a single macro. After contracting edge (v, c) (Fig. 6.25), graph G is as before the introduction of c, thus well clustered. □

When adding a crossover vertex on each occurrence of a multiedge, a similar procedure is applied (Fig. 6.26).

6.5 C-Rectangular Dual of a Hierarchically Clustered Graph

The schema of the algorithm described in Section 6.4 adapts to HCGs with only minor changes, concerning the computation of the quotient graph and the creation of the rectangle graph Q_4^C for a cluster C (steps 2 and 4(a)).

Let $\mathscr{G} = (G, T)$ be a c-connected c-planar HCG (G is assumed to be a PTP graph), ν be a cluster (including the root cluster) and $\mu_1, ..., \mu_j$ the children clusters of ν in T. We define graph $\mathscr{G}(\nu)$ as the CG $(G(\nu), M)$ such that (Fig. 6.27):

1. $G(\nu)$ is the subgraph of G induced by the leaves of T in the subtree rooted at ν.

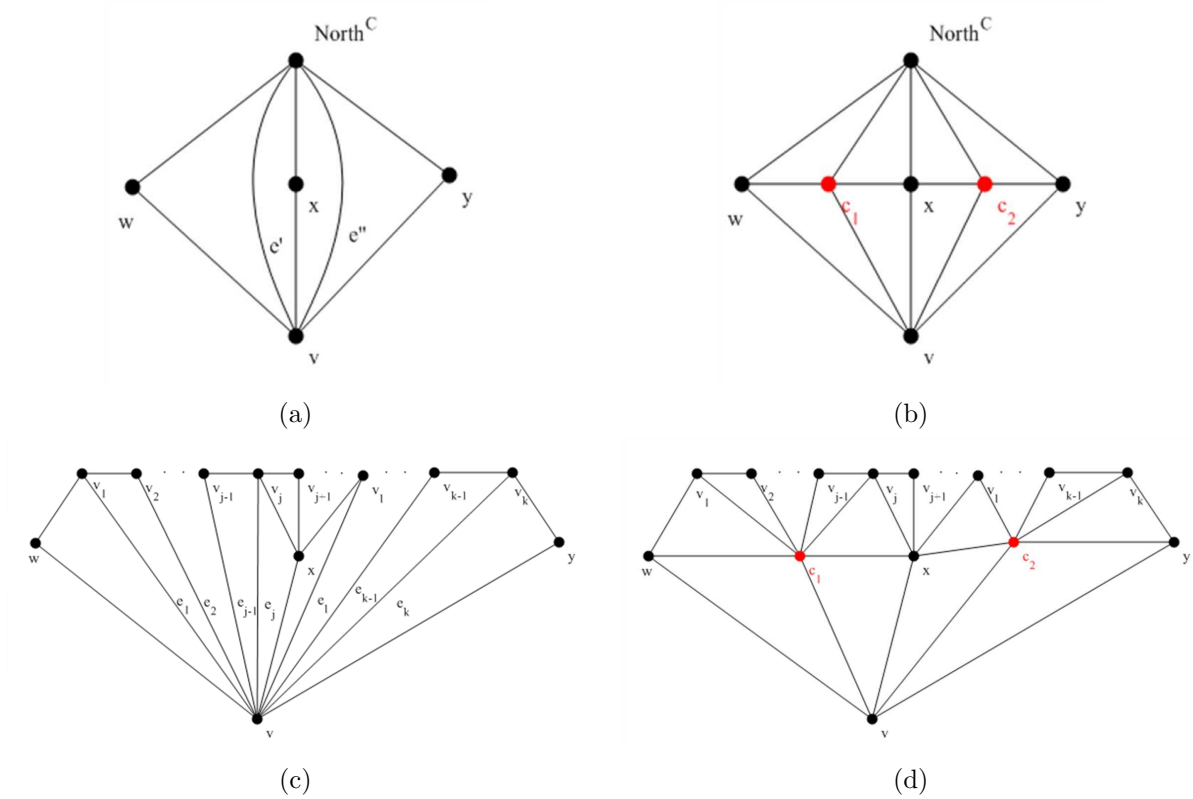


Figure 6.26: (a) $(North^C, v)$ is a multiedge to break in Q_4^C (b) c_1 and c_2 are added on each occurrence of $(North^C, v)$. (c) Situation in graph G: edges $e_1, ..., e_j$ are in the contraction list of e'; edges $e_l, ..., e_k$ are in the contraction list of e''. (d) c_1 and c_2 are added in G.

2. $M = \{M_1, ..., M_j\}$ is a set of clusters such that M_i contains the nodes of subgraph $G(\mu_i)$, $1 \leq i \leq j$.

The algorithm works as follows:

1. The quotient graph Q of $\mathscr{G}(\nu)$, ν being the root of T, is created.
2. If $\mathscr{G}(\nu)$ is badly clustered, Q is turned into a PTP graph, as done in Section 6.4.1. Any crossover vertex corresponds to a new child cluster of ν
3. For each child cluster μ of ν
   (a) The orientation of the neighbours of μ with respect to μ is obtained from a REL of Q

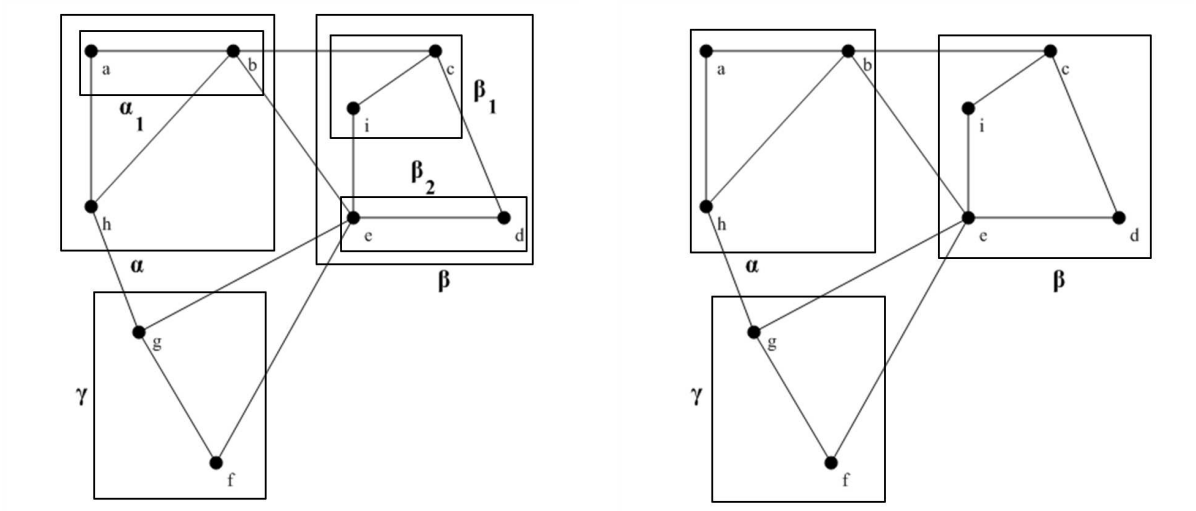


Figure 6.27: (a) A HCG $\mathscr{G} = (G, T)$. (b) Graph $\mathscr{G}(\rho)$, with ρ being the root of T.

(b) If μ contains other clusters, the rectangle graph of the HCG $(G(\mu), T_\mu)$, T_μ being the subtree of T rooted at μ, is created and this procedure is called on it recursively.

(c) If μ contains no cluster, its rectangle graph is created as done in Section 6.4.3.

4. A REL ρ is obtained merging the RELs of the rectangle graphs of the each child of ν.

There is no need of explicitly creating $\mathscr{G}(\nu)$ in Step 2 . Procedure QuotientGraph (Section 2.8.1) can be adapted to work directly on $\mathscr{G}$, although it is a HCG. In fact, when contracting an edge, it only needs to test whether it is intracluster. In a CG, this trivially reduces to checking whether the endpoints have the same parent cluster. In a HCG this is not enough, as two nodes may have different parent cluster and yet belong to the same cluster. Since clusters do not partially overlap, this occurs only when one parent cluster is subcluster of the other. However, this can be tested in constant time (procedure IsSubCluster, Section 2.8.1), thus the quotient graph of $\mathscr{G}(\nu)$ is created in $O(|V(\nu)|)$ time.

The rectangle graph Q_4^μ of a cluster μ containing other clusters is a HCG, thus procedure CreateRD described in Section 6.4.3 must be slightly modified. First of all, checking whether μ contains other clusters is accomplished in constant time. Recall that the children of μ are stored in array $Child(\mu)$ (Section 2.8), where the non-terminal nodes occur after the leaves. Thus, it suffices to check whether the last item is a leaf or a non-terminal node. Next, Procedure CreateRD (Section 6.4.3) is modified as follows (Algorithm 6.5):

Algorithm 6.5 Create the rectangle graph for a cluster containing other clusters

1: **procedure** CREATERDH($\mathscr{G}$: Graph, μ: cluster, $orientation_\mu$: array)
2: Create nodes $North^C$, $West^C$, $South^C$ and $East^C$
3: $Adj(North^C) \leftarrow [West^C, East^C]$; $Adj(West^C) \leftarrow [South^C, North^C]$;
4: $Adj(South^C) \leftarrow [East^C, West^C]$; $Adj(East^C) \leftarrow [North^C, South^C]$;
5: Create edges $e_1 = (North^C, West^C)$, $e_2 = (West^C, South^C)$, $e_3 = (South^C, East^C)$, $e_4 = (East^C, North^C)$
6: $Inc(North^C) \leftarrow [(e_1, e_4)]$; $Inc(West^C) \leftarrow [(e_2, e_1)]$;
7: $Inc(South^C) \leftarrow [(e_3, e_2)]$; $Inc(East^C) \leftarrow [(e_1, e_3)]$;
8: **for all** cluster ν in T_μ in BFS order **do**
9: CREATECLUSTER(Q_4^μ, ν, μ)
10: **end for**
11: **for all** $v \in \mu$ in a post order visit of T_μ **do**
12: INSERTNODE(Q_4^C, v_{new}, $P(v)$)
13: **end for**
14: **for all** v_{new} **do**
15: CREATENODE(Q_4^C, v_{new}, v)
16: **end for**
17: find the corner vertices $v_{top,right}$, $v_{top,left}$, $v_{bottom,left}$, $v_{bottom,right}$
18: Create adjacency and incidence lists of the four exterior vertices
19: Eliminate multiedges and separating triangles
20: **end procedure**

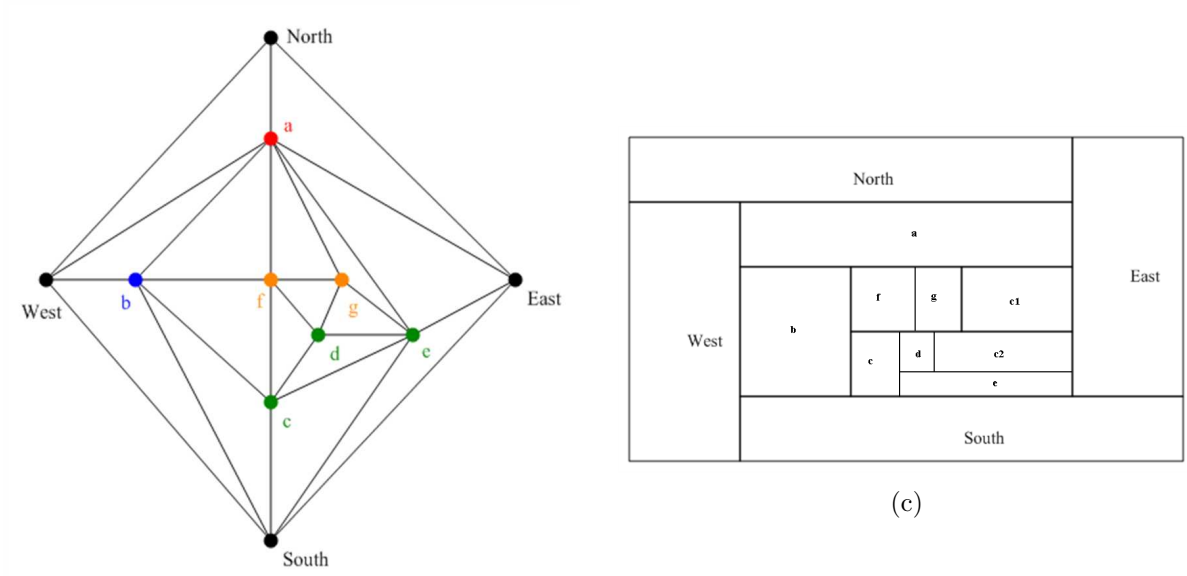
- Instructions are added to create the hierarchy tree of Q_4^μ, which is the subtree T_μ of T rooted at μ. This is done with a simple DFS visit of T.
- A postorder visit of T is used to create a node v_{new} for each node v of μ. The postorder visit maintains the invariant that nodes belonging to the same cluster are consecutively numbered so that they can be represented as the union of k_{max} intervals.

Clearly, the cost is $O(|V(\mu)|)$. Since each cluster can be contained in at most n clusters, the overall cost is $O(n^2)$.

6.6 Conclusions

In this long chapter we laid the foundations of c-rectangular dualization. We gave the admissibility conditions and we described a $O(n)$ time algorithm creating a c-rectangular dual for any c-planar c-connected CG. We also easily extended the algorithm to create a c-rectangular dual of a HCG in $O(n^2)$ time.

There are many open problems which need to be carefully investigated. As the admissibility conditions are restrictive, a c-rectangular dual is likely to have many gates. Each multiedge in fact, requires the introduction of as many crossover vertices as its occurrences. Worse than that, in a c-rectangular dual two rectangles may be adjacent through more than one gate (Fig. 6.28). This makes the drawing algorithms presented in the previous chap-



(a) (c)

Figure 6.28: (a) A CG with four clusters. (b) a and e are adjacent through two clusters (c_1 and c_2).

ter more difficult to realize; also the corresponding drawings may be negatively affected. Unfortunately, the goodness of a CG does not depend on its embedding; consequently, nothing can be done to make the input graph well-clustered without adding new vertices. Conversely, whether the rectangle graph of a cluster is a PTP depends on the REL of the quotient graph. Therefore, the number of new vertices may be minimized by choosing an appropriate REL. Our future research will investigate this point.

The algorithm creating a c-rectangular dual of a given HCG does not run in linear time in the current implementation. However, this depends on the fact that, when visiting a cluster ν, the nodes of all clusters contained in ν are visited too. To avoid this, the input HCG should be initialized in such a way that all clusters are contracted to a single macro-node. In this way, each recursive call to the procedure works on a CG (instead of a HCG), in which each cluster contains either leaf vertices or macro-nodes representing the non-terminal nodes. However, the implementation of this approach is not trivial. In fact, macros can receive one or more incident edge, when enforcing good clustering or turning a rectangle graph into a PTP. Consequently, when they are expanded, the new edges must

be correctly assigned to the vertices belonging to the macro. We are currently investigating this point.

Finally, as already pointed out, a rectangular dual of a plain graph somehow suggests a clustering of the vertex set. In other words, it is as if a rectangular dual (or, alternatively, a REL) was able to "see" a hierarchical structure which is hidden in the input graph. It would be interesting to "drive" the computation of a REL to create a c-rectangular dual (if possible) or a rectangular dual with a high number of clusters.

Chapter 7

OcORD

In this chapter, we briefly describe the main features of OcORD, our software tool for rectangular dualization.

7.1 A bit of history

When we began to get down to rectangular dualization, we were struck by the lack of tools for it. Lots of papers propose solutions and algorithms, motivating them with the relevance of rectangular dualization in important applications. We then expected to find something more related to the applications. However, it is certainly true that rectangular dualization raises challenging and interesting graph-theoretic issues; this thesis analyzed upon them extensively. For this reason, rectangular dualization has also attracted researchers more interested in these theoretical aspects, rather than the applications. This is not a blame, it is a remark. Mathematicians do not dislike to measure themselves with new issues, regardless of their practical relevance. And this is a noble attitude, as nothing can be judged only on the basis of its immediate practical implications.

Based on these considerations, we started to work on a tool entirely devoted to rectangular dualization [ACDQ06, ADQB07, AQP09, Cos04, Maz03, Vas02]. Our first goal was to extend the class of the rectangular graphs, as this is the necessary and sufficient condition for rectangular dualization to be effective in practice. Therefore, we devised the algorithm described in Section 3.8. The first implementations did not run in linear time, due to the step of breaking the separating triangles. However, in a fit of optimism, we called our tool Optimal Constructor Of a Rectangular Dual (OcORD); "optimal" does not stand for "linear-time" only, but also implies that the algorithm modifies the input graph as little as possible to turn it into a rectangular dual.

Graph Drawing was the first application we tried to apply OcORD to. This was a novelty, as rectangular dualization was never used in Graph Drawing, except for some mentions in [Kan93]. More surprisingly, the second application of OcORD was the generation of virtual worlds specified on the basis of the Electronic Institution metaphor (Section 5.4). We say "surprisingly", as this application apparently has nothing to do with the traditional applications of rectangular dualization.

OcORD has been implemented using Microsoft Visual Studio C# 2005. Its core is the rectangular dualization algorithm described in Section 3.8. Each application has been realized as a separate module, in order to make OcORD independent of the applications using it.

7.1.1 OcORD Today

One of the worst drawback of the past versions of OcORD were the lack of a graphic interface. OcORD was indeed a simple command-line tool; thus, the only way to input a graph was to manually specify the adjacency list of each vertex. Evidently, this task was unbearable and error-prone even with small graphs ($40 - 50$ nodes). Moreover, the visualization of the results (rectangular dual and graph drawings obtained from a rectangular dual) was committed to external tools such as GFig. It was soon evident that to use OcORD for visualizing graphs we needed to provide it with a (even simple) graph editor. Indeed, a command-line tool for visualization would be quite a funny contradiction!

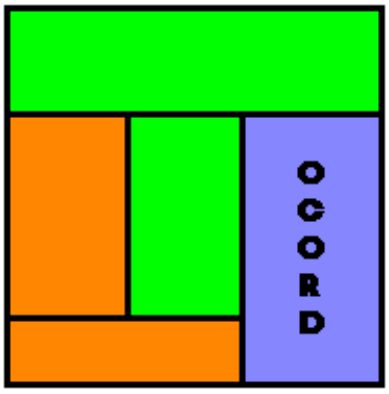


Figure 7.1: OcORD Logo. Realized by Federica Pian.

Besides a graph editor, OcORD also features:

- An simple XML-based language for describing the graphs created with the text editor.

- A full linear-time algorithm for rectangular dualization, implementing all algorithms described in this thesis.
- A simple random graph generator, which we use for testing purposes.
- A tool for drawing graphs, based on rectangular dualization. For now, only orthogonal drawings and bus-mode drawings are used.
- A tool for c-rectangular dualization.
- A tool for creating the floorplan of a 3D virtual building, based on the Performative Structure describing an Electronic Institution.

7.2 The Graph Editor

The OcORD graph editor is composed of a drawing area and a toolbar providing basic facilities for drawing graphs (Fig. 7.2). The editor provides for two working modes. The *drawing mode* allows nodes and edges to be drawn; in the *selection mode*, the appearance of nodes and edges can be modified as well as their position and shape. Edges are modelled as Bezier curves, which are widely used in computer graphics, as they appear reasonably smooth at all scales, as opposed to polygonal lines [1]. The interaction with a Bezier curve may not be intuitive. In fact, the curve is modified dragging its control points, which do not lay on the curve. Consequently, the curve may not change as expected. At a first glance, the use of curvilinear edges may appear a kind of unneeded whim; after all, every planar graph admits a straight-line plane embedding (Section 1.2.4). However, a straight-line drawing requires the nodes to be appropriately placed in the drawing area, to prevent edge crossings. Typically, one realizes that the position of a node is wrong only when two edges cross. If curvilinear edges are not allowed, edge crossings can be eliminated only by moving (potentially all) vertices, which makes cumbersome the creation of new graphs.

7.2.1 How does OcORD Conceive a Drawing?

Graph drawing tackles the problem of translating a set of sterile adjacency lists into a human-readable drawing. This is needed to better understand the output of an algorithm working on a graph. To produce an output, however, the algorithm has first to understand the input. A drawing is not an acceptable input. Therefore, we need to realize a "translator", which turns a graph drawing into a set of adjacency lists. Luckily, realizing

[1] The use of Bezier curves in computer graphics was promoted by a French automobile engineer, Pierre Bezier, hence the name.

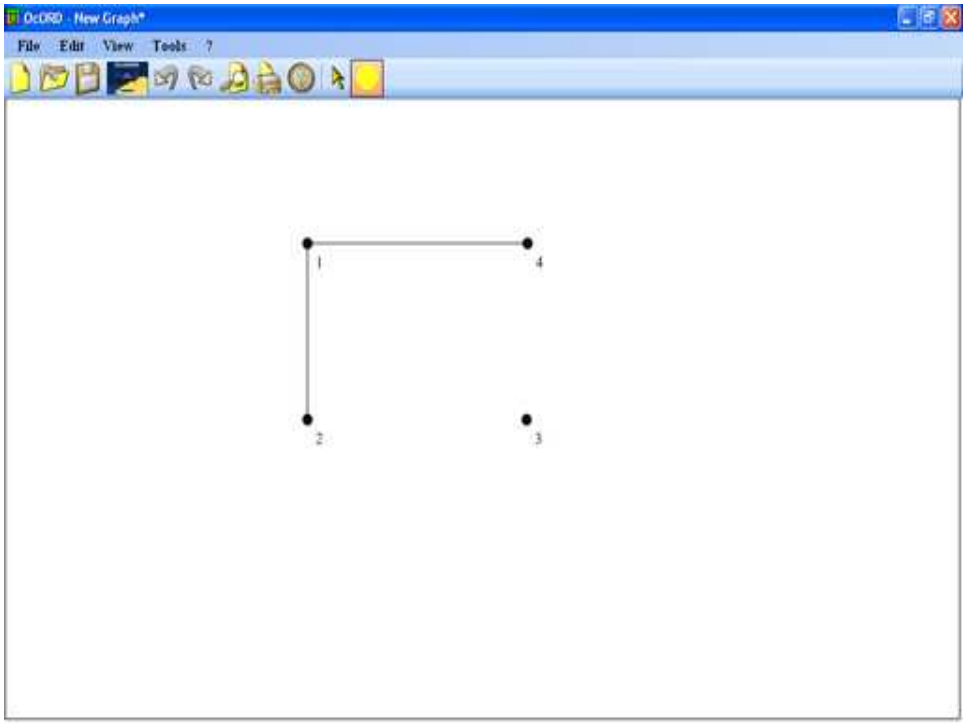


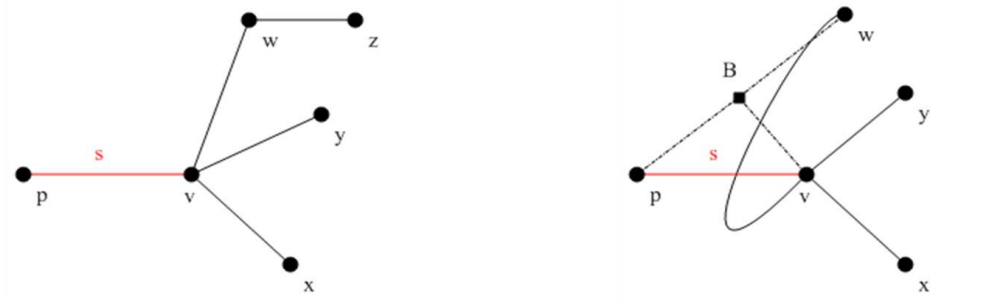
Figure 7.2: OcORD Graphic Interface.

such a "translator", though it takes a lot of coding, is not as tricky as designing a graph drawing algorithm. The adjacency list $Adj(v)$ of a node v must contain all neighbours of v in clockwise order around v. If u precedes w in clockwise direction, and (v, u), (v, w) are straight lines, then u is on the left of the straight line directed from v to w (Fig. 7.3). This does not hold for x and z, as $e = (v, z)$ is a curvilinear edge (Fig. 7.3). In this case, we approximate e with a straight-line joining v to z'. Thus, the adjacency list of $Adj(v)$ is obtained by radially sort the neighbours of v around v; using Merge Sort, the creation of all adjacency lists takes $O(|N(v)| \log |N(v)|)$ time.

Specifying the adjacency list for all vertices does not uniquely identify an embedding of a graph. Also, the nodes incident with the exterior face must be determined. Let (v_x, v_y) be the coordinates of the node v having minimum $x-$coordinate. Clearly, if all edges are straight lines, v is an exterior node. To determine the vertex w following v in clockwise direction on the exterior cycle (or path), a node p, with coordinates $(-1, v_y)$ is added to $Adj(v)$, which is sorted in clockwise order around v, as done before (Fig. 7.4 (a)). w is the vertex following p in $Adj(v)$. Similarly, the next exterior node after w is the one following v in $Adj(w)$. Iterating this procedure, all exterior vertices are found. If some edge is curvilinear, three cases can occur (Fig. 7.4 (b) and 7.5). First, the edge s connecting p to

(a) (b)

Figure 7.3: (a) u precedes w in clockwise order around v and is on the left of the straight line directed from v to w. The same does not hold for z and x. (b) Edge (v, z) is approximated with a straight line (v, z').



(a) (b)

Figure 7.4: (a) All edges are straight lines. v is an exterior node (b) s intersects a curvilinear edge incident with v. B is the barycentre of triangle (p, v, w). Since v precedes p in clockwise order around B, v precedes w on the exterior path/cycle.

v has zero or an even number of intersection points with a a curvilinear edge incident with a node different than v; v is an exterior node and the exterior path/cycle is determined as before. Second, s intersects a curvilinear edge $e = (v, w)$incident with v; v and w are exterior vertices. In order to determine whether v precedes w, the triangle (p, v, w) is created. If the three points are collinear, the $x-$coordinate of p is changed to -2. Then, the barycentre of this triangle is computed and the three vertices are radially sorted around it. If v precedes p, then v precedes w on the exterior cycle/path. Third, s intersects an odd number of times a curvilinear edge $e = (u, w)$ incident with a node different than v. We proceed as in the second case. Finally, if s intersects more than one curvilinear edge, the most "external" one is considered. The most external edge e is such that its intersection point with s has the least $x-$coordinate. If e is found to be an exterior edge, the algorithm stops. Otherwise, the next more external edge is considered.

Figure 7.5: (a) s has an even number of intersections with a curvilinear edge. v is an exterior node (b) s intersects a curvilinear edge (w, u) an odd number of times. w and u are exterior vertices.

7.3 XML for Graphs

Lots of words have been spent to (rightly) praise XML and its virtues. Jon Bosak, who led the creation of XML at the W3C, claimed that "XML is the digital dial tone of the Web"; Peter Murray Rust (Unilever Centre for Molecular Sciences Informatics) added: "I assume that there are now (or soon will be) chips that are XML-aware. I love it.". These two colourful quotations define XML better than thousands words. Born to satisfy the demands for a more flexible markup language to complement HTML, it is now widely used in a number of different applications. XML, in fact, is a *metalanguage*, which allows the creation and definition of new markup languages, which adapt to different contexts. SVG, for example, is a standard XML-based language for describing two-dimensional graphics and graphical applications. XML-RPC is a remote procedure calling protocol, working over the Internet, defined with XML. SMIL allows the integration of a set of independent multimedia objects into a synchronized multimedia presentation.

Technically, an XML file is nothing more than a textual file, characterized by a hierarchical structure described with a *document tree*. The nodes of the document tree are called *elements*; each element may contain other elements and may be associated with some *attributes*, which contain additional information. A special element, called *root element*, contains all elements in the file. The syntax is defined in a Document Type Definition (DTD), which describes the elements of the language, their relationships and the attributes allowed for each element.

Up till now, there is no standard XML language for describing graphs. The most complete are undoubtedly GXL and GraphML. GXL was created to enable interoperability between software reengineering tools and components like parsers, analyzers and visualizers [HW00, Win02]. Examples of visualization tools using GXL are TouchGraph[2], a tool popular

[2]http://www.touchgraph.com/

among Facebook users, and Graphviz [EGK+02]. GXL has also been designed to represent typed, attributed, directed, ordered graphs, and allows the definition of extensions to handle hypergraphs and hierarchical graphs. GraphML provides a language for describing the structure of a graph and a mechanism to easily define extensions to add application-specific data [BEH+02]. It supports hypergraphs, nested graphs and directed graphs.

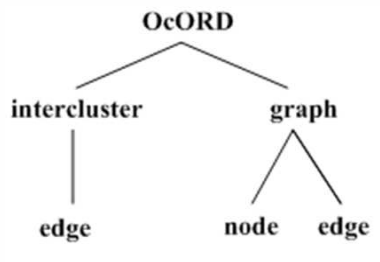


Figure 7.6: The document tree of the XML language used by OcORD for describing a graph.

OcORD uses its own XML-language for describing a graph (Fig. 7.7). This choice is motivated both by the lack of a standard and the need of a simple language. Although the DTD is very simple, this language can represent simple plain and clustered graphs (possibly multigraphs with loops). Five elements are defined: <OcORD>, <graph>, <node>, <edge> and <interclusterEdge>, which are related as shown in Fig. 7.6. The root element <OcORD> has three attributes:

- **version**. A progressive number, identifying the current release of OcORD. As OcORD may change, the XML-language may need to adapt to the new requirements of OcORD. With this attribute, it will possible to use in the current release any file generated with older versions.
- **width** and **height**. The screen resolution of the computer used to generate the file. Each node, in fact, is identified with coordinates depending on screen resolution. If the file is opened using a different resolution, the graph is resized to fit into the screen.

In OcORD, a graph is described as a collection of graphs, each representing a cluster. A cluster is defined using the element <graph>, identified by an integer number (attribute **id**). Each graph is composed of nodes and edges. Elements <node> and <edge> have a set of attributes describing what nodes and edges look like. The following are attributes common to <node> and <edge>:

- **id**. An integer identifier.

```
<?xml version="1.0" encoding="utf-8"?>
<OCORD version="1" width="1280" height="681">
<graph id="1">

    <node id="0" label="Marco" X="360" Y="158" sizeX="10" sizeY="10"
    color="Black" shape="Circle" labelX="338" labelY="128" labelColor="Black"
    FontFamily="Times New Roman" FontSize="11" FontStyle="Regular" />

    <node id="1" label="Luca" X="487" Y="158" sizeX="10" sizeY="10"
    color="Black" shape="Circle" labelX="480" labelY="128" labelColor="Black"
    FontFamily="Times New Roman" FontSize="11" FontStyle="Regular" />

    .
    .
    .

    <edge id="6" label="" endlID="2" end2ID="3" isBezier="False" ctrl1X="487" ctrl1Y="272"
    ctrl2X="495.3" ctrl2Y="272" ctrl3X="561.7" ctrl3Y="272" ctrl4X="570" ctrl4Y="272"
    color="Black" thickness="1" style="Solid" labelX="508.5" labelY="252" labelColor="#000000"
    FontFamily="Times New Roman" FontSize="11" FontStyle="Regular" />

    <edge id="7" label="" endlID="1" end2ID="3" isBezier="False" ctrl1X="487" ctrl1Y="158"
    ctrl2X="495.3" ctrl2Y="169.4" ctrl3X="561.7" ctrl3Y="260.6" ctrl4X="570" ctrl4Y="272"
    color="Black" thickness="1" style="Solid" labelX="508.5" labelY="195" labelColor="#000000"
    FontFamily="Times New Roman" FontSize="11" FontStyle="Regular" />

</graph>
</OCORD>
```

Figure 7.7: The graph displayed in Section 1.1 described with the OcORD XML language.

- **label**. It is an alphanumeric string, which is displayed in the drawing area close to the node/edge it refers to.
- **color** and **labelcolor**. Color of the node/edge and of their label.
- **labelX** and **labelY**. Coordinates of the label.
- **FontFamily**, **FontStyle**. Describe the font used to write the label.

Attributes specific for nodes are:

- **X** and **Y**. Coordinates of the node.
- **sizeX** and **sizeY**. Size of the node.
- **shape**. The geometric shape used to represent a node. In the current release, the allowed shapes are circle, square and triangle.

Finally, the following are attributes specific for edges:

- **end1ID** and **end2ID**. Identifiers of the endpoints of the edge.

- **isBezier**. A boolean attribute stating whether the edge is curvilinear or not.
- **ctrl1X**, **ctrl1Y** and **ctrl4X**, **ctrl4Y**. Coordinates of the first and second endpoints of the edge.
- **ctrl2X**, **ctrl2Y** and **ctrl3X**, **ctrl3Y**. Coordinates of the control points of the the edge.
- **thickness** and **style**. Edge thickness and style (Solid, Dashed, Dot-Dashed).

The element <intracluster> is used to list all intercluster edges. In this case, the endpoint v of each edge e is identified by a pair like $idg : idv$, where idg is the identifier of the cluster v belongs to and idv is the identifier of v.

7.4 Electronic Institutions and OcORD

As described in Section 5.4, an Electronic Institution is specified using a software called ISLANDER. The output of ISLANDER is an XML file, which defines the roles allowed within the institution, the language that agents use to interact and the performative structure, that is the graph representing the institution.

The only element of the ISLANDER output file which OcORD is concerned of is that defining the Performative Structure, which contains all the elements describing the graph. The Performative Structure has two special nodes, called *initial node* and *final node*, which determine where the activity flow within the institution begins and ends. Usually, these nodes are not mapped to any room in the 3D virtual building created from the rectangular dual. The other nodes can be either *scene nodes* or *transition nodes*. The first correspond to rooms where the activities actually take place, whereas the second only serve as transition between two scene nodes. Consequently, not always transition nodes are visible in the virtual building.

The performative structure is provided with an embedding. The coordinates of each node are given as well as the control points of the Bezier curves representing the edges. If the embedding is plane, it is trivial to import the performative structure into OcORD. Fig. 7.8 shows the performative structure of the Trade Institution described in Section 5.4.1; the initial and final nodes are coloured in red, the scene nodes are in black and the transition nodes are blue triangles. Although the performative structure in ISLANDER is a directed graph, in OcORD the orientation of the edges does not matter. When a performative structure is displayed in the drawing area, a toolbar appears to let users hide/show the initial and final nodes and the transition nodes. The removal of the initial and final node

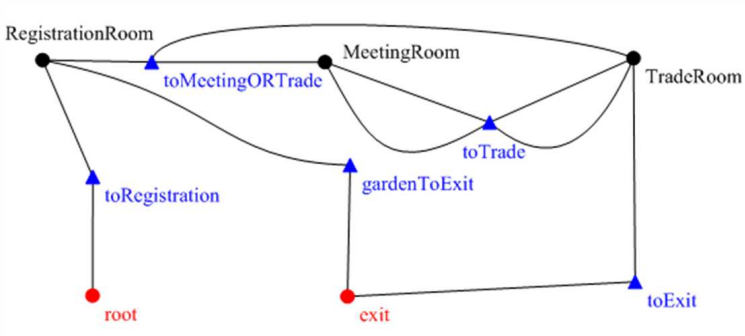


Figure 7.8: The Performative Structure of the Trade Institution visualized in OcORD.

maintains the graph planar and connected, as their degree is always 1. Removing the transition nodes is more complicated (as explained in Section 5.4.2).

If the embedding provided by ISLANDER is not plane, OcORD computes another. Trying to modify the given embedding to obtain a plane one would not be an efficient approach, as the performative structure may also have no plane embedding. Thus, OcORD uses the Boyer-Myrvold algorithm [BM99] to create a plane embedding, if one exists. The drawing of the Performative Structure is then created using one of the algorithms seen in Section 5.1.

Chapter 8

Conclusions and Future Work

In this thesis, we studied rectangular dualization and its applications. We described a linear-time algorithm that creates a rectangular dual of a plane graph, after enforcing the admissibility conditions. This is an important contribution of the thesis, as the lack of efficient algorithms has been the major limitation of rectangular dualization. In particular, a linear-time algorithm is absolutely needed to use rectangular dualization as a graph drawing method.

8.1 The algorithm

The algorithm described in Section 3.8 can still be improved. The edges added on the biconnectivity augmentation step can create new separating triangles, which need to be broken. As we have seen, each crossover vertex makes the area of the rectangular dual grow. Moreover, it breaks the adjacency between two nodes. Most of biconnectivity augmentation algorithms aim at minimizing the number of edges. However, any new edge does not impact on the area of the rectangular dual. Therefore, in our algorithm, it is better to avoid the creation of new separating triangles, even if this implies a higher number of edges than needed. As for separating triangles , the heuristic algorithms we presented in Chapter 4 provide a highly acceptable solution, at least in all cases that are likely to occur in practice. In the light of the obtained results, looking for an exact linear-time algorithm still remains our objective, but is more interesting from a theoretical point of view. Up to now, we have analyzed the behaviour of the heuristics only on randomly generated graphs, in which large islands of separating triangles are unlikely to occur. However, evaluating them in more complex islands might give precious hints towards a simple exact algorithm.

8.2 Contribution to Graph Drawing

The first results obtained in graph visualization seem to be promising. In its current version, OcORD creates box-rectangular drawings, orthogonal drawings and bus-mode drawings of plane graphs using their rectangular dual. In all cases, the algorithms are fast and simple and produce drawings with the following properties:

1. The angular resolution is optimal. In particular, a bus-mode drawing extends orthogonal drawing to graphs with degree greater than 4, while maintaining a $\frac{\pi}{2}$ angular resolution.
2. The area of the drawing can be optimized, by first optimizing the area of the rectangular dual. To this extent, techniques used in VLSI floorplanning can be used.
3. The aspect ratio is maximized similarly, as the drawing is contained in the enclosure rectangle of the rectangular dual.

As for bend minimization, some issues have to be carefully analyzed. In particular, the orthogonal drawings created with the algorithm described in Chapter 5 may have more bends than needed. On the other hand, in a bus-mode drawing bends are small curved edges, which do not negatively impact the appearance of a drawing.

Bus-mode drawing is particularly attractive, as it greatly reduces the number of edges. This makes it appealing for the visualization of dense graphs as well as graphs having vertices with high degree. Moreover, when it comes to non-planar graphs, one of the key issues is the minimization of edge crossings. A bus-mode drawing answers perfectly this need. In fact, some edge crossings may be hidden in the bundles; as a result, even a non-planar graph may have an embedding with no apparent edge crossings. It would be interesting to quantify the edge reduction in a bus-mode drawing and evaluate its impact on the readability of a graph, using experiments such as the ones described in Section 1.3.1.

One major contribution of rectangular dualization to graph visualization is that it does not impose any particular drawing styles; this makes it adaptable in many contexts. As future work, we reserve to study algorithms for visualizing graphs with other widely-used drawing styles (such as, for instance, rectangular drawing). Also, the use of c-rectangular dualization to create drawings of hierarchically clustered graphs will be further investigated

8.3 C-Rectangular Dualization

We observed that a rectangular dual of a plain graph is likely to contain subgraphs that are themselves rectangular duals. At a first glance, it may appear that such a hierarchical

structure depends on some choice of the rectangular dualization algorithm. This may be true for the Bhasker-Sahni algorithm, as it disposes the rectangles hierarchically from top to bottom. However, it is not the case of the Kant-He algorithm, which computes a rectangular dual from a labelling of the edges and does not work directly with the coordinates of the rectangles. Thus, we can conclude that a rectangular dual has an intrinsic hierarchical structure. Dually, the primal graph has the same structure, although it is not visible. It is as if the rectangular dual was able to bring it to light. This is particularly interesting, as a rectangular dual may suggest a clustering of the vertex set in the primal graph.

Based on these observations, we laid the foundations of c-rectangular dualization, which extends rectangular dualization to HCGs. The admissibility conditions are quite restrictive and enforcing them is not trivial. In particular, this may imply the addition of a high number of crossover vertices, with the consequent increase of the area of the rectangular dual. In Chapter 6 we described an algorithm which creates a c-rectangular dual of a c-planar c-connected HCG, after enforcing the admissibility conditions. Much work has to be done to improve the performances of this algorithm. In particular, in a CG, great part of the efficiency of the result seems to depend on the REL of the quotient graph. As a PTP may have many RELs, it is not trivial to decide which one minimizes the number of crossover vertices added. We are currently investigating this point.

Bibliography

[AAV00] A. Accornero, M. Ancona, and S. Varini. All separating triangles in a plane graph can be optimally "broken" in polynomial time. *International Journal of Foundations of Computer Science*, 11(3):405–421, 2000.

[ABFN00] J. Alber, H. Bodlaender, H. Fernau, and R. Niedermeier. Fixed parameter algorithms for PLANAR DOMINATING SET and related problems. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 97–110. Springer-Verlag, London, UK, 2000.

[ACDQ06] M. Ancona, W. Cazzola, S. Drago, and G. Quercini. Visualizing and managing network topologies via rectangular dualization. In 11^{th} *IEEE Symposium on Computers and Communications*, pages 1000–1005, Washington, DC, USA, 26-29 June 2006. IEEE Computer Society.

[ADQB07] M. Ancona, S. Drago, G. Quercini, and A. Bogdanovych. Rectangular dualization of biconnected plane graphs in linear time and related applications. In Vincenzo Cutello, Giorgio Fotia, and Luigia Puccio, editors, *Applied and Industrial Mathematics in Italy II*, volume 75 of *Series on Advances in Mathematics for Applied Sciences*, pages 37–48. World Scientific Publishing, Singapore, August 2007.

[AN02] J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In Sergio Rajsbaum, editor, *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, volume 2286 of *Lecture Notes In Computer Science*, pages 613–628. Springer-Verlag, London, UK, 2002.

[AQP09] M. Ancona, G. Quercini, and P. Pastorelli. A new tool for rectangular dualization. Accepted for publications in Communications to SIMAI Congress, vol. 3, 2009.

[BBDL01] T. C. Biedl, P. Bose, E. D. Demaine, and A. Lubiw. Efficient algorithms for Petersen's matching theorem. *J. Algorithms*, 38:110 – 134, 2001.

[BCF01] J. D. Boissonnat, F. Cazals, and J. Flötotto. 2D-structure drawings of similar molecules. In *Proceedings of the 8th International Symposium on Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 115–126. Springer-Verlag, London, UK, 2001.

[BD06] A. Bogdanovych and S. Drago. Euclidean representation of 3d electronic institutions: automatic generation. In *Proceedings of the working conference on Advanced visual interfaces*, pages 449–452, New York, NY, USA, 2006. ACM.

[BD07] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *Proceedings of the 6th International Asia-Pacific Symposium on Visualization,*, pages 133–140. IEEE Press, 2007.

[BEF+98] P. Bose, H. Everett, S. P. Fekete, M. E. Houle, A. Lubiw, H. Meijer, K. Romanik, G. Rote, T. C. Shermer, S. Whitesides, and C. Zelle. A visibility representation for graphs in three dimensions. *Journal of Graph Algorithms and Applications*, 2(3):1 – 16, 1998.

[BEH+02] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report structural layer proposal. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 109–112. Springer-Verlag, London, UK, 2002.

[BES+07] A. Bogdanovych, M. Esteva, S. Simoff, C. Sierra, and H. Berger. A methodology for 3D electronic institutions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, New York, NY, USA, 2007. ACM.

[BEW95] R. A. Becker, S. G. Eick, and A. R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995.

[BGPV08] A. L. Buchsbaum, E. R. Gansner, C. M. Procopiuc, and S. Venkatasubramanian. Rectangular layouts and contact graphs. *ACM Transactions on Algorithms*, 4(1):1–28, 2008.

[BGV07] A. L. Buchsbaum, E. R. Gansner, and S. Venkatasubramanian. Directed graphs and rectangular layouts. In *Asia-Pacific Symposium on Visualisation*, pages 61–64. IEEE Press, 2007.

[BK97] T. Biedl and M. Kaufmann. Area-efficient static and incremental graph drawings. In *Proceedings of the 5th Annual European Symposium on Algorithms*,

volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, London, UK, 1997.

[BKK94] T. Biedl, G. Kant, and M. Kaufmann. On triangulating planar graphs under the four-connectivity constraint. In *Algorithm Theory - SWAT '94*, volume 824 of *Lecture Notes in Computer Science*, pages 83–94. Springer, Berlin, 1994.

[BM99] J. Boyer and W. Myrvold. Stop minding your P's and Q's: a simplified O(n) planar embedding algorithm. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 140–146, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[BMK96] J. Blythe, C. McGrath, and D. Krackhardt. The effect of graph layout on inference from social network data. In *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, London, UK, 1996.

[Bod93] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–22, 1993.

[Bog07] A. Bogdanovych. *Virtual Institutions*. PhD thesis, University of Technology Sydney, Sydney, Australia, 2007.

[Bol02] B. Bollobas. *Modern Graph Theory*. Springer-Verlag, London, UK, 2002.

[Bra01] J. Branke. *Dynamic graph drawing*, volume 2025 of *Lecture Notes in Computer Science*, chapter 9, pages 228–246. Springer-Verlag, London, UK, 2001.

[Bro02] H. J. Broersma. On some intriguing problems in hamiltonian graph theory: a survey. *Discrete Mathematics*, 251(1-3):47–69, 28 May 2002.

[BS86] J. Bhasker and S. Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. In *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 108–114, Piscataway, US, 1986. IEEE Press.

[BS07] S. Bafna and S. Shah. The evolution of orthogonality in built space: an argument from space syntax. In *Proceedings of the 6th International Symposium on Space Syntax*, pages 054–01 – 054–14, 2007.

[CKW04] M. Chimani, G. W. Klau, and R. Weiskircher. Non-planar orthogonal drawings with fixed topology. In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 96–105. Springer-Verlag, London, UK, 2004.

[CMS99] S. K. Card, J. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, San Francisco, USA, 1st edition, 1999.

[CN85] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[CN89] N. Chiba and T. Nishizeki. The hamiltonian cycle problem is linear-time solvable for 4-connected planar graphs. *J. Algorithms*, 10(2):187–211, June 1989.

[CON85] N. Chiba, K. Onoguchi, and T. Nishizeki. Drawing planar graphs nicely. *Acta Informatica*, 22:187 – 201, 1985.

[Cos04] M. Costa. Disegno ortogonale automatico di grafi planari attraverso la costruzione del duale rettangolare. Master's thesis, University of Genoa, Department of Computer Science, Genoa, Italy, 2004.

[CZQ+08] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.

[dBMS08] M. de Berg, E. Mumford, and B. Speckmann. On rectilinear duals for vertex-weighted plane graphs. *Discrete Mathematics*, 2008. Article in press.

[DDGC97] T. K. Dey, M. B. Dillencourt, S. K. Ghosh, and J. M. Cahill. Triangulating with high connectivity. *Comput. Geom. Theory Appl.*, 8(1), 1997.

[DF96] R. G. Downey and M.R. Fellows. *Parameterized Complexity*, volume XV of *Monographs in Computer Science*. Springer, Berlin, Heidelberg, 1996.

[dFdM01] H. de Fraysseix and P. O. de Mendez. Connectivity of planar graphs. *J. Graph Algorithms Appl.*, 5(5):93–105, 2001.

[DFPP90] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41 – 51, 1990.

[Die05] R. Diestel. *Graph Theory*. Springer-Verlag, London, UK, 2005.

[DSK97] P. Dasgupta and S. Sur-Kolay. Slicibility of rectangular graphs and floorplan optimization. In *Proceedings of the 1997 international symposium on Physical design*, pages 150–155, New York, NY, USA, 1997. ACM.

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congr Numer.*, 42:149 – 160, 1984.

[EdlCS02] M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1045–1052, New York, NY, USA, 2002. ACM.

[EF97] P. Eades and Q. W. Feng. Drawing clustered graphs on an orthogonal grid. In *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, London, UK, 1997.

[EFH00] P. Eades, Q. W. Feng, and M. L. Huang. Navigating clustered graphs using force-directed methods. *Journal of Graph Algorithms and Applications*, 4(3):157–181, 2000.

[EFK00] M. Eiglsperger, U. Fössmeier, and M. Kaufmann. Orthogonal graph drawing with constraints. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 3–11, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[EFK01] M. Eiglsperger, S. Fekete, and G. W. Klau. *Orthogonal graph drawing*, volume 2025 of *Lecture Notes in Computer Science*, chapter 6, pages 121–171. Springer-Verlag, London, UK, 2001.

[EFL96] P. Eades, Q. W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, London, UK, 1996.

[EFN99] P. Eades, Q. W. Feng, and H. Nagamochi. Drawing clustered graphs on an orthogonal grid. *Journal of Graph Algorithms and Applications*, 3(4):3–29, 1999.

[EGK+02] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz: Open source graph drawing tools. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer-Verlag, London, UK, 2002.

[EK05] C. Erten and S. G. Kobourov. Simultaneous embedding of planar graphs with few bends. *Journal of Graph Algorithms and Applications*, 9(3):347–364, 2005.

[ELMS91] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. *Proc. of Compugraphics*, 91:24 – 33, 1991.

[ELT96] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry and Applications*, 6(2):145–156, 1996.

[Eul41] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae*, 8:128 – 140, 1741.

[Far48] I. Fary. On straight line representation of planar graphs. *Acta Sci. Math. Szeged*, 11:229 – 233, 1948.

[FCE95a] Q. W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. In *Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, London, UK, 1995.

[FCE95b] Q. W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs. In *Algorithms - ESA '95*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer-Verlag, London, UK, 1995.

[Fen96] P. Eades Q. W. Feng. Multilevel visualization of clustered graphs. In *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 101–112. Springer-Verlag, London, UK, 1996.

[Fen97] Q. Feng. *Algorithms for Drawing Clustered Graphs*. PhD thesis, University of Newclastle, Newcastle, Australia, April 1997.

[Fer06] H. Fernau. Roman Domination : A parameterized perspective. In *Theory and Practice of Computer Science*, volume 3831 of *Lecture Notes in Computer Science*, pages 262–271. Springer, Berlin, Heidelberg, 2006.

[FH01] R. Fleischer and C. Hirsh. *Graph drawing and its applications*, volume 2025 of *Lecture Notes in Computer Science*, chapter 1, pages 1–22. Springer-Verlag, London, UK, 2001.

[FK96] U. Fössmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In *Proceedings of the Symposium on Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 254–266. Springer-Verlag, London, UK, 1996.

[FM98] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 260–269, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[Fre80] R. S. Frew. A survey of space allocation algorithms in use in architectural design in the past twenty years. In *Proceedings of the 17th conference on Design automation*, pages 165–174, New York, NY, USA, 1980. ACM.

[Fre00] L. C. Freeman. Visualizing social networks. *Journal of Social Structure*, 1(1):128 – 140, 2000.

[Fri26] O. Frink. A proof of Petersen's theorem. *Ann. Math. Ser. 2*, 27:491 – 493, 1925-1926.

[FT04] Y. Frishman and A. Tal. Dynamic drawing of clustered graphs. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 191–198, Washington, DC, USA, 2004. IEEE Computer Society.

[GFC05] M. Ghoniem, J. D. Fekete, and P. Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2), 2005.

[Gra71] J. Grason. An approach to computerized space planning using graph theory. In *Proceedings of the 8th workshop on Design automation*, pages 170–178, New York, US, 1971. ACM.

[GT02] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2002.

[Gus88] D. Gusfield. A graph theoretic approach to statistical data security. *SIAM J. Comput.*, 17(3):552–571, 1988.

[Har94] F. Harary. *Graph Theory*. Westview Press, Boulder, CO, US, 1994.

[He96] X. He. Grid embedding of 4-connected plane graphs. In *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 287–299. Springer, Berlin, Heidelberg, 1996.

[He97] X. He. On floorplans of planar graphs. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 426–435, New York, US, 1997. ACM.

[HH01] R. Hadany and D. Harel. A multi scale algorithm for drawing graphs nicely. *Discrete Appl. Math.*, 113(1):3–21, 2001.

[HK00] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *Proceedings of the working conference on Advanced visual interfaces*, pages 282–285, New York, NY, USA, 2000. ACM.

[HMM00] I. Herman, G. Melanon, and M.S. Marshall. Graph visualization and navigation in information visualisation: a survey. *EEE Transactions on Visualization and Computer Graphics*, 6(1):24 – 43, 2000.

[Hom06] H. Homayouni. A survey of computational approaches to space layout planning. Technical report, Department of Architecture and Urban Planning, University of Washington, Washington, US, 2006.

[Hsu00] T. S. Hsu. On four-connecting a triconnected graph. *Journal of Algorithms*, 35(2):202–234, 2000.

[HW00] R. C. Holt and A. Winter. A short introduction to the GXL software exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 299, Washington, DC, USA, 2000. IEEE Computer Society.

[JAY05] S. G. Kobourov J. Abello and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 431–441. Springer-Verlag, London, UK, 2005.

[JG86] S. P. Jain and K. Gopal. On network augmentation. *IEEE Transactions on Reliability*, R-35(5):541–543, 1986.

[Jor93] T. Jordán. Increasing the vertex-connectivity in directed graphs. In *Algorithms. ESA '93*, volume 726 of *Lecture Notes in Computer Science*, pages 236–247. Springer, Berlin, Heidelberg, 1993.

[Jor95] T. Jordán. On the optimal vertex-connectivity augmentation. *J. Comb. Theory Ser. B*, 63(1):8–20, 1995.

[Kan93] G. Kant. *Algorithms for drawing planar graphs.* PhD thesis, Dept. Comput. Sci., Univ. Utrecht, Netherlands, 1993.

[KB91] G. Kant and H. L. Bodlaender. Planar graph augmentation problems. In *Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 286–298. Springer, Berlin, 1991.

[KH94] G. Kant and X. He. Two algorithms for finding rectangular duals of planar graphs. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 396–410, London, UK, 1994. Springer-Verlag.

[KK84] K. Kozminski and E. Kinnen. An algorithm for finding a rectangular dual of a planar graph for use in area planning for VLSI integrated circuits. In *Proceedings of the 21st conference on Design automation*, pages 655–656, Piscataway, US, 1984. IEEE Press.

[KK88] K. Kozminski and E. Kinnen. Rectangular dualization and rectangular dissections. *IEEE Transactions on Circuits and Systems*, 35(11):1401–1416, 1988.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989.

[KM98] G. W. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.

[KMBW02] E. Kruja, J. Marks, A. Blair, and R. C. Waters. A short note on the history of graph drawing. In *Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 272–286. Springer-Verlag, London, UK, 2002.

[LL88] Y. T. Lai and S. M. Leinwand. Algorithms for floorplan design via rectangular dualization. *IEEE Transactions on Computer-Aided Design*, 7(12):1278 – 1289, 1988.

[LL90] Y. T. Lai and S.M. Leinwand. A theory of rectangular dual graphs. *Algorithimica*, 5(1-4):467–483, 1990.

[LLY03] C. C. Liao, H. I. Lu, and H. C. Yen. Compact floor-planning via orderly spanning trees. *J. Algorithms*, 48(2):441–451, 2003.

[LR01] I. Ljubic and G. R. Raidl. An evolutionary algorithm with stochastic hill-climbing for the edge-biconnectivity augmentation problem. In *Proceedings of the EvoWorkshops on Applications of Evolutionary Computing*, pages 20–29, London, UK, 2001. Springer-Verlag.

[Mav70] T. W. Maver. A theory of architectural design in which the role of the computer is identified. *Building Science*, 4(4):199–207, 1970.

[Maz03] P. A. Mazzarello. Costruzione ottimale del duale rettangolare di un grafo planare arbitrario. Master's thesis, University of Genoa, Department of Computer Science, Genoa, Italy, 2003.

[MB95] T. Munzner and P. Burchard. Visualizing the structure of the World Wide Web in 3D hyperbolic space. In *Proceedings of the first symposium on Virtual reality modeling language*, pages 33–38, New York, NY, USA, 1995. ACM.

[Mor32] J. L. Moreno. *Application of the Group Method to Classification.* National Committee on Prisons and Prison Labor, New York, 1932.

[Ott83] R. Otten. Efficient floorplan optimization. In *Proc. Int. Conf. on Computer Design*, pages 499–502, Piscataway, US, 1983. IEEE Press.

[Pau93] F. N. Paulisch. *The Design of an Extendible Graph Editor.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[PT00] A. Papakostas and I. G. Tollis. Efficient orthogonal drawings of high degree graphs. *Algorithmica*, 26(1):100 – 125, 2000.

[Pur97] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, London, UK, 1997.

[QE01] A. Quigley and P. Eades. A multilevel algorithm for force-directed graph drawing. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–80. Springer-Verlag, London, UK, 2001.

[QZW07] H. Qu, H. Zhou, and Y. Wu. Controllable and progressive edge clustering for large networks. In *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 399–404. Springer-Verlag, London, UK, 2007.

[Rai04] M. Raitner. Maintaining hierarchical graph views for dynamic graphs. Technical Report MIP-0403, University of Passau, Passau, Germany, 2004.

[RBM01] S. Roy, S. Bandyopadhyay, and U. Maulik. Proof regarding the NP-completeness of the unweighted complex-triangle elimination (CTE) problem for general adjacency graphs. *IEE Proceedings. Computers and Digital Techniques*, 148(6):238–244, 2001.

[Rea87] R.C. Read. A new method for drawing a graph given the cyclic order of the edges at each vertex. *Congr. Numer.*, 56:31–44, 1987.

[RMB04] S. Roy, U. Maulik, and S. Bandyopadhyay. Search strategies to solve the complex-triangle elimination (CTE) problem. *IETE Journal of Research*, 50(6):409–423, 2004.

[RMC91] G. G. Robertson, , J. D. Mackinlay, and S. K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194, New York, NY, USA, 1991. ACM.

[RMN09] Md. S. Rahman, K. Miura, and T. Nishizeki. Octagonal drawings of plane graphs with prescribed face areas. *Comput. Geom. Theory Appl.*, 42(3):214–230, 2009.

[RNG04] Md. S. Rahman, T. Nishizeki, and S. Ghosh. Rectangular drawings of planar graphs. *J. Algorithms*, 50(1):62 – 78, 2004.

[RS86] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of treewidth. *Journal of Algorithms*, 7(3):309–322, September 1986.

[Ruc78] J. Ruch. Interactive space layout: A graph theoretical approach. In *Proceedings of the 15th conference on Design automation*, pages 152–157, Piscataway, NJ, USA, 1978. IEEE Press.

[Sar93] M. Sarrafzadeh. Transforming an arbitrary floorplan into a sliceable one. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 386–389, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Sch90] W. Schnyder. Embedding planar graphs on the grid. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 138–148, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[SH84] R. N. Shepard and S. Hurwitz. Upward direction, mental rotation, and discrimination of left and right turns in maps. *Cognition*, 108:161–193, 1984.

[Shi76] Y. Shiloach. *Arrangements of Planar Graphs on the Planar Lattice*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1976.

[Shn92] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

[SKB88] S. Sur-Kolay and B. B. Bhattacharya. Inherent nonslicibility of rectangular duals in vlsi floorplanning. In *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 88–107, London, UK, 1988. Springer-Verlag.

[SKB91] S. Sur-Kolay and B. B. Bhattacharya. On the family of inherently nonslicible floorplans in VLSI layout design. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 2850– 2853, Los Alamitos, CA, US, 1991. IEEE Computer Society Press.

[ST81] M. V. Swany and N. K. Thulasiraman. *Graphs, Networks and Algorithms*. Wiley Interscience, New York, 1981.

[Ste06] P. Steadman. Why are most buildings rectangular? *Architectural Research Quarterly*, 10(2):119–130, 2006.

[Sto83] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Inf. Control*, 57(2-3):91–101, 1983.

[Sto84] J. A. Storer. On minimal node-cost planar embeddings. *Networks*, 14:181–212, 1984.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man and Cybernetics*, 11(2):109–125, 1981.

[SY93] Y. Sun and K.-H. Yeap. Edge covering of complex triangles in rectangular dual floorplanning. *Journal of Circuits, Systems, and Computers*, 3(3):721–731, 1993.

[SY99] S. M. Sait and H. Youssef. *VLSI Physical Design Automation: Theory and Practice*. World Scientific Publishing Company, 1st edition, 1999.

[Tam87] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[TDBB88] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1):61–79, 1988.

[Tho80] C. Thomassen. Planarity and duality of finite and infinite planar graphs. *J. Combinatorial Theory, Series B*, 29:244 – 271, 1980.

[Tho92] C. Thomassen. Plane representations of graphs. *Combin. Probab. Comput.*, 1:336 – 342, 1992.

[TKS86] S. Tsukiyama, K. Koike, and J. Shirakawa. An algorithm to eliminate all complex triangles in a maximal planar graph for use in VLSI floor-plan. In *IEEE Proc. Int. Symp. On Circuits and Systems*, pages 321–324, Los Alamitos, CA, USA, 1986. IEEE Computer Society.

[TMSS89] S. Tsukiyama, T. Maruyama, S. Shinoda, and I. Shirakawa. A condition for a maximal planar graph to have a unique rectangulardual and its application to VLSI floor-plan. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 931–934, Los Alamitos, CA, US, 1989. IEEE Computer Society Press.

[Tob04] W. Tobler. Thirty five years of computer cartograms. *Annals of the Association of American Geographers*, 94(1):58–73, 2004.

[TST89] K. Tani, I. Shirakawa, and S. Tsukiyama. An algorithm to enumerate all rectangular dual graphs. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(3):34 – 46, 1989.

[Tut56] W. T. Tutte. A theorem on planar graphs. *Trans. Amer. Math. Soc.*, 82:99–116, 1956.

[Tut60a] W. T. Tutte. Convex representations of graphs. *Proc. London Math. Soc.*, s3-10(1):304 – 320, 1960.

[Tut60b] W. T. Tutte. How to Draw a Graph. *Proc. London Math. Soc.*, s3-13(1):743 – 768, 1960.

[Ung53] P. Ungar. On diagrams representing maps. *J. London Math. Soc.*, 28:336–342, 1953.

[Val81] L. Valiant. Universality considerations in VLSI circuits. *IEEE Transactions on Computers*, C-30(2):135 – 140, 1981.

[Vas02] A. Vassalli. Disegno automatico di un grafo planare mediante la costruzione del proprio duale rettangolare. Master's thesis, University of Genoa, Department of Computer Science, Genoa, Italy, 2002.

[vKS07] Marc van Kreveld and B. Speckmann. On rectangular cartograms. *Comput. Geom. Theory Appl.*, 37(3):175–187, 2007.

[Wal01] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 171–182. Springer-Verlag, London, UK, 2001.

[Win02] A. Winter. Exchanging graphs with GXL. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 54–58. Springer-Verlag, London, UK, 2002.

[WS89] D. F. Wong and P. S. Sakhamuri. Efficient floorplan area optimization. In *Proceedings of the 26th ACM/IEEE conference on Design automation*, pages 586–589, New York, NY, USA, 1989. ACM.

[YS91] K. H. Yeap and M. Sarrafzadeh. A theorem of sliceability. In *Proceedings of the Second Great Lakes Computer Science Conference*, Kalamazoo, US, 1991. Western Michigan University.

[YS95] G. K. H. Yeap and M. Sarrafzadeh. Sliceable floorplanning by graph dualization. *SIAM J. Discret. Math.*, 8(2):258–280, 1995.