L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Livaniaina Hary Rakotomalala

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Adding a new dimension to the Upward Drawing

TITRE DE LA THÈSE / TITLE OF THESIS

Nejib Zaguia

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

G. V. Jourdan

E. Kranakis

Gary W. Slater

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Adding a new dimension to the Upward Drawing

by

Livaniaina Hary Rakotomalala

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University Of Ottawa

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-14923-X
Our file Notre référence
ISBN: 0-494-14923-X

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

© Livaniaina H. Rakotomalala, Ottawa, Canada, 2005

The undersigned hereby recommend to
The faculty of Graduate Studies and Research
acceptance of this thesis

Adding a new dimension to the Upward Drawing

Submitted by
Livaniaina Hary Rakotomalala

In partial fulfillment of
the requirements for the degree of
Master of Computer Science

____

Nejib Zaguia, Ph.D.
(Thesis Supervisor)

To my parents Arthur and Lala,
my sisters Lalaina and Vola
and my brother Mamy.

Contents

List of Figures vii
List of Tables ix
List of Algorithms x
Abstract xi

1. Introduction 1
1.1. Data Visualization 1
1.2. Data as an Ordered Set 1
1.3. Upward Drawing of ordered sets 2
1.4. Parameters of a good drawing 3
1.4.1. Edges Crossing and Upward Planarity. 3
1.4.2. Area Used 4
1.4.3. Symmetry, Colinearity and others parameters. 4
1.5. Difficulties and Solutions 6
1.5.1. Aesthetics versus Algorithms Efficiency 6
1.5.2. General Tentative Solutions 6
1.6. Contributions 7
1.6.1. Good Overall View versus Easy Relation Identification 7
1.6.2. Drawing using Chain Decomposition of a Linear Extension 8
1.6.3. Main Contributions 9
1.7. Thesis Outline 10
2. Ordered Set Basics 12
2.1. Preliminaries: Definitions and Notations 12
2.1.1. Ordered sets and Covering Relation 12
2.1.2. Hasse Diagram 13
2.2. Decomposition Scheme of ordered sets 15
2.2.1. Chains and Dilworth's Theorem 15
2.2.2. Linear Extension and Jump 16
3. Review of Existing Approaches 19
3.1. Hasse-Diagram versus Digraph Approach 19
3.2. Conventional ideas of upward drawing 20
3.2.1. The "Hierarchical" method 20
3.2.2. The "Force Directed" method 22
3.2.3. The "Divide-And-Conquer" method 23
3.2.4. Related results on ordered sets 24
3.2.5. Results for some classes of graph 24
3.3. Existing software 27
4. The LR-Drawing 29
4.1. Drawing Assumptions and Criteria Hypothesis 30
4.2. The "Bottom-to-top" and "Left-to-Right" Idea 31

4.2.1. Definition of LR-Drawing 32
4.2.2. A new aesthetic metric: the "Flow of Comparability" 33
4.2.3. Maximization of vertical relations by "Greedy Linear Extension". 35
4.3. Drawing Algorithm 39
4.3.1. Pseudo-Code. 39
4.3.2. Time Complexity. 40
4.3.3. Examples of execution 41
4.3.4. Problematic and Need for a Parameterization 43
5. N-Free ordered sets 44
5.1. Motivations 44
5.2. The N-Free Subclass 45
5.3. The N-Free Converter methods 46
5.3.1. The "naïve" method 46
5.3.2. The "space efficient" method 47
5.3.3. The proof of correctness of the space-efficient algorithm 49
5.4. Properties and Characteristics 52
5.4.1. Jump optimality 52
5.4.2. The X-cycle Free Assumption 52
5.4.3. "Existing LR edges" Lemma 54
5.4.4. "Unicity of Links from Top and Bottom" Lemma 56
5.5. The Chains Interchange technique 58
5.5.1. The technique of interchanging two consecutive chains 59
6. LR Drawing on N-Free 62
6.1. The Interchange-Chains Algorithm 62
6.1.1. Execution samples 63
6.2. Local Reorganization around a Chain 65
6.2.1. The left neighbors chains restructuring 67
6.2.2. The right neighbors chains restructuring 70
6.2.3. The local picture enhancement algorithm 73
6.2.4. Crossings and Limitations 74
6.3. Overall View Improvement 79
6.3.1. Difficulties and Problematic 79
6.3.2. The global picture enhancement algorithm 82
6.4. The N-Free drawing algorithm 83
6.4.1. Example of Execution and Comparisons 85
6.4.2. Planarity, Symmetry and Space used. 87
7. Implementation 90
7.1. Testing and ordered sets database 90
7.1.1. Graph database to ordered set database. 91
7.1.2. Extraction of N-Free ordered sets. 92
7.1.3. Particular Testing 92
7.2. Algorithm Implementation and Data Structure 92
7.2.1. Matrix of Comparability vs Matrix of Coverability. 92
7.2.2. The java classes Architecture 94
7.2.3. The N-free ordered set viewer: The LR-Flow Order. 95

7.3. "Time versus Space" Discussion 98
8. Conclusion 101
8.1. Achievements 101
8.2. Future work 102

Bibliography 104

Appendix 1: The Greedy Linear Extension Algorithm in Java 107
Appendix 2: The Implementation of the LR-Drawing algorithm in Java 110
Appendix 3: The Interchange-chains Algorithm in Java 112
Appendix 4: The Arranging-Chain Algorithm in Java 114
Appendix 5: The Arranging-All-Chains Algorithm in Java 116
Appendix 6: LR-Drawing pictures versus Existing Systems. 117

List of Figures

Figure 1.1: An example of an ordered set modeling a set of data 2
Figure 1.2: Example of drawing dilemma between Upward versus Planarity 3
Figure 1.3: Symmetry and Colinearity of a Hasse Diagram 5
Figure 2.1: Samples of an Hasse Diagram 14
Figure 2.2: Example of a disjoint sum and a linear sum.. 16
Figure 2.3: The N ordered set and its linear extensions... 17
Figure 4.1: Example of Hasse Diagram with polyline and curve edges 30
Figure 4.2: An example of pseudo grid-drawing and the rectangle area associated... 31
Figure 4.3: Example of "flow of Comparability" differences between a symmetric drawing and a LR Drawing. 34
Figure 4.4: The LR Drawing's flow of comparability 34
Figure 4.5: The end of a rise in a Greedy Linear Extension for a ordered set P......... 37
Figure 4.6: Some execution scenarios of the greedy linear extension algorithm 38
Figure 4.7: Samples of execution of the LR-Drawing on different linear extensions 42
Figure 5.1: The N ordered set. 45
Figure 5.2: Example of diagrams containing N. 45
Figure 5.3: Example of N-Free ordered set 46
Figure 5.4: Execution of the space efficient N-Free converting algorithm. 48
Figure 5.5: Links from "Top" for N-free 56
Figure 5.6: Links to "Bottom" for N-free 56
Figure 5.7: The forbidden links between two chains 56
Figure 5.8: Unicity of Link towards "Bottom" of any chain 58
Figure 5.9: Unicity of Link from "Top" of any chain.. 58
Figure 5.10: The chains interchange between two consecutive chains 59
Figure 5.11: The remaining possible configuration between two consecutive chains. 60
Figure 5.12: An example of configuration where a chain interchanging is directly undoable. 60

Figure 6.1: Samples of the Chains Interchange highlighting the 4 different cases configuration between two consecutive chains.. 64
Figure 6.2: The proposed chain neighborhood after the local improvement.............. 66
Figure 6.3: The worst case configuration of the chain neighborhood 66
Figure 6.4: The basic technique of improving the left-links of a chain...................... 68
Figure 6.5: An execution sample of the Arrange-Left-Chains algorithm.................. 70
Figure 6.6: The crossing cases for the left-neighbors chains..................................... 74
Figure 6.7: The crossing cases for the right-neighbors chains................................... 74
Figure 6.8: The dilemma on solving the crossing around a particular chain.............. 77
Figure 6.9: The worst case configuration of the chain neighborhood 78
Figure 6.10: Example of tentative picture improvement by successive arrangement of each chain.. 81
Figure 6.11: A sample execution of the Arrange-All-Chains algorithm. 86
Figure 6.12: Sample of additional operations improving the picture 87
Figure 7.1: Sample of comparability and coverability matrix of a ordered set P....... 92
Figure 7.2: The ordered set Viewer GUI 95
Figure 7.3: An execution sample of the ordered set viewer. 97
Figure 7.4: The set of Upper-Covering and Lower-Covering into an HashTable...... 99

List of Tables

Table 3.1: Time complexity of planarity testing and crossing minimization on Graphs 25
Table 3.2: Existing upper bounds and lower bounds on the area of upward planar drawing. 26
Table 3.3: Time Complexity of various Upward Drawing of selected graph............. 26
Table 7.1: Template of the principal class Ordered Set. 94
Table 7.2: The ratio time/space on the initial drawing of P.. 99
Table 7.3: The ratio time/space on the drawing improvement after the initial picture. 100

List of Algorithms

Algorithm 4.1: The generation of a "greedy" linear extension................................. 36

Algorithm 4.2: The General ordered set Drawing Algorithm using LR strategy....... 40

Algorithm 5.1: The naïve algorithm to convert an ordered set into N-Free. 47

Algorithm 5.2: The space efficient algorithm to convert an ordered set into N-Free. 48

Algorithm 6.1: The Chain-Interchanging algorithm of two consecutive chains. 63

Algorithm 6.2: The Arranging-Left-Chains algorithm... 69

Algorithm 6.3: The Arranging-Right-Chains algorithm... 72

Algorithm 6.4: The Arranging-Chain algorithm.. 73

Algorithm 6.5: The Arranging-all-Chains algorithm.. 83

Algorithm 6.6: The LR-drawing algorithm of N-Free X-cycle free ordered sets....... 84

Algorithm 7.1: The Transitive-Reduction algorithm in Java.................................... 93

Algorithm 7.2: The CoverabilityMatrix-To-HashTable preprocess in Java. 98

Abstract

In this thesis, we principally investigated the visualization of data that can be represented by an ordered set by means of its upward drawing (Hasse Diagrams). Difficulties are such that "Upward Planarity Testing" is NP-complete [GT95] and "Upward Crossing Minimization" is NP-Hard [Lin94]. Also, the "drawing area" requirement of upward planar directed graphs using straight-line edges on a grid has an exponential lower bound [DTT92].

We present a new approach of drawing Ordered Sets which is a modified version of Hasse Diagrams. The new restriction is that all existing edges are only directed from "Bottom to Top" (upward) and from "Left to Right" (LR). This new dimension is motivated by the rise of a flow of comparability between elements and thus produces readability improvement. The idea is achieved by using special types of decompositions of the ordered set P that will serve as the underlying structure for the drawing. We propose an $O(n * width(P))$ algorithm which satisfies this new requirement.

Furthermore, this new idea is studied for the subclass of N-free ordered sets. We present some techniques of improving the readability by locally reducing crossings around some component of the drawing. We also present a polynomial algorithm which brings an enhancement of the overall picture. The resulting rectangle used has an $O(n^2)$ area. Additionally, to have an approximation of the general ordered set drawing, we present a space-efficient algorithm to convert any ordered set into N-Free.

Acknowledgement

I extend my sincere gratitude and appreciation to the people who made this Master Thesis possible. My special thanks to my supervisor Dr. Nejib Zaguia for giving me his unrestrained advice and encouragement. His valuable suggestions during the course of work are greatly acknowledged. This work is as much his creation as it is mine.

I would like to thank Dr Guy Vincent-Jourdan for his constructive criticism and his valuable time that he was willing to sacrifice. Merci infiniment.

I am indebted to all reviewers who have gone through my work and helped me fill in the knowledge gaps. I would also like to pass my sincere thanks to my laboratory colleague Mustafa Al-Hashem, Yedneke Asfaw and my friends for motivating and encouraging me in many ways through our university years. A great appreciation also goes to Jean Moussi who offered me his friendship ever since I came to Canada.

Finally, I also want to thank my family for their unconditional support. I would like to share this moment of happiness with them.

This thesis was financially supported by the Ontario Graduate Studies Scholarship (O.G.S.), the admission Scholarship and the Excellency Scholarship of the University Of Ottawa and my supervisor Dr Nejib Zaguia.

Sincerely,

Livaniaina H. Rakotomalala.

Chapter 1

Introduction

1.1. Data Visualization

As more data is stored and explored, a good visualization of such data provides a much better and faster understanding of its structure. As the complexity of data increases, the better visualization quality is required. There are many ways to visually symbolize data; however the most frequently used method is to represent data by means of *graphs*. Theoretically, data items are represented by vertices and the relations among the data are represented by edges. Graph theory is widely used, especially in computer science. For instance, graphs are frequently used to represent classical data-flow diagrams in a software development process, Petri-nets in real-time systems, classes diagram of an object oriented programming language, and entity-relationships in a database. Several other fields also employ graphs: circuit schematics in electronics, network paths in telecommunications, molecular diagrams in chemistry and organization charts in management.

1.2. Data as an Ordered Set

There are various types of relations between data items. Among them, we can distinguish special categories called Hierarchies, *Partial Orders or* simply *Orders.* An order is a relation which is *reflexive*, *antisymmetric* and *transitive.* A set of data respecting the rules of an order among its elements is called an *ordered set.* Simple examples of ordered sets include subsets of a given set ordered by inclusion, natural numbers ordered by divisibility, or strings ordered lexicographically. More practical and application based examples are java classes ordered by the inheritance property, tasks and jobs ordered by precedence in scheduling, or university courses ordered by prerequisite (*Figure 1.1(a)*).

1.3. Upward Drawing of ordered sets

The standard method of graphically representing an ordered set is by way of what is called a *Hasse Diagram* or *upward drawing*.

The notion of drawing an ordered set is via the concept of a covering relation. An element x covers an element y if x is greater than y and there are no other elements between x and y. By exploiting the transitivity property of the ordered set, we can eliminate the transitive edges and represent only the covering edges without losing any information about the ordered set.

The *covering graph* of an ordered set P is the graph where the edges represent the covering relations of P. The *upward drawing*, or *Hasse diagram of an ordered set P is* defined as the covering graph of P drawn in the plane such that the elements are represented by small empty circles and covering relations are shown as lines going upward. The element x is lower than the element y whenever x is covered by y. Since all covering relations point upward, the arrowheads are eliminated. For instance, the *Figure 1.1(a)* shows a set of university courses ordered by the prerequisites constraints and the *Figure 1.1(b)* is the ordered set modeling it through its upward drawing.

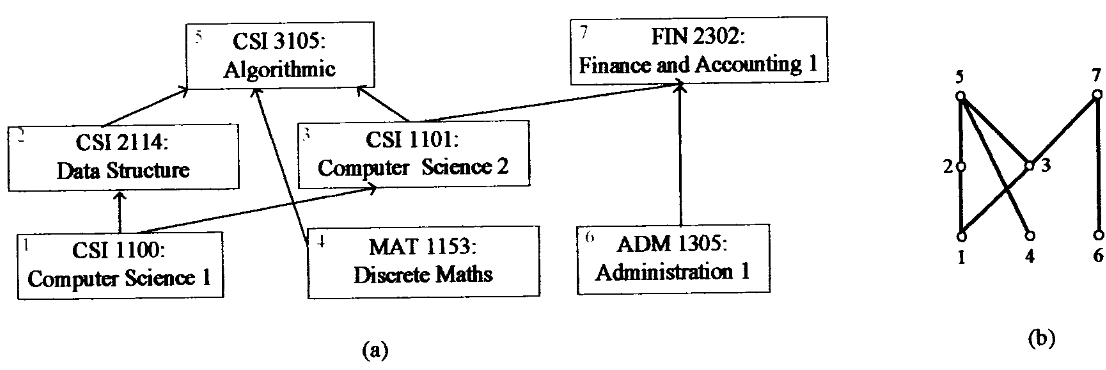


Figure 1.1: An example of an ordered set modeling a set of data

- In the domain of ordered data visualization, a good diagram is a helpful resource for understanding the hierarchies between the data.

- In the domain of ordered sets, a good diagram can be a real asset in helping understand theories and in proving theorems.

A valid diagram can be easily drawn; however, two important criteria must be respected: *drawing aesthetics* and *algorithmic efficiency*. The union of both criteria provides the main difficulty in upward drawing.

1.4. Parameters of a good drawing

"What is pleasant to the eye?" is a question that could define the aesthetic parameters of a good drawing. However, for two elements a, b in an ordered set P, deciding easily whether or not a is less than b, is the most important feature of any acceptable drawing. The upward drawing is meant to do just that.

Although the definition of a "good" drawing depends, in many cases, on the application and its context of use, several standard criteria are crucial for a readable drawing. For instance, one indicates the concepts of crossing, symmetry, angular resolution, space used by the drawing, and curves on edges. The influences of these properties on comprehension and their adequate visualization in respective drawings have been systematically investigated by many researchers.

1.4.1. Edges Crossing and Upward Planarity.

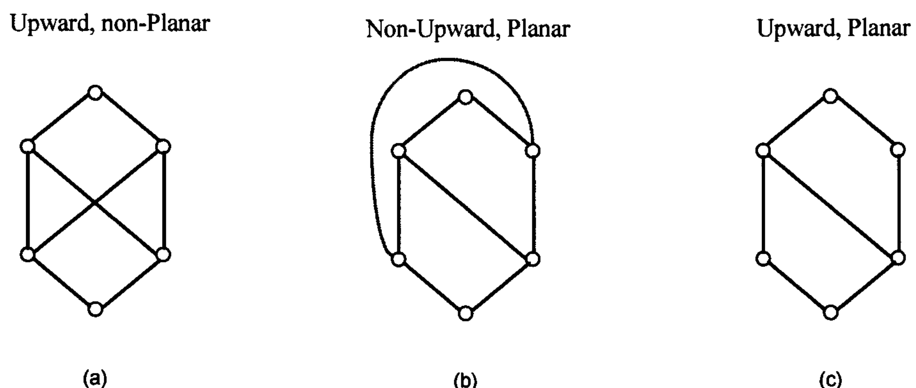


Figure 1.2: Example of drawing dilemma between Upward versus Planarity

Among these aesthetic parameters, the one which has received the most attention among researchers during the last few years is probably the concept of "edge

crossing". A drawing with fewer crossings is more readable as it produces less confusion to the reader who wants to see the links between the various vertices. Technically, a drawing which does not produce any crossing is described as "planar". *Tarjan-Hopcroft* [HT74] presents a linear algorithm to test the planarity of a graph. Testing if a directed graph is upward can be computed in a polynomial time. Unfortunately, the combination of "Upward and Planarity" testing is an NP-Hard problem and was proven to be NP-complete by *Garg and Tamassia* [GT95] in the mid-nineties.

For certain types of ordered sets, it is impossible to produce an upward planar drawing (*see Figure 1.2 (a)*). Thus, instead of solving the Upward-Planarity issue, certain researchers have widened the problem and have chosen to reduce the number of crossings to a minimum. However, the problem of reducing the number of crossings for the upward drawing was also proven to be NP-Hard [Lin94].

1.4.2. Area Used.

One of the parameters which we also take into account is the space occupied by the drawing. For this, we assume that the drawing is fulfilled on a grid. The minimization of the surface used by the drawing is essential because the space used by the screen has to be minimized. Formally, the area is defined by one of two characteristics: either the smallest convex polygon which covers the drawing called "Convex-Hull", or the smallest rectangle which can embed it. If polyline edges are allowed into the upward drawing, then good results are known as drawing algorithms which satisfy planarity and polynomial space exists. Unfortunately, the area requirement is "exponential" for planar upward drawings using straight-line edges. Indeed, *Di Battista et al* showed that there is a class of planar acyclic digraphs that requires exponential area in any planar straight-line upward drawing [DETT99].

1.4.3. Symmetry, Colinearity and others parameters.

An important parameter often mentioned is the symmetry of the drawing. A formal way to characterize symmetric Hasse Diagram is by use of its chains length

and their positions in the diagram [Far97]. For instance, the *Figure 1.3* illustrates three different diagrams of the same ordered set where version *(a)* is symmetric but *(b)* is not.

A frequent problem which can also occur is colinearity, especially when the drawing is fulfilled on a grid. For a Hasse diagram, an element should not overlap any edge to which that element is not incidental. Indeed, such a superposition can lead to a false interpretation of the links which can exist between the various implied vertices. As an example, *Figure 1.3* illustrates such a colinearity issue which generates a new edge which was previously non-existent, and thus a new comparability relation is created in our ordered set. (Here for example, the element 3 becomes comparable to 2).

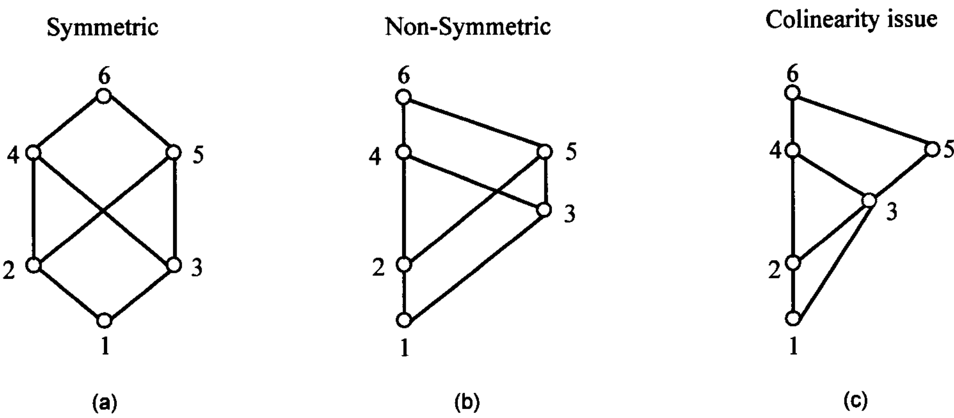


Figure 1.3: Symmetry and Colinearity of a Hasse Diagram

Certain parameters, which are more graphic, are also mentioned, such as the maximization of the smallest angle between two incidental edges from the same vertex, the minimization of variance of edges length, the minimization of maximum length of an edge.

1.5. Difficulties and Solutions

1.5.1. Aesthetics versus Algorithms Efficiency

The aim is not only to develop an algorithm which produces a diagram which satisfies all the aesthetic conditions previously mentioned. The algorithm must also be implemented in an efficient manner. It is known that:

- Aesthetic parameters often conflict with each other. Hence, it is difficult to simultaneously assure all of them. Consequently, some parameters will be sacrificed in favor of others. For instance, the criterion of minimizing the maximum length of edges may be sacrificed to increase symmetry, or the convex-hull space at the expense of crossing minimization. The decision regarding the assignment of priorities between the different parameters is generally made on an application basis.
- Even if there are no conflicts between the chosen aesthetic criteria that the drawing must satisfy, finding an efficient solution is algorithmically difficult.

1.5.2. General Tentative Solutions

Generally, testing aesthetic criteria leads to the drawing algorithm. Indeed, the technique used to test an aesthetic parameter induces the drawing algorithm which satisfies this same parameter. For instance, the Upward-Planarity testing algorithm can be modified to compute an upward planar embedding of a graph if eventually this graph is planar.

Several attempts to produce acceptable solutions were undertaken with respect to each specific aesthetic criterion:

- Regarding upward planarity and crossing minimization, their NP completeness enforces the heuristic approach to both problems. Various techniques, such as parametrization, were used to solve the issue. Efficient algorithms are only known for certain subclasses of graphs. For instance, *Bertolazzi et al.* recently presented a linear time algorithm to draw upward planar single-source digraphs [BDBD02]. We will discuss these solutions and these

heuristics in more detail in the next chapter where a survey of existing literature is presented.

- Colinearity is often considered as a shortcoming to the drawing instead of a conceptual error into the algorithm. Thus, the presence of colinearity is more of a measure of the algorithm's robustness than its validity. To remedy colinearity, an approach was offered by researchers from the University of Ottawa involving ordered set Explorer software [Jou95]. In this approach, instead of producing a complete drawing, edges were eliminated and the user can choose to view them or not by interacting with the software. This idea, known as "Edge elimination", is part of an important method called "Information hiding" which is well recognized in "huge" graphs drawing.

Usually, the aesthetic criteria being targeted by an upward drawing algorithm are explicitly stated. Even if differences exist between several drawing algorithms, they are mainly inspired from similar conventional ideas. Among the most famous approaches, we can distinguish the Hierarchical layering inspired from the Sugiyama approach [STT81]. Different conventional ideas, such as the "Force Directed" method and the "Divide-and-Conquer" strategy also exist. The majority of visualization software implements algorithms that were inspired from these ideas. Due to the concurrency of the different aesthetic parameters, the user is often given the option of selecting, by ways of interacting with the software, which parameters are to be used.

1.6. Contributions

1.6.1. Good Overall View versus Easy Relation Identification

Conceptually, while these conventional ideas are thoroughly different with respect with their approach to the drawing process, they have a common aim of producing a pleasant overall view of the picture. However, practical aspects of the drawing are usually forgotten by these ideas. Indeed, instead of producing a beautiful global view (e.g. a symmetric picture), the majority of practical applications require a drawing that provides a good indication of the relations between data components.

This is especially true for software visualization of ordered sets as the most important functionality that a user requires it that of being able to quickly visualize the comparability between the elements.

In this thesis, we investigate a new approach to the drawing problem which reflects this concept of easy identification of relations between elements. We are proposing what we call the *LR-Drawing*. It is a modified version of the upward drawing concept, a new way to visualize ordered sets. Our approach is to use a chain decomposition of ordered sets as the underlying structure to position the vertices of the order. The main goal is to suppress more edges, without a loss of information.

1.6.2. Drawing using Chain Decomposition of a Linear Extension

A subset of an ordered set P is a chain (an antichain) if each pair of elements x, y is comparable (noncomparable). Let $P = C_1 \cup C_2 \cup \ldots \cup C_n$ be a chain decomposition of P. By placing each chain C_i on a different vertical line, it becomes much easier to spot lots of comparabilities: the ones within the same chain. This simple idea allows a better visualization for some classes of ordered sets, especially classes where the number of chains in the decomposition is small. However, there is a difficulty (similar to the upward drawing) in easily spotting the ordering relations between pairs of vertices when they are in distinct chains. In fact, the covering relations between different chains could be either from left to right or from right to left. That is, if x covers y then x is higher than y but x could be either on the left or on the right of y.

A total ordering L of the elements of P is a *linear extension* if L preserves the relations of P. Every linear extension L of P induces a natural chain decomposition of P, since L is the linear sum $C_1 \oplus C_2 \oplus \ldots \oplus C_n$ of chains $C_1, C_2, \ldots, C_n$ of P such that, in P, for each $i=1,2, \ldots ,n-1$, the maximum element of C_i is non comparable to the least element of C_{i+1} in P.

Instead of drawing the order using any chain decomposition as the underlying structure, we consider the chain decompositions derived from a linear extension. This type of upward drawings will only have covering relations from left to right. Our new

approach will still satisfy the conventional upward drawing (Bottom to Top) and will insert a new "implicit information" which is "from Left to Right". Hence, the drawing will display a certain "flow" with respect to those directions thereby giving the viewer the ability to easily trace the path from one element to another. It is what we call the *LR-drawing* which stands for *"Left to Right upward drawing"*.

The results of our investigation of the LR upward display have been very promising. For instance, by using an algorithm based on a modified version of the interchanging chain techniques for N-free Orders (this technique allows exchanging consecutive chains without increasing the number of chains), the LR-drawing applied on some classes of N-free ordered sets gave very interesting results on terms of the space required and the number of edge crossings.

1.6.3. Main Contributions

The main contributions of this thesis are the following:

1. We propose a new type of drawing called the "LR Drawing" (abbreviated from Left-to-Right). All existing edges will only have Bottom-To-Top or Left-To-Right directions. Consequently, it produces a "Flow of Comparability" between elements. Thus, the insertion of this extra *implicit information* brings a *new dimension* to the Upward Drawing.
2. We give an $O(n^2)$ algorithm, and more precisely $O(n * width(P))$, to draw any ordered sets diagram which respects this new aesthetic dimension.
3. The LR Drawing is deeply studied on N-Free ordered set with respect to Crossing-Minimization and Area Used. We propose some techniques to improve the N-free class drawing by some crossing reduction methods. These techniques can be used interactively on some components of the drawing; or they can be used systematically to bring a global improvement to the drawing. The rectangle embedding the N-free picture has an $O(n^2)$ area on the number of elements.
4. We present an efficient algorithm, in terms of additional element expenses, that converts any general ordered set into N-Free. This algorithm is especially useful to have an approximation of general ordered set drawing.

5. We develop "an ordered set drawer" software, called the *LR-FlowOrder*, which incorporates our LR-Drawing ideas. It implements all algorithms which were theoretically investigated and automatically draws the ordered set. The software also allows us to perform comparisons against existing software pictures. The *LR-FlowOrder* can be exploited as a tool for some theoretical proofs on ordered sets (especially N-Free). Its functionalities can also be exported into visualization software.

1.7. Thesis Outline

The organization of this thesis is as follows:

In Chapter 2, we will develop the formal specifications which define the theory of ordered sets and its principal components. This will also provide the terminology that will be used in the remainder of this thesis. Assumptions and hypotheses will also be introduced. Useful properties and theorems will be also presented.

In chapter 3, we review some conventional ideas that gave rise to the actual known algorithm of Upward Drawing. We will list certain well known software which benefited from these inspirations. We will also discuss different methods which solve certain esthetic restrictions, including some heuristics to solve Upward Planarity, efficient algorithms that attempt to give good drawing for some classes of graph. Finally, we also list some bounds on the computational complexity and the area used of the upward drawing of certain classes of graphs.

The next 3 chapters form the core of this thesis.

In Chapter 4, the real motivations which give rise to our approach and the effectiveness of our new drawing technique called the LR-Drawing are discussed. Also in this chapter, our algorithm is presented.

Chapter 5 begins with a discussion about one subclass of ordered set called N-free. We propose a space-efficient algorithm which converts a general ordered set into this class. Special properties of this subclass which are useful to refine the

drawing are also presented. We also introduce in this chapter a technique which will later allow us to manipulate the drawing.

Chapter 6 will mainly discuss the drawing of the N-free ordered set. Firstly, we present different techniques that bring local and global enhancement to the drawing. We also list the limitations of our techniques. Finally, we conclude the chapter by listing the drawing algorithm of that class.

Chapter 7 will discuss the implementation of the algorithms presented in the two preceding chapters. The first part will elaborate on the creation of an ordered set database which is required to test our algorithms. Then, we will study the implementation of the algorithms in the Java programming language, as well as the choice of the data structure used. Lastly, we will discuss the classic compromise of "time versus space" or the dilemma between the choice of data structure and the time profit/waste which is concomitant with that choice

We will give our conclusion in Chapter 8 as well as the possible various extensions of our work.

Chapter 2

Ordered Set Basics

In this chapter, we list all necessary definitions about ordered sets. We also familiarize the reader with the notations to be used throughout this thesis and present all required theorems.

2.1. Preliminaries: Definitions and Notations

2.1.1. Ordered sets and Covering Relation

Definition 2.1. Ordered Set

An *ordered set* $P = (X, \leq)$ is a pair consisting of a non-empty set X and a binary relation $\leq$ on X, satisfying *reflexivity*, *antisymmetry* and *transitivity*.

For two elements $x, y \in P$, we say that x is *comparable* to y, denoted $x \sim y$, if $x \leq y$ or $y \leq x$ in P. We will interchangeably use the signs "≤", "≥" and "~" to denote the comparability. Conversely, x and y are called *incomparable*, and we note $x \parallel y$, if neither $x \leq y$ nor $y \leq x$ holds. We say that P has a bottom element called *Bottom(P)*, if for all $x \in P, Bottom(P) \leq P$. Dually, we say that P has a top element denoted *Top(P)*, if for all $x \in P$ $, x \leq Top(P)$.The element a is a "*maximal*" element of P if $a \leq x$ and $x \in P$ implies that a = x. A "*minimal*" element of P is defined dually by just reversing the order.

The *predecessors* of an element x, denoted $\downarrow x$, is a set where all elements are smaller than x. Formally, $\downarrow x = \{y \in P \mid y \leq x \wedge y \neq x\}$. Conversely, the *successors* of x, denoted $\uparrow x$, is a set where all elements are greater than x. Formally, $\uparrow x = \{y \in P \mid y \geq x \wedge y \neq x\}$

Definition 2.2. The covering relation

The *covering relation* of an ordered set P is the transitive reflexive reduction of P. An element y of P *covers* another element x, and we note $x \prec y$ or $y \succ x$, if there exists no third element z in the ordered set for which $x < z < y$.

The element y is called an "*upper cover*" of x (or y is an *immediate successor* of x) and x a "*lower cover*" of y (x is an *immediate predecessor* of y). The sets $\uparrow^{im} x$ and $\downarrow^{im} x$ respectively denote the set of upper-cover and the set of lower cover of x.

2.1.2. Hasse Diagram

The first natural way to represent an ordered set is to draw a graph where the vertices represent the elements and edges represent the relations of comparability. It is called a *comparability graph.* To reflect the hierarchy between elements, the graph can be transformed into a directed one by putting arrows where the greater element is the one targeted. By putting the edges in an upward direction, we can explicitly delete the arrows and keep usual edges. It will then implicitly tell that between two linked elements, the one in an upper position is greater than an element in a lower position. But still, this kind of graph is still carrying extra information. By exploiting the transitivity property of the order, we can eliminate the transitive edges and put only the covering edges.

Definition 2.3: The Hasse Diagram or Upward Drawing

A *Hasse Diagram*, also called *upward drawing* of P, is a graphical rendering of P displayed via the covering relation with an implied upward orientation. Each element of the ordered set is represented by a *small empty circle*, and *line segments* are drawn between these points according to the following 3 rules:

1. If $x \leq y$ in P, then the point corresponding to x appears lower in the drawing than the point corresponding to y.
2. The line segment between the points corresponding to any two elements x and y of the ordered set is included in the drawing if only if $x \prec y$ or $y \prec x$.
3. Line segments are strictly upward.

Graphically,

- Each element $e \in P$ is associated to a point $f(e)$ of the Euclidian plane R^2, depicted by the small circle with centre at $f(e)$. (This process is usually called the "labeling").
- For each covering pair $x \prec y$ in *P*, a line segment will join the circle at $f(x)$ to the circle at $f(y)$.
- For each covering pair $x \prec y$ in *P*, $f(x)$ is lower than $f(y)$. In a standard Cartesian coordinates, $f(y)$ has a strictly greater second coordinate.

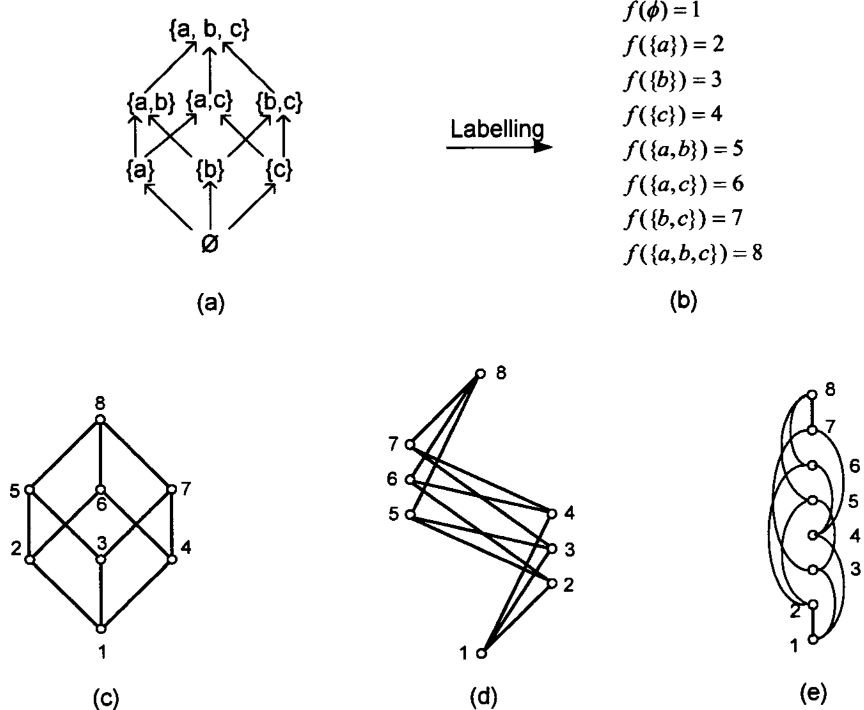


Figure 2.1: Samples of an Hasse Diagram

For instance, consider the ordered set depicted by *Figure 2.1* which is the power set $\wp(A)$ of the set $A = \{a,b,c\}$ ordered by inclusion "$\subseteq$". After labeling each element of $\wp(A)$ in *(b)*, the pictures *(c)*, *(d)* and *(e)* shows some valid Hasse Diagram. The pictures *(c)* and *(d)* use straight-line edges while *(e)* has curve edges.

2.2. Decomposition Scheme of ordered sets

The idea of decomposing an ordered set is significant in its drawing.

2.2.1. Chains and Dilworth's Theorem

Definition 2.4. Chain, Antichain, Length and Width

A *chain* C in P is a set of pairwise comparable elements. That is, it is a totally ordered subset of P such that $a \sim b$ for all $a,b \in C$. The *size* of a finite chain is $|C|$-1 where $|C|$ is the cardinality of C.
An *antichain* A in P is a set of pairwise incomparable elements. That is, it is a non-ordered subset of P such that $a \parallel b$ for all $a,b \in A$. The size of a finite antichain is its cardinality.
The *length* of an ordered set P, denoted *length(P)* is the *size* of its longest chain.
The *width* of an ordered set P, denoted *width(P)* is the size of its longest antichain.

Computationally, we rank the elements into the chain C from *Bottom(C)* as the first element and *Top(C)* as the last element.

Definition 2.5: Linear Sum and Disjoint Sum
Suppose P and Q are disjoint ordered sets.
The *disjoint sum* of P and Q, denoted $P \otimes Q$, is the ordered set which incorporates all elements of P and Q which are ruled by a new relation R such that
($x \leq_R y$ in $P \otimes Q$) if and only if either ($x, y \in P$ and $x \leq y$ in P) or ($x, y \in Q$ and $x \leq y$ in Q).
The *linear sum* of P and Q, denoted $P \oplus Q$, is the ordered set which also incorporates all elements ruled by a new relation R such that:
($x \leq_R y$ in $P \oplus Q$) if and only if ($x, y \in P$ and $x \leq y$ in P) or ($x, y \in Q$ and $x \leq y$ in Q) or ($x \in P$ and $y \in Q$)).

The *Figure 2.2* shows an example of disjoint and linear sum of two ordered sets.

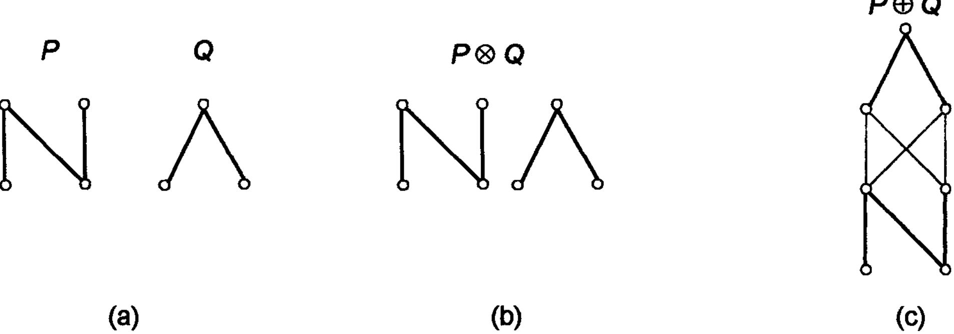


Figure 2.2: Example of a disjoint sum and a linear sum

Definition 2.6. Chain Decomposition and Antichain Decomposition

A *Chain Decomposition* of *P* is a partition of *P* into chains. Conversely, an *Antichain Decomposition* of *P* is a partition into Antichains.

According to our example of *Figure 2.1 (c)*, $P = (\wp(A), \subseteq)$ can be partitioned into 3 chains $\{1,2,5\},\{3,7\},\{4,6,8\}$ or into 4 antichains $\{1\},\{2,3,4\},\{5,6,7\},\{8\}$. It is crucial to produce the minimum possible number of chains because it will define the drawing width. Clearly, we cannot do better than *width(P)* chains. The assertion of *Dilworth*'s theorem indicates that this bound can be achieved:

Theorem 2.1 [Dilworth] *Let P be an ordered set of finite width w. Then there exists a partition of P into w chains, and this is best possible.*

2.2.2. Linear Extension and Jump

Definition 2.7. Linear Extension

A *linear extension* L of *P* is a total ordering $x_1, x_2, \ldots, x_n$ of *P* such that if $x_i < x_j$ *in P* then $i < j$. We denote by L(*P*) the set of all linear extension of *P*

Figure 2.3 shows an example of ordered set *P*, called the N ordered set *(i)*. Here L*(P)={ L_1, L_2, L_3, L_4, L_5}*. For each *L*∈L*(P)*, the comparability between elements in *P* is always respected in *L*, e.g. $(a \le c)$, $(b \le c)$ and $(b \le d)$. The total

order L_x represented by the picture *(iii)* is not a valid linear extension because $(b \leq c)$ in P but $(b \geq c)$ in L_x.

It is trivial that any finite ordered set P has linear extensions. The main concern about linear extension is usually the question that, given any ordered set P, how many linear extensions P has (e.g. $|L(P)|$) and how to construct the optimal linear extension. The meaning of optimal is also uncertain and is usually related to the context of its use. In our case, our interpretation of optimality is that our linear extension should give us a chain decomposition that gives benefits to the drawing.

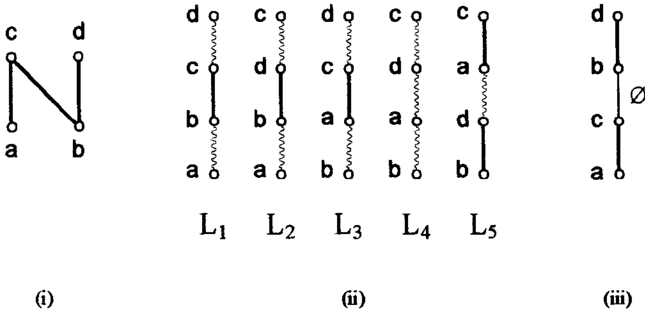


Figure 2.3: The N ordered set and its linear extensions

Definition 2.8. The Jump

Let L be a linear extension of P composed by total ordered elements $x_1, x_2, \ldots, x_n$.

A *jump* is a consecutive pair (x_i, x_{i+1}) in L if $(x_i \parallel x_{i+1})$ in P. The *jump-number* of P, denoted *jump(P)*, is the minimum number of jump that a possible linear extension L of P can have.

In *Figure 2.3 (ii)*, jumps are represented by spring (zigzag line). Since L_5 has the minimum number of jump, we say that L_5 is *jump-optimal*.

Corollary 2.1. Jump decomposes a linear extension L of P into chains. Hence,

$L = C_1 \oplus C_2 \oplus C_3 \ldots \oplus C_n$ and $jump(P) = n - 1$

In *Figure 2.3 (ii)*, the linear extension L_5 is decomposed into 2 chains $C_1 = \{b,d\}$ and $C_2 = \{a,c\}$ separated by the jump (d,a) . By notation, we can write $L_5 = C_1 \oplus C_2$.

Chapter 3

Review of Existing Approaches

In this chapter, we review the literature related to the upward drawing of ordered sets and of digraphs in general. We start by clarifying the similarities and differences of the drawing approach between ordered sets and digraphs. We also review some known upward drawing algorithms satisfying some aesthetic criteria such as planarity, crossing-minimization and area used. Finally, we conclude this chapter by listing the major existing academic and commercial software that deal with graph drawing and which has some upward drawing or Hasse Drawing as components.

3.1. Hasse-Diagram versus Digraph Approach

As previously mentioned, the directed covering graph of an ordered set is a special type of graph. A finite directed graph is the directed covering graph of a finite ordered set if it is acyclic (i.e. it has no directed cycles) and essential (i.e. for each directed edge (x, y) there is no other directed path from x to y). The upward drawing process is approached using two strategies: a drawing using the theory of ordered set or using a graph theoretical approach.

The upward drawing with respect to graph theory abandons the order theory. Thus, ordered sets elements are only considered as regular vertices. Likewise, covering relations into the order specify an existence of edges between the vertices. These edges are necessarily acyclic, essential, directed and upward. Consequently, drawing the diagram is mapped to find an upward drawing of directed edges without transitive or acyclic edges. Upward drawing of digraphs has received a significant amount of attention from researchers. A collection of algorithms can be consulted in the Graph Drawing book of *Di Battista et al* [DETT99]. Different algorithms for

some particular digraph subclasses are also listed in [BBLM96], [BCD+94], [EW94], [GGT96] and [STT81].

On the other hand, the drawing of a Hasse Diagram using the concept of ordered sets uses more its background theory. In other words, the drawing is more investigated with respect to its properties. For instance, the algorithm can take profit from the order class type (Interval Order, Series-Parallel Order, lattice...) or others assets as its Dimension, its Chains, its Antichains, its linear extension... Algorithms using an ordered set approach can be found in [AS92], [FT99], [Fre04] and [Jou95]. The algorithm we present in this thesis belongs to this type of approach.

In the review, we state the known results using both approaches.

3.2. Conventional ideas of upward drawing

3.2.1. The "Hierarchical" method

The *Hierarchical approach* is the most intuitive way to draw an upward digraph. Introduced by Sugiyama et al [STT81], it is also referred to as the Layered Drawing or the K-Level-Hierarchy Drawing. The hierarchical approach consists of 3 steps:

1. The first step is to assign vertices into levels. Thus, all nodes are assigned to horizontal layers. In order to have proper hierarchy, edges can be split by inserting virtual nodes until each edge connect two nodes on neighbored layers. (These newly introduced nodes are called dummy nodes and are eliminated at the end of the drawing. It may create bends on the final drawing but polyline edges can always be straightened). Consequently, all nodes receive a y-coordinate at the end of this step.
2. The second step is to reorder the vertices into each layer in order to satisfy some aesthetic criteria such as planarity or crossing minimization.
3. The last step is the assigning of the horizontal coordinate to all nodes.

Algorithms employing the hierarchical strategy usually differ on some minor details and are usually variations of one another:

- For instance, the first step can be fulfilled by a depth-first-search algorithm by ranking the elements. Initially, a forest F is created from the graph G. A vertex will receive a ranking 0 (lowest layer) if it is a root in F. Otherwise, it receives a rank one value bigger than its parent. Another example of layering strategy called the *Coffman-Graham* method [CG72], specifies a number of vertices for each layer. This method proposes the sorting of nodes by their maximal distance from the sources. Then, it assigns at most k nodes to each layer by choosing the largest node whose descendants have already been placed.
- However, these variants using the hierarchical strategy usually differ on the step 2 due to the combinatorial fact of the node rearrangement. The *Barycenter heuristic* reorders nodes on the variable layer according to the barycenter weight [STT81]. Thus, each vertex will be placed in the average center of its neighbors. That is, the barycenter of a vertex v (thus defining the position) in the changeable layer is equal to 0 if v does not have neighbors in the fixed layer. Otherwise, the barycenter will be equal to the value of the "sum of all neighbors positions" divided by "number of neighbors". Another heuristic, called the *Sifting Heuristic* [MSM00], first determines the order of the vertices on the changeable layer. Following this order, each vertex will be placed on a different position (sifting) where the number of crossing is minimal. Also, ordering of all other vertices remains fixed during the sifting. Note that the global crossing minimization over n layers is still an open problem. An intuitive heuristic to decrease the number of crossings is to alternately traverse the layers top-down and bottom-up, until the total number of crossings no longer decrease. For instance, a heuristic to test planarity of k-level hierarchy is investigated by *Di Battista et al.* in [DN88]. They characterize some forbidden configurations between two layers and build successively the layers from bottom to top till the prohibited pattern is met.
- The number of dummy nodes inserted is also crucial if only straight-line edges are authorized since these nodes create bends on edges. Thus, the challenge that can be raised is to minimize them. Also, another issue is the minimization of the width of each layer since this width also sways the area used by the drawing. This situation is

investigated by the Coffman-Graham method by assigning a fixed number of vertices per layer.

Nowadays, many slightly different versions of the Sugiyama approach exist depending on the context of its use. For instance, *Seemann* produced a version of the algorithm to build UML class Diagram [See98].

3.2.2. The "Force Directed" method

The *Force Directed* strategy utilizes the laws of physics. The upward digraph is viewed as a system containing bodies on which the forces of repulsion or attraction act. The goal of the algorithm is to find a design for the bodies that brings equilibrium to the force system. The *Force Directed* method consists of 2 steps:

1. Build a model of forces which is described by the nodes and edges.
2. Implement an algorithm that discovers the equilibrium of the system.

Usually, attractive and repulsive forces are used in the step 1. Attractive forces are used along each edge and repulsive forces are used between each pair of nodes. Various forces can also be in place such as the system gravity centre or the angular forces between edges incident from the same node.

In practice, the most usual force model used is a "spring force". Nodes are regarded as electrically charged particles that repel one another while Edges are regarded as springs connecting the particles. Hence, particles that are far away from one another attract each another by spring forces while particles that are too close repel one another. Unfortunately, spring forces do not take the direction of edges into account. Hence, a "magnetic force" system can be used for directed graphs because all edges should have a uniform direction to point to. For upward drawing, edges are interpreted as magnetic needles that align themselves according to a magnetic field which necessarily points upward.

- The various algorithms using the Force-Directed strategy usually differ in the force or the energy model used, as well as in the method used to find the equilibrium of the system. *Fruchterman et al.* [FR91] use the initial idea of mass particles to draw graphs. *Tutte* also uses the same concept to place each vertex in the center of gravity

of its neighbors [Tut63]. Several "force-directed" algorithms applied on upward drawing are presented in [DETT99].

- The advantage of this approach is that it is easily extensible. The equilibrium concept also moves the nodes proportionally to the force applied to them. Hence, symmetry of the drawing is one of this approach's significant assets. However, it is often criticized by its running time complexity due to the computation of all forces. Indeed, the repulsive force computation has a quadratic time complexity since a graph can have quadratic edges. Another problem is that one must decide which forces (magnetic, spring...) are to be exploited and how to define termination conditions. Moreover, solving techniques of other aesthetic criteria (such as crossing minimization) are not necessarily incorporated into the force systems. Thus, graph drawing researchers usually qualify this approach as a "bag of tricks".

3.2.3. The "Divide-And-Conquer" method

In contrast to both previous approaches mentioned previously, the *Divide-And-Conquer* approach does not act on any arbitrary upward graphs. It only operates on some special types of digraphs that can be expressed recursively. For instance, trees or series-parallel digraphs present some recursive characteristics. The *Divide-And-Conquer* approach is split into two steps.

1. The first step is to identify a base case and to find a configuration to draw it.
2. The second step is to find a strategy to decompose the original graph into pieces in order to retrieve the base case at the end of the recursive path.

Various algorithms using the *Divide-and-Conquer* strategy usually proceed with the same step 2 but generally diverge with respect to step 1. Algorithms using this paradigm on rooted trees are showed in [GGT96]. A drawing algorithm of series-parallel digraph using this approach is shown in [BCD+94]. Other algorithms for both types of graph are surveyed in [DETT99]. *Divide-and-Conquer* is also used in [BBLM96] to portray upward tri-connected digraphs.

3.2.4. Related results on ordered sets

In the theory of ordered sets, the Sugiyama layered drawing can be seen as an antichain decomposition if the hierarchy is proper. Indeed, all elements belonging to the same layer are incomparable. A decomposition of the digraph into layers is then identical to a decomposition of the ordered set into sets of Antichains.

A force-directed approach is used by *Freese* in [Fre04] to automatically draw some lattices. Firstly, a rank function computes each point of the ordered set to determine its height representing its z-coordinate. Points of the same rank are placed around a circle parallel to the x-y plane. The position of points belonging to the same circle (x and y coordinate) is defined by a system of forces. All comparable elements attract one other while non-comparable elements repel one another. Finally, the points are projected from 3D-space to 2D-space.

The Divide-And-Conquer approach can also be executed on drawing Hasse Diagram. The most famous subclass that can present recursive assets is the series-parallel ordered sets. These ordered sets can be constructed recursively within some linear or disjoints sums between the elements. The base case will be the way to draw two elements separated by disjoint or linear sum. The recursive path will be the way of decomposing the ordered set itself.

3.2.5. Results for some classes of graph

After presenting these conventional ideas of drawing, we now list some bound results according to some drawing satisfying some aesthetic criteria. The tables listed below are based on the graph drawing book referenced by [DETT99].

We recall that testing planarity for a general graph is known as not only polynomial but overall linear [HT74]. Nevertheless, the complexity changes for the upward drawing. *Garg and Tamassia* [GT95] proved that upward planarity is NP-Complete by reducing it into a network flow problem. Thus, good results are only known for some classes of digraph. For instance, *Eades et al.* [EW94] show that the process of verifying the upward planarity is linear for a single source digraph.

It is clear that in certain cases, planarity cannot be satisfied. For this situation, minimizing the crossing is then the new resolution. For an upward drawing, the problem is as difficult as the planarity because Crossing Minimization is NP-Hard [Joh82]. Moreover, crossing Minimization for upward bipartite graphs is NP-Hard [EW94].

The above automatically implies that both problems receive the same complexity for an Hasse Diagram. A detailed study of the crossing number of ordered sets was made by Lin in [Lin94] confirming the NP-Hardness of minimizing the crossing number. *Table 3.1* summarizes the time complexity of upward planarity testing and crossing minimization for some classes of digraphs.

Table 3.1: Time complexity of planarity testing and crossing minimization on Graphs

Graph class	Aesthetic Criteria	Time Complexity	References
General Graph	Testing Planarity	$\Theta(n)$	[HT74]
Digraph	Testing upward Planarity	NP-Complete	[GT95]
General graph	Minimize Crossing	NP-hard	
Bipartite Graph (pre-assignment order on one 1 layer)	Minimize Crossing	NP-hard	[EW94]
Single-source digraph	Testing upward planarity	$\Theta(n)$	[EW94]

We now discuss the area occupied by the upward drawing. It is clear that specification of the type of edges plays an important role in the area taken by the drawing. Obviously, straight-line edges present more restrictions on the drawing than polyline edges. Hence, straight-lines increase the drawing surface. It is known that there exist some classes of planar acyclic digraph that needs exponential lower bound in any straight line embedding on a grid [DTT92]. However, if polyline edges are allowed, then this area may become quadratic. The area is again exponential even for upward planar series-parallel digraph if the embedding is preserved [BCD+94]. However for a rooted tree, both upper-bound and lower-bound of the area spent is $n \log n$ [GGT96]. *Table 3.2* gives a survey of the bound complexity of the area used of some classes of upward drawing.

Table 3.2: Existing upper bounds and lower bounds on the area of upward planar drawing.

Graph class	Drawing type	Area Used		References
		Ω	O	
Upward planar digraph	grid, straight-line	$\Omega(a^n)$ $a>1$	$O(b^n)$ $b>1$	[DTT92]
Upward planar digraph	Grid, polyline	$\Omega(n^2)$	$O(n^2)$	[DTT92]
Series-parallel digraph	Upward, planar, grid, straight-line, Embedding-preserving	$\Omega(a^n)$ $a>1$	$O(b^n)$ $b>1$	[BCD+94]
Series-parallel digraph	Upward, planar, grid, straight line	$\Omega(n^2)$	$O(n^2)$	[BCD+94]
Rooted tree	Upward, planar, grid, straight line,	$\Omega(n \log n)$	$O(n \log n)$	[GGT96]

The NP-Completeness and The NP-Hardness aside, we now present some good results that are known for upward drawing: (*Table 3.3*)

- A divide-and-Conquer strategy by *Chan et al* [CGKT97] succeeds in producing a linear time algorithm to draw an upward rooted tree within a planar embedding on a grid and spending $O(n \log n)$ area.
- If the upward graph is planar and polyline edges are accepted then a linear time algorithm is presented by *Di Battista et al* in [DTT92].

Both algorithms use a divide-and-Conquer strategy.

Table 3.3: Time Complexity of various Upward Drawing of selected graph

Graph class	Drawing Type	Time Complexity	References
Rooted tree	Upward, straight-line, Planar, grid, with $O(n \log n)$ area	$\Theta(n)$	[CGKT97]
Upward planar digraph	Upward, Polyline, planar, grid with $O(n^2)$ area and $O(n)$ bends	$\Theta(n)$	[DTT92]

3.3. Existing software

There is a wide variety of graph viewers that exist. Generally, most graph software contains all the functionalities related to upward drawing (or downward drawing). These different pieces of software commonly implement the classical conventional approach of upward drawing. For instance, any existing software that deals with hierarchy necessarily contains the Sugiyama approach. However, some software packages may also contain particular algorithms that were developed for a specific reason. However, the software is usually supported by a collection of actual literature in the field. Here, we only list some of them.

Practically, all researchers or developers in the field of graph visualization have encountered *LEDA* at one time or another. It is the most well-known library that contains the most efficient data structure and algorithms known in the field of graph drawing. It was designed using an object oriented approach and implemented with C++. All classical problems in graph theory (shortest path, flows, matching, coloring ...) are included in the package, thereby facilitating the jump between the graph drawing concept and various graphs problems that may be related. The package is also continuously updated to incorporate new results in graph drawing research. Many algorithms related to Upward-drawing have already been implemented in this package. Several academicals and commercials software have been known to get profits from its library.

As an example, *AGD-GraphWin* is a software package that uses *LEDA*. It is simple visualization software which draws basic shapes of a graph and classical improvement that can be executed on the drawing. It also implements upward drawing by using Sugiyama method. It can also draw hierarchies with a force directed strategy. Another software package that uses the LEDA library is *GDT* (Graph drawing Toolkit). It was designed to efficiently manipulate several types of graphs. One of its interesting features is that it automatically portrays graphs according to many different aesthetic criteria and constraints.

Another benchmark for graph visualization is *GraphWiz*. In addition to the same functionalities as LEDA, it also provides suitable User Interface libraries.

Mocha is an "Algorithm animation" that can be run over the World Wide Web. It is useful for understanding graph algorithms as it allows user to visually follow visually step-by-step execution of the algorithms. Famous algorithms are embedded in this software including the computation of the Convex-Hull animations of any upward drawing.

Aisee is a commercial software which automatically calculates a customizable layout of graphs specified from an input file of a specific format. In this software, hierarchies are drawn downward in the original picture. However, the drawing can be easily flipped to retrieve the upward version.

Software which visualizes upward drawing using ordered sets concepts are quite rare. *LatticeDraw* is one example of software that automatically generates lattice diagrams from its abstract representations. It can also draw ordered sets. The algorithm used by latticeDraw is inspired from a force directed approach.

OrderExplorer is a software built by researchers from the University Of Ottawa. Along with *LatticeDraw*, they are the only known systems that uniquely investigate upward drawing using ordered sets theory.

The systems used to accomplish the comparisons against the LR-FlowOrder are the *AGD GraphWin* (due to its uses of Sugiyama method) and the *LatDraw* (force-directed). GraphWin implements several layering methods including the Coffman-Graham and the DFS ranking. Different crossing-minimization heuristics is also offered (BaryCenter, Sifting …). *LatticeDraw* is our benchmark for force-directed software.

Chapter 4

The LR-Drawing

The *LR-Drawing* is a modified version of the upward drawing concept. It is a new way to visualize ordered sets. Our approach is to use a chain decomposition of the ordered set as the underlying structure for positioning the vertices of the order.

Fundamentally, the process is to place each chain on a different vertical line. Just this very simple idea will allow a better visualization for some classes of ordered sets, especially the ones which are decomposed by small number of chains. The reason is in this case a major part of the relations will be within the same vertical line. However, there is still difficulty (similar to the upward drawing) in spotting the ordering relations between pairs of vertices which are not drawn on the same vertical line. In fact the covering relations between different chains could either be from left to right or from right to left. That is if x covers y then x is higher than y but x could be either on the left or on the right of y.

Instead of drawing the order using any chain decomposition as the underlying structure, we consider the chain decompositions derived from a linear extension. This type of upward drawing will only have covering relations from left to right. The majority of direct applications need to have the ability to find out if many relations of the order are easy to spot, that is, given any two elements, one wants to visually and quickly find whether these elements are comparable or non-comparable. Our new approach will still satisfy the conventional upward drawing (Bottom to Top direction) and will insert a new "extra implicit information" which is "from Left to Right" direction. Hence, the drawing will display a certain "flow" with respect to those directions so that the viewer will be able to trace the path from an element to another easily. It is what we call the *LR-Drawing* or "Left to Right upward drawing".

4.1. Drawing Assumptions and Criteria Hypothesis

We list here all assumptions and hypothesis concerning *P*, the drawing type of *P* and the drawing workspace.

The restriction that we have on the ordered set *P* is that:

- *P* is a *Finite* ordered set. In other words, the number of elements of *P* is finite.
- Let (a, b) be an edge from a diagram of *P*. We can also express it as ($a \prec b$) to stress the upward direction. Also, *P* is represented by a matrix of coverability for a theoretical study of the time complexity; i.e. verifying if ($a \prec b$) hold is in $O(1)$.

The drawing convention of *P* is that:

- Hasse Diagram will only have "*straight line*" edges. Polyline (*Figure 4.1(a)*) and curves (*Figure 4.1(b)*) edges are not considered.

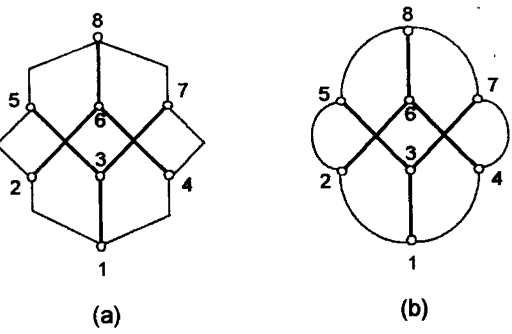


Figure 4.1: Example of Hasse Diagram with polyline and curve edges

Graphically, the hypothesis is that the drawing will be fulfilled in the 1st Quadrant of a two dimensional Cartesian coordinate of range 1. In other words:

- It is a pseudo *grid drawing*. (see *Figure 4.2* (a))
- Edges can intersect anywhere in the plan. It is a slight difference from the conventional grid drawing where crossings have integer coordinates (see *Figure 4.2* (a) for crossing between edges ($3 \prec 5$) and ($2 \prec 6$)).
- The colinearity issue is beyond the scope of this thesis and will not be investigated. Due to its geometry particularity, we assume at the end of the drawing, an external procedure rearranges some the vertices to avoid such shortcoming.

The aesthetic criteria that are considered are

- The number of crossings.
- The area used by the drawing. We will consider the *smallest rectangle* embedding the drawing. (*Figure 4.2* (b)). We use R to denote the rectangle, ω (R) to indicate its width and ℓ (R) respectively its length.

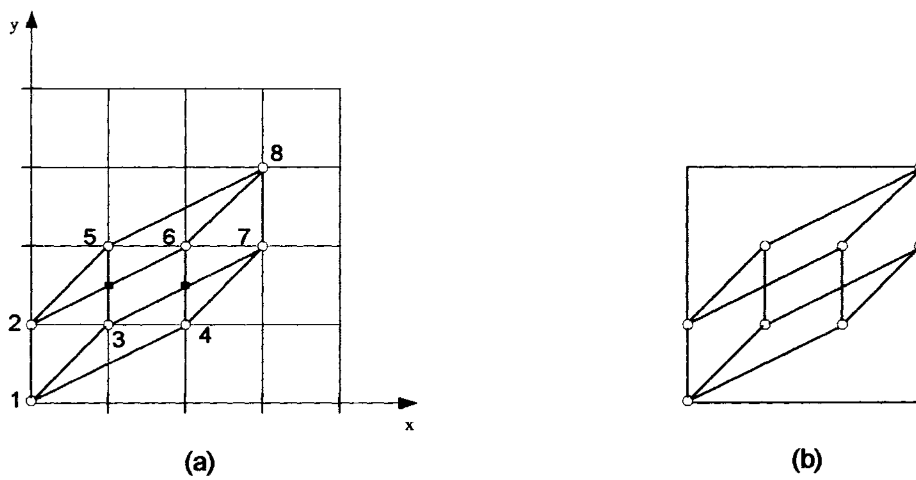


Figure 4.2: An example of pseudo grid-drawing and the rectangle area associated

4.2. The "Bottom-to-top" and "Left-to-Right" Idea

The "Idea" is motivated by a simple concept: "Make the relations easy to spot". To achieve this idea, our proposition is to:

- *Motivation 1: Eliminate the crossing between left-to-right and right-to-left edges.* The point is to avoid the confusion between crossings of inclined edges. To do so, one type of edges should not exist and the choice has been made to remove the existence of right-to-left edges. This choice is not "randomly" made because the motivation is based on the idea that more people tend to look from left to right. (Of course, transforming the algorithm to satisfy the other opposite direction requires a non-significant amount of work).
- *Motivation #2: Maximize the number of elements that will be placed on the same vertical line.* The motivation is based on the idea that maximizing the number of

vertical relations will minimize the number of slanted ones, and consequently improve the readability.

Informally the idea can be summarized as follows:

1. Generate an "appropriate" Linear Extension that will produce a decomposition of the ordered set into Chains.
2. Take the chains one by one (starting from the lowest position on the linear extension to the greatest) and place them vertically from "Left to the right".
   a. For each element on a chain, make sure its location is above its covered elements already drawn.
   b. Progressively, Link edges between the actual chain and the remaining chains already drawn.
3. Improve the drawing.

The decomposition of P into chains (Step 1) and the placement of the chains from left to right (Step 2) will satisfy the absence of right-to-left edges (*Motivation #1*). The choice of the "appropriate" linear extension is essential in order to maximize vertical comparability (*Motivation #2*). We show that "greedy" linear extensions (with respect to the number of jumps) will satisfy this criterion.

We will study the steps of the idea in details in the upcoming sections. First, we will start by convincing the reader of the benefits of both motivations mentioned above. Secondly, we will discuss the maximization of the vertical relations using the notion of a greedy linear extension and how to generate one. Finally, we will conclude with the pseudo-code of the drawing algorithm of a general ordered set P.

4.2.1. Definition of LR-Drawing

In any upward drawing, we recall that there are exactly 3 types of edges: the vertical one (strictly upward), the upward left-to-right one and the upward right-to-left one. In a general drawing, combinations of crossing between these 3 types can occur. By avoiding right-to-left edges, we eliminate two types of crossing including

the major one which leads to the biggest confusion, the crossing between the left-to-right and the right-to-left edges.

Definition 4.1 The LR Drawing
The *LR Drawing*, LR being abbreviated from Left-To-Right, is an upward drawing where existing edges are only vertical or strictly left-to-right upward.

4.2.2. A new aesthetic metric: the "Flow of Comparability".

The advantage of the LR Drawing is felt "significantly" from the visual point of view, especially when there is a considerable number of crossing edges. Indeed, taking element $\alpha, \beta \in P$, checking the comparability between them can be easily tracked by a path starting from α and going only "upward" or "left-to-right" till we reach β. For instance, take the ordered set in *Figure 4.3* and its two different representations. The picture *(a)* shows a symmetric drawing and *(c)* a LR Drawing, where *(b)* and *(d)* are respectively their comparability path between the elements 2 and 10. From an overall view, the symmetric representation is probably aesthetically appealing. However, if the viewer wants to quickly find out the relation between elements 1 and 10, the LR Representation definitely offers more facility. The reason is that in the symmetric picture the viewer has to do a "zigzag", instead of following a defined bottom-to-top left-to-right path in the LR Drawing. Consequently, this situation of a comparability path will induce a "flow of comparability" going from bottom to top and from left to right (*Figure 4.4*).

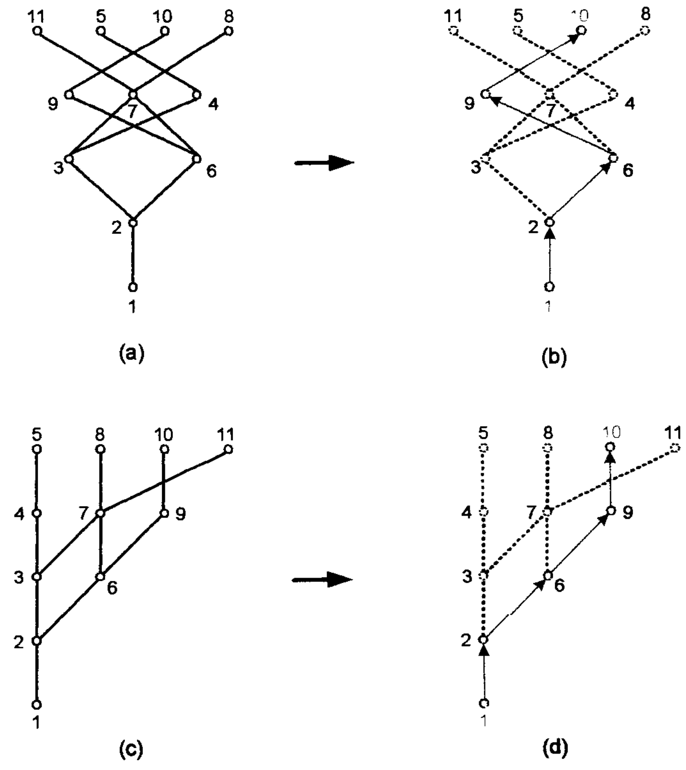


Figure 4.3: Example of "flow of Comparability" differences between a symmetric drawing and a LR Drawing.

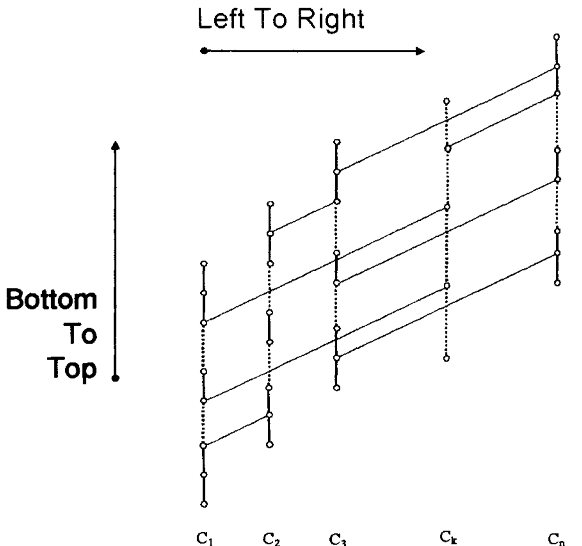


Figure 4.4: The LR Drawing's flow of comparability

4.2.3. Maximization of vertical relations by "Greedy Linear Extension".

Assume P is a finite ordered set and L be a linear extension of P where $L = C_1 \oplus C_2 \oplus C_3 \ldots \oplus C_n$ and where C_i are the chains. It is trivial that if we put the chains from left to right in the drawing then we cannot have right-to-left edges. Thus, any linear extension will satisfy the idea of flow of comparability.

The specific choice of a "greedy linear extension" instead of another is based on some special motivations:

- *To optimize the number of elements that will be ranked into the same chain:* Minimizing the number of jumps is equivalent to maximizing the number of consecutive pairs which are covering pairs in *P*.
- It *has special characteristics for N-Free ordered sets*. We will show later that the drawing will be efficient in terms of time complexity, area used and drawing improvement techniques. This motivation is redoubled since we know that any ordered set can be converted into N-Free by a linear time algorithm.

The first question raised is if any ordered set can produce a linear extension which is greedy. It is a well known fact that *every ordered set P contains at least one linear extension which is greedy.*

The next question is how to generate a greedy linear extension. In a usual language, the algorithm principle is simple and follows one rule: "Climb as high as you can". Formally, for an arbitrary finite ordered set *P*, the generation of one greedy linear extension can be written as follow:

1. Choose $x_1 \in \{\min P_1\}$ where $P_1 = P$
2. After each step i,
   - choose $x_{i+1} \in \{x \in \min P_{i+1} : x > x_i\}$ where $P_{i+1} = P_i - \{x_i\}$ for $i = 1,2,\ldots,n-1$ if there is one;
   - Otherwise choose $x_{i+1} \in \min P_{i+1}$.

The pseudo-code of the algorithm, using a Divide-and-Conquer Strategy and having a $O(n*width(P))$ time complexity can be seen here (*Algorithm 4.1*). As the algorithm induces the decomposition of the linear extension L into greedy chains C_i, L is not explicitly symbolized by a variable but is represented by a succession of chains C_i. Then, computationally, the initial call of the algorithm is called with parameters the ordered set P, one arbitrary minimal element *min* and an empty chain C_1.

```
Greedy Linear Extension (P: ordered set ; min: Element; Ci: Chains)
Input: An ordered set P, an arbitrary minimum element min and an empty chain C1.
Output: A Linear extension L where L= C1 ⊕ C2 ⊕ C3 ⊕... ⊕ Cn and width(P)
Begin
        findASuccessorMinimum: Boolean := false
        NewMin: Element := null
        add the element min to Ci
        if min is not maximal then
                for each element j such that j ≻ min
                        if j is not covering another element than min then
                                findASuccessorMinimum := true
                                NewMin := j
        if findASuccessorMinimum is true then
                Greedy Linear Extension(P, NewMin , Ci)
        else
                NewMin = Get a New minimum from P
                if NewMin ≠ null then
                        Create a new empty chain Ci+1
                        Greedy Linear Extension(P, NewMin, Ci+1)
                else
                        width(P) := i
End
```

Algorithm 4.1: The generation of a "greedy" linear extension.

We also note in the algorithm the presence of one function called "Get a New minimum from P" which is in charge of selecting a minimal element into P which has not been selected yet. Two cases occur when a new minimal is required; the rise into the actual chain of P stops when, first a maximal element is reached (*Figure 4.5(b)*) or; secondly when an element which has other non-selected immediate successor is

met (*Figure 4.5(d)*). In each case, a Jump occurs into the linear extension before the new minimum is selected (respectively *Figure 4.5(c)* and *Figure 4.5(e)*).

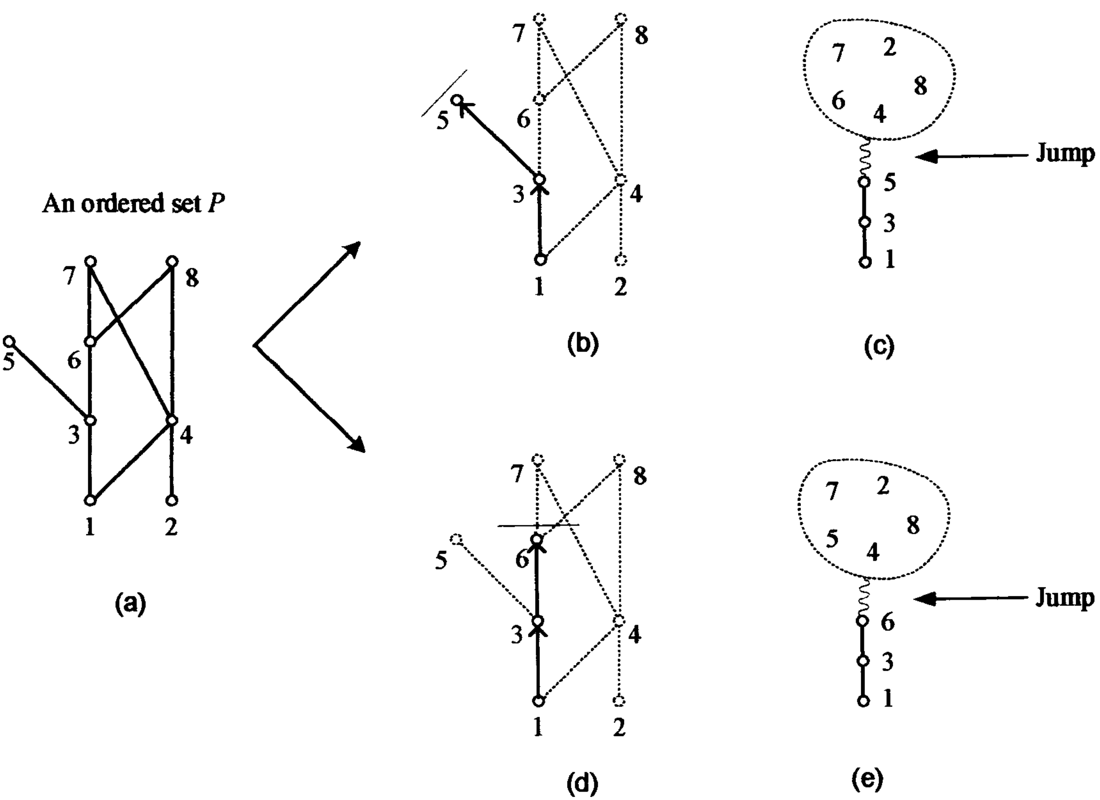


Figure 4.5: The end of a rise in a Greedy Linear Extension for a ordered set P.

If the new minimum selected is the immediate successor of an element just visited then our selection can be seen as a backtracking of a Depth-First-Search algorithm applied to *P*'s diagram (for instance, the generation of L_a from the *Figure 4.6*). Evidently, the choice of a minimum will influence the drawing since the chain decomposition generated will be different. Note that the combinatorial problem of the generation of all linear extensions of an ordered set *P* is a well studied question (see *Figure 4.6* which is an extension of *Figure 4.5(c)*).

The algorithm also computes *width(P)* and can easily be updated to compute *length(P)*. The Java-Code of the algorithm can be consulted in the Appendix section at page 107.

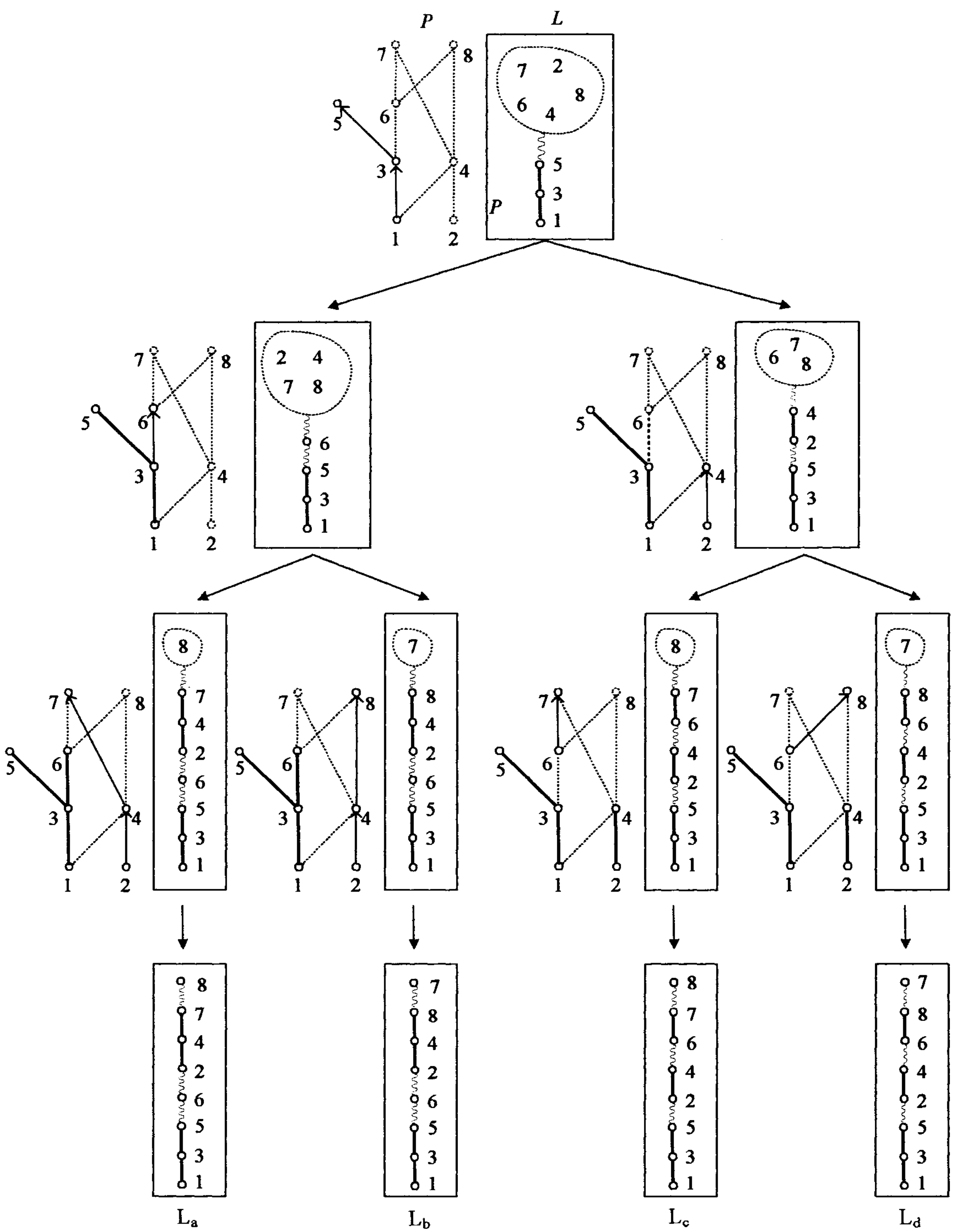


Figure 4.6: Some execution scenarios of the greedy linear extension algorithm

4.3. Drawing Algorithm

The algorithm is a direct interpretation of the idea of decomposing the ordered set into chains based on a greedy linear extension. The chains are progressively placed one by one on screen from left to right.

4.3.1. Pseudo-Code.

The *Algorithm 4.2* shows a pseudo-code of the drawing algorithm. The first step (i) is to generate a linear extension L which will decompose the ordered set P into chains C_1 to C_n. For each chain C_i, we know that all elements belonging to C_i will have X-coordinate equal to i (the range being 1 since it is a grid drawing). The second step (ii) is the assignation of the Y-coordinate for the elements e of C_i. This value is the maximum of the Y-coordinate of all the predecessors of e (hence $t \in \downarrow^{im} e$ and $t \in C_x$ where $1 \leq x \leq i$) incremented by 1. Elements and edges of P can be drawn progressively (as shown here) or at the end of the algorithm when coordinates of all elements are known (as in the implementation). The java-code of the LR-Drawing algorithm can be consulted in the Appendix section.

Even though an enhancement strategy of the picture is not investigated here, we will show in the upcoming chapters that aesthetic improvement can be brought into a special class of ordered sets.

Drawing-Algorithm(P: ordered set)
Input: an ordered set P
Output: a drawing of P in a 2D Cartesian plan
Begin
(i) L:= Generate_A_Greedy_Linear_Extension(P)
Let $L := C_1 \oplus C_2 \oplus C_3 \oplus \ldots \oplus C_n$ where C_i are the chains.
For i =1 **to** n
Assume C_i = {Bottom(C_i), …, y, z, Top(C_i)}
For each element e of C_i from Bottom(C_i) to Top(C_i)
Let $maxY$ the highest vertical coordinate of all $t \in \downarrow^{im} e$
(ii) Place e at the position x=i and y=$maxY$+1
Draw the edges between t and e
End

Algorithm 4.2: The General ordered set Drawing Algorithm using LR strategy

4.3.2. Time Complexity.

Let $T(n)$ be the time complexity of the function *Drawing-Algorithm*, where n is the number of elements of P. Hence,

$$T(n) \in (O(i) + O(ii)) = O(\max(O(i), O(ii))) = O(\max(n * width(P), O(ii)))$$

Even if we have two "for" loops into the part including *(ii)*, both combined actually run exactly n times. Finding the maximum Y-coordinate of all predecessors of any element e will require, in the worst case, at most $width(P)$ comparisons. Drawing the edges between the chains will use the same amount of time. Hence, we can affirm that $ii \in O(n * width(P))$. Consequently $T(n) \in O(n * width(P))$. With a large upper-bounding of $width(P)$ by n, the drawing algorithm has a quadratic time on the number of elements.

4.3.3. Examples of execution

Figure 4.7 shows possible execution of the LR-Drawing algorithm applied to the example of ordered set P presented in the *Figure 4.5*. Each level represents one complete execution of the algorithm. We assume the four linear extensions L_a, L_b, L_x and L_y are obtained from the greedy linear extension algorithm already presented in *Algorithm 4.1* meaning they are all greedy. The first two executions use linear extensions from *Figure 4.6* (L_a and L_b) and require 4 steps of positioning the chains in the drawing since each linear extension is decomposed into 4 chains. The last two executions use two new possible extensions of P (L_x and L_y) and require 5 steps.

It is clear to see that the first two pictures, Figure 4.7*(a)* and Figure 4.7*(b)*, have isomorphic shape. Thus, two different linear extensions can produce similar embedding. Both pictures also present only one crossing, making the drawing acceptably readable. Moreover, a planar embedding can be fulfilled if the last element from the linear extension (element "8" into L_a and element "7" into L_b) has a greater Y-Coordinate. The *Figure 4.7(c)* has two crossings and the *Figure 4.7(d)* is the least readable with four crossings. We can also note that these last two drawing are generated from two linear extensions (L_x and L_y) where the jump-number are greater than the first two linear extensions (L_a and L_b). Indeed, $jump(L_a) = jump(L_b) = 4$ while $jump(L_x) = jump(L_y) = 5$. Obviously, the rectangle's width used by the embedding is directly proportional to the jump-number of the linear extension because

ω(R) of the LR-drawing of P using $L = jump(L) + 1$.

Consequently, the last two drawing are less efficient in terms of space used by the drawing.

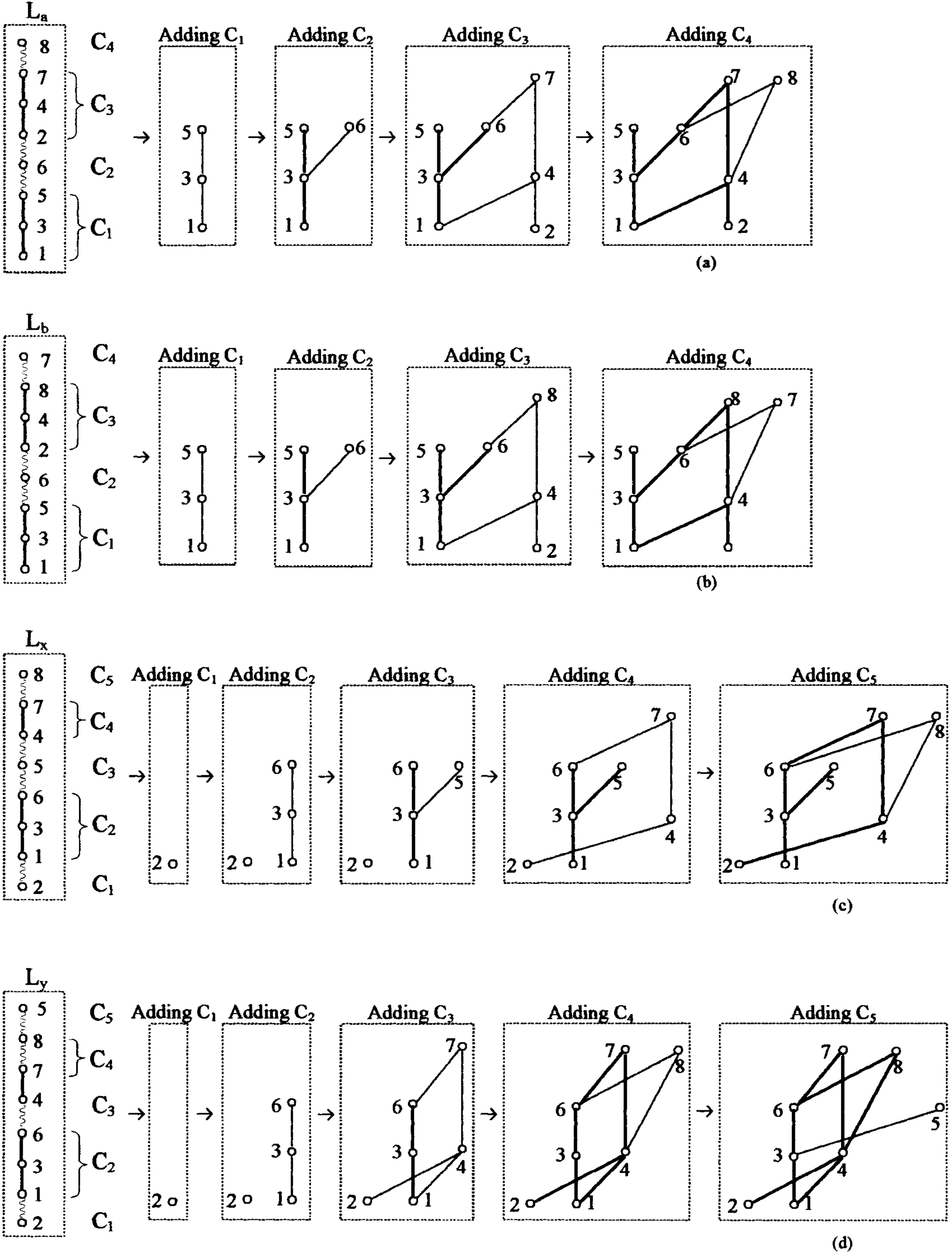


Figure 4.7: Samples of execution of the LR-Drawing on different linear extensions

4.3.4. Problematic and Need for a Parameterization

As seen from the previous section, the structure of the greedy linear extension is crucial. Indeed, two of the main aesthetic parameters of a drawing are related: the number of crossings and the space used. The major issues are:

- How to characterize the ranking of chains into the greedy linear extension in order to get a more readable drawing? Obviously, generating all possibilities and selecting the best one is not a feasible approach.
- How to ensure that the rectangle space used is the minimum? Based on the preceding example, generating a greedy linear extension with a minimum jump-number in a polynomial time is not possible for a general ordered set *P*.

In the upcoming chapters, we approach the problems with respect to a special class of ordered set called N-Free. We first introduce the N-Free Class and its characteristics.

Chapter 5

N-Free ordered sets

A good drawing solution that works for all ordered sets is clearly out of reach. However, if every ordered set could be embedded into another with a particular structure, and at the same time we are able to nicely draw these particular structures then this could lead to an interesting approximation of general ordered set drawing.

In the subsequent chapters we will investigate the LR-Drawing using the approach described above. Our special structures are the N-free ordered sets. It is an important class in the theory of ordered sets and has been very well investigated due to its relations with many applications.

5.1. Motivations

Our *"first motivation"* is the fact that finding a jump-optimal linear extension of an N-free ordered set is an easy problem.

The *"second motivation"*, surely the most fundamental, answers the first problem concerning the readability of the diagram. We will show that our LR-Drawing introduces new innovations to the drawings of that class.

The *"third motivation"* to study this class is that every finite ordered set is embedded as a subset in an N-free ordered set. Indeed, every finite ordered set can be converted into N-Free within a linear time for an extra space representing the total number of covering edges in the worst case. By taking advantage of this idea of conversion, for instance, an implementation of the drawing of a general ordered set P can be split into three steps:

1. The first step is to convert P into an N-Free ordered set Q,
2. The second step is to apply the LR-Drawing to Q,
3. The third and last step is to retrieve P from the drawing of Q.

5.2. The N-Free Subclass

The N ordered set

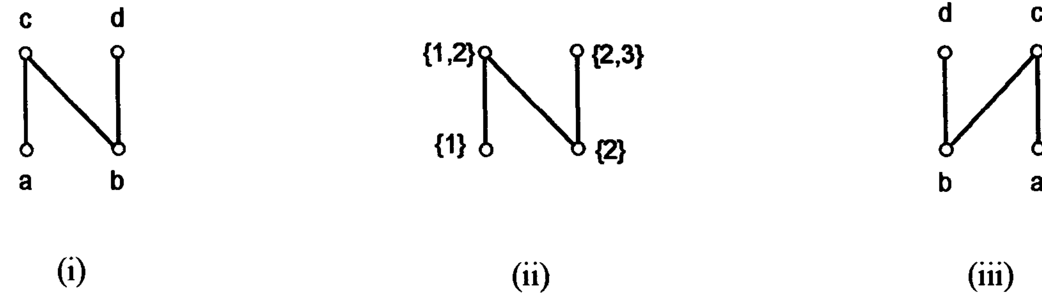


Figure 5.1: The N ordered set.

Definition 5.1 The N ordered set and the N-freeness

The *N ordered set* is an ordered set of 4 distinct elements *a*, *b*, *c* and *d* such that the relations $a \prec c$, $b \prec d$ and $b \prec c$ are the only comparabilties among these elements

An ordered set is *N-free* if it diagram contains "no" sub-diagram isomorphic to N.

The *Figure 5.1(i)* shows the natural diagram of the N ordered set. The *Figure 5.1(ii)* shows a practical configuration of an N where the order is the inclusion between sets. Moreover, its diagram is not necessarily shaped as the alphabet N. For instance, the *Figure 5.1(iii)* represents another embedding of the N ordered set. Since we focus on the LR-Drawing, this last representation will certainly be the most referred because the diagonal edge $b \prec c$ has a left-to-right direction.

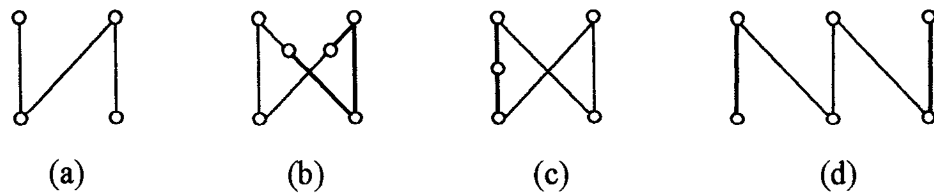


Figure 5.2: Example of diagrams containing N.

All ordered sets, whose diagram are exemplified by the *Figure 5.2*, contains sub-diagram (or sub-diagrams) isomorphic to N. Hence, they are not N-Free. Neither of the ordered sets illustrated in *Figure 5.3* contains N; hence they all are N-Free.

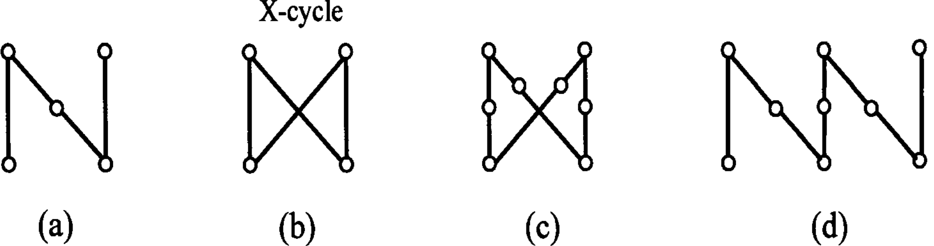


Figure 5.3: Example of N-Free ordered set

From the two previous figures (*Figure 5.2* and *Figure 5.3*) we can note that there are two distinct "natural" approaches to convert an ordered set with N into an N-Free one. The first approach, considered as an "edge approach" is to insert an extra "diagonal" edge into the N order (for instance, the transformation of the ordered set in *Figure 5.2 (a)* into the ordered set in *Figure 5.3(b)* follows this approach). In a comparability point of view, this approach is inaccurate because the comparability relation between elements on the original ordered set is violated. The second approach, considered as a "vertex approach" is to insert an extra element on the "diagonal" edge (e.g. ordered set in *Figure 5.2 (a)* to ordered set in *Figure 5.3(a)*). This last approach preserves the comparability relation between elements and is usually considered as the process of converting an ordered set into an N-Free one. The next subsection will develop two different algorithm of the "vertex approach".

5.3. The N-Free Converter methods

5.3.1. The "naïve" method

A natural approach can obey this rule: Every finite ordered set can be transformed as an N-free one by "subdividing" every edge in the diagram. That is, for every edge ($x \prec y$) is the diagram we add a dummy vertex z, such that $x \prec z \prec y$.

The algorithm can be implemented as follow:

```
Naïve-N-Free-Converter (P: ordered set)
Input: An ordered set P
Output: P which is N-Free
Begin
    P': a temporary variable representing a set

    P':= empty
    Add all elements of P into P'
    For each edge e = (x ≺ y) of P
        Create a new element z and Add z into P'
        Add covering relations x ≺ z and z ≺ y into P'
    P: = P'
End
```

Algorithm 5.1: The naïve algorithm to convert an ordered set into N-Free.

It is clear that this algorithm has a linear time complexity and linear space expenses on the number of covering relations (or edges in the diagram).

5.3.2. The "space efficient" method

We now present one of our contributions that allow us to save additional elements. We propose a more efficient algorithm in term of space earning which can still reach the polynomial time complexity. Instead of inserting a dummy element at every edge, it only targets the group of edges which has an N and inserts a new element in the diagonal edge (e.g. the diagram of *Figure 5.2(d)* into the diagram of *Figure 5.3 (d)*). However, a principal issue exists: Assume P' is the new ordered set after inserting dummy elements to all diagonal of N in P; The main issue is: will the insertion of these new elements in P give rise to some new N's in P'?. We show this possibility as in the example of *Figure 5.4* where the new elements labeled 6 and 7 create some new N (for instance the set {2,3,1,6} is an N). Consequently, the issue is to know when to stop subdividing the diagonal edges? We will prove that if this subdivision is run twice, the final ordered set produced is guaranteed to be N-Free.

The *Algorithm 5.2* , which is clearly polynomial in time complexity, interprets this idea.

Optimal-N-Free-Converter (P: ordered set)
Input: An ordered set P
Output: P which is N-Free
Begin
  P': a temporary variable representing a set

  Do twice
    P':= empty
    Add all elements of P into P'
    Add all covering relations of P into P'
    For all distinct elements a, b, c, d of P
      if (N with c, d on top and a, b below) is in P and $b \prec c$ has not been subdivided yet
        Create a new element z and Add z into P'
        Add covering relations $b \prec z$ and $z \prec c$ into P'
        Delete the covering relation $b \prec c$ into P'
    $P := P'$
End

Algorithm 5.2: The space efficient algorithm to convert an ordered set into N-Free

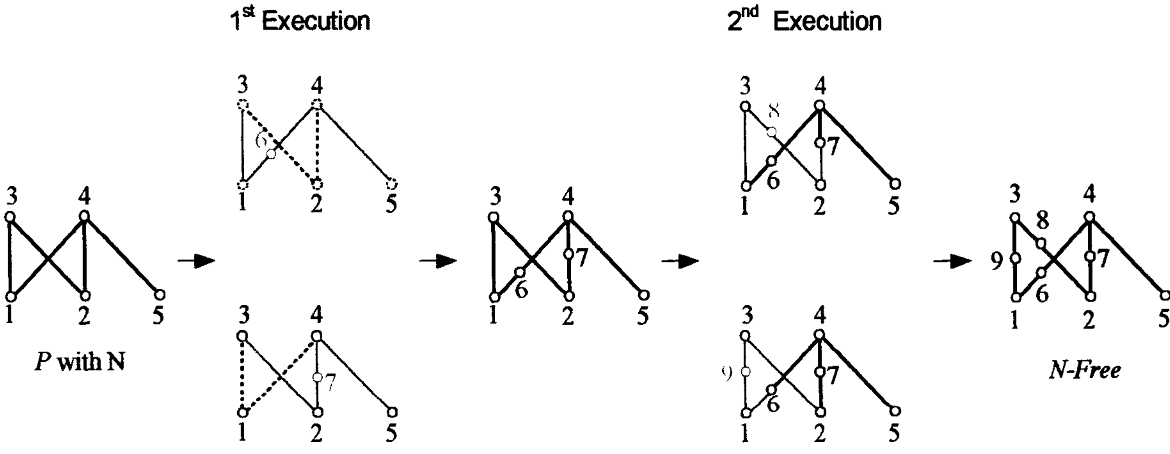


Figure 5.4: Execution of the space efficient N-Free converting algorithm.

The *Figure 5.4* shows an execution of this algorithm. Two dummy elements 6 and 7 were created during the first execution. Another two, 8 and 9, were produced during the second execution. Clearly, it shows that the ordered set after the second execution does not contain N. Moreover, the edge $(5 \prec 4)$ does not need to be subdivided and shows the space economy not satisfied by the naïve algorithm. This economy can be significant depending on the initial configuration of P.

5.3.3. The proof of correctness of the space-efficient algorithm

We formally prove that the N subdivision must be run at most twice. Let f be the function which subdivides diagonal edges of all N of an ordered set *P*. By notation, assume that $f(P) = P'$ and $f(P') = P''$. Our goal is to prove that *P''* is N-Free.

By notation, for two ordered sets A and B, (A \ B) denotes the set of elements belonging to A but not to B. These following lemmas are useful for our proof.

Lemma 5.1 Let *P* be an ordered set and let $f(P) = P'$. If (N with elements a, b, c, d) is an N in *P'* then *a* or *d* are in $(P' \setminus P)$.

PROOF: the elements *a* or *d* are necessarily in $(P' \setminus P)$ otherwise $(b \prec c)$ would have been subdivided.

Lemma 5.2 Let *P* be an ordered set. Let $f(P) = P'$.

i. If only *a* (or *d*) is in $(P' \setminus P)$ then *P'* contains (figure)

ii. If both *a* and *d* are in $(P' \setminus P)$ then *P* contains (figure)

PROOF: We prove both assertions hold.

- Assertion (*i*) hold.

Suppose is in *P'* and only a is new. Let a' in P such that $a \succ a'$. It means we have this following configuration in *P*. Since is not an N in P (otherwise, we should subdivide $c \succ b$), then $a' < d$. (Also $a' < b$ is false otherwise c does not cover a' in *P*).

Suppose $d > a'$ is not a covering relation in *P*, then there exists an element x such that $d \succ x > a'$ in *P*, i.e. . Clearly, $x \parallel b$ because $x > b$ implies that d cannot cover b and $x < b$ implies d cannot cover x. Thus, is an N in P but $d \succ b$ was not subdivided in *P'*. Therefore, x does not exist and so we have this following configuration in *P'*.

Similar argument applies if d is the only new element in *P'*.

- Assertion (*ii*) hold.

Suppose that both a and d are new in *P'*. Similar arguments as previously presented shows that we will have this configuration where d' and a' are in P and that $d' > a'$.

Suppose d' does not cover a' then there exists an element x such that $a' < x < d'$. Therefore in *P*, we had the configuration in case x is in *P*. In case x is not in *P*, then we have this configuration . The same conclusion holds if $d' \succ a'$.

Since the edge (c, a') have been subdivided in *P'*, there was an N in *P*, i.e. .

The original picture *P* has this following configuration . Thus, $v > b$ and $u < d'$ because of the N's and . But, this situation will create an

N in P, which is , where (c, b) is a diagonal. Since $c \succ b$ is not subdivided, we have once again a contradiction. Therefore, our hypothesis which stipulates that x is not in P cannot stand. We conclude that we have the configuration in P.

Lemma 5.3 Let P be an ordered set. Let $f(P) = P'$ and $f(P') = P''$. Necessarily, P' and P'' do not contain

Proof: Suppose P' contains then clearly x must be new. There are no other elements in P' between a and x. Therefore, are all cover relations in P. Since we subdivided $a \prec c$, then there is an N in P, i.e. (Note that either u, v or both can exist). Then, we have this configuration . But then, none of the edges $a \prec d$, $b \prec c$ and $b \prec d$ is a diagonal in any N in P since none was subdivided in P'. Thus, $v > b$ and $u < d$. Suppose $u \prec d$, then is an N. But, we have a contradiction since $d \succ a$ was not subdivided. Suppose $u < x \prec d$ then is an N. Once again, we have a contradiction for the same reason. We conclude that our hypothesis which stipulates that P' contain is not valid. We conclude that P' does not contain .

A similar method can be used to prove that neither P'' contains the same configuration. In the proof, P'' substitutes P' *and P'* replaces P. And thus, our proof is concluded.

We use the previous lemmas to finally prove that P'' do not contain N

Lemma 5.4: Let P be an ordered set. Let $f(P) = P'$ and $f(P') = P''$. Consequently, P'' is N-Free.

PROOF : Suppose P'' contains an N c d a b,

- According to *Lemma 5.2(i)* , only a (or only d) is in $(P'' \setminus P')$ and therefore P'' contains ⋈ . But we have a contradiction to *Lemma 5.3*. Consequently, our hypothesis which stipulates that P'' contains an N cannot stand.
- According to *Lemma 5.2 (ii)*, both a and d are in $(P'' \setminus P')$ and therefore P' contains ⋈ . But we have a contradiction to *Lemma 5.3*. Consequently, our hypothesis which stipulates that P'' contains an N cannot stand.

5.4. Properties and Characteristics

5.4.1. Jump optimality

Theorem 5.1 [Rival] *For a finite N-free ordered set every greedy linear extension is jump-optimal. Indeed, in this case, every jump optimal linear extension is greedy too.* [Riv83]

Every greedy linear extension algorithm does not guarantee a minimum number of jumps for a general ordered set. However, the jump-optimality is met for an N-Free ordered set according to *Rival*'s theorem. It means, our LR-Drawing, which uses a greedy linear extension, will always satisfy the minimum number of chains for N-Free ordered sets. Consequently, the minimum width of the rectangle area used by the drawing is also met.

5.4.2. The X-cycle Free Assumption

The Figure 5.3*(b)* shows one special type of an N-Free sub-diagram called a X-cycle. Its' particularity is the fact that the subdiagram itself has an N and the N-Freeness is established with the extra-diagonal edge. We formally define the cycle X:

Definition 5.2 X-cycle-Free.

An *N-Free* ordered set is *X-cycle-Free* if it does not contain 4 distinct elements *a*, *b*, *c* and *d*, such that: $a \prec c$, $b \prec d$, $b \prec c$ and $a \prec d$.

The N-Free ordered set illustrated by the diagram in *Figure 5.3(b)* is not *X-cycle-Free* while the remaining (a), (c) and (d) are. We note that the N-Free converting algorithm, presented in *Algorithm 5.1* will always produce an N-Free ordered set which is *X-cycle-Free*. This is not the case if we transform the ordered set by adding new edges.

The following Lemmas will be very useful in simplifying several proofs.

Lemma 5.5: Let *P* be an N-free ordered set which is X-cycle-Free. If *P* contains a four element subset {a, b, c, d} such that $c \succ a$, $d > a$ and $c > b$ (we call it a forbidden quadruple) then necessarily either $b < a$ or $c < d$.

PROOF : Suppose that neither $b > a$ nor $c < d$. Consider two elements c' and a' such that $b < c' \prec c$ and $a \prec a' < d$. Since {c', a, c, a'} cannot be an N then $a' > c'$. (c' cannot be larger than a' for otherwise (a, c) won't be a covering relation). If $a' \prec c'$ then we have an X. So, we suppose that $a' \succ e > c'$ for some element e. Clearly {e, a', a, c} will form an N in the ordered set. It contradicts our assumption that *P* is N-free and X-cycle free ordered set.

Lemma 5.6: Let *P* be an N-free ordered set which is X-cycle-Free. There are no forbidden quadruples {a, b, c, d} in *P* such that $a \prec d$ in *P* and simultaneously $b \prec c$ in *P*.

PROOF : Suppose that {a, b, c, d} is a forbidden quadruples in *P* and suppose that $a \prec d$ and $b \prec c$. According to *Lemma 5.5*, we either have $b < a$ or $c < d$. If $b < a$ then $b < a < c$ and so (b, c) won't be a covering relation in *P*. If $c < d$ then $a < c < d$ and so (a, d) won't be a covering relation in *P*.

5.4.3. "Existing LR edges" Lemma

By using the LR-Drawing on N-Free ordered sets which are X-cycle-Free, the first note that we notice is that only "special type of left-to-right edges" can hold into the drawing. These LR edges are composed by any element of one chain to the bottom of others chains ("* to Bottom") and the top of one chain to any others element of other chains ("Top to *"). The next Lemma is the formal definition of this property.

Lemma 5.7: Let P be an N-Free ordered set which is X-cycle-Free. Let L be a greedy linear extension of P where $L = C_1 \oplus C_2 \oplus C_3 \oplus \ldots \oplus C_n$. Let C_i and C_j two chains from L such that i<j. There are at most two possible links between the two chains:

i. $(Top(C_i) \prec \beta)$, where $\beta \in C_j \setminus Bottom(C_j)$ (*Figure 5.5*)

ii. $(\alpha \prec Bottom(C_j))$, where $\alpha \in C_i \setminus Top(C_i)$ (*Figure 5.6*)

PROOF : We will use the *Figure 5.7* as a reference to the proof.

Let C_i and C_j two chains from L and suppose we have a covering relation $\alpha \prec \beta$ such that:

$\alpha \in C_i \setminus Top(C_i)$ and $\beta \in C_j \setminus Bottom(C_j)$

Let α' and β' be two element in P such that $\alpha' \in C_i$, $\beta' \in C_j$. Also, assume $\alpha \prec \alpha'$ and $\beta' \prec \beta$ in P. Therefore $\alpha' \| \beta'$. Indeed since $\alpha' < \beta'$ in L then α' cannot be larger that β' in P, moreover if $\beta' < \alpha'$ then $\alpha < \alpha' < \beta' < \beta$ and thus $\alpha \prec \beta$ won't be a covering relation in P.

Thus { α', α, β, β'} is an N in P (*Figure 5.7 (a)*), which is a contradiction. Notice that this property is true even if P contains X cycles.

The only possibilities left are the ones described in (i) and (ii).

- Proof of the proposition (i):

Suppose that there exists i such that $Top(C_i) \prec Bottom(C_j)$. We may assume that i and j are as close as possible with such property. Since $Top(C_i) \prec Bottom(C_j)$ holds,

then *Bottom*(C_j) should cover some element x in a chain C_k where $I<k<j$. (Otherwise, since L is a greedy linear extension, *Bottom*(C_j) should be a part of the chain C_i). Due to the choice of i, x cannot be the top of C_k. Thus, x is covered by y for some y in C_k (*Figure 5.7(b)*). Therefore $\{Top(C_i), x, Bottom(C_j), y\}$ will form a forbidden configuration and according to *Lemma 5.5,* either $Top(C_i) < x$ (it is impossible, for otherwise x will be between *Top*(C_i) and *Bottom*(C_j) and this contradicts that *Top*(C_i) is covered by Bottom(C_j) in P) or $y > Bottom(C_j)$ (it is impossible since $y < Bottom(C_j)$ in the linear extension). Therefore for every $i<j$, *Top*(C_i) cannot be covered by *Bottom*(C_j).

Notice that for every pair $i<j$ we could have at most one covering relation of the type

$$(Top(C_i) \prec \beta) \quad , \text{where } \beta \in C_j \setminus Bottom(C_j)$$

- Proof of the proposition (ii):

The exact same proof presented in (i) also works for *Bottom*(C_j) and the elements placed in chains located on C_j left side.

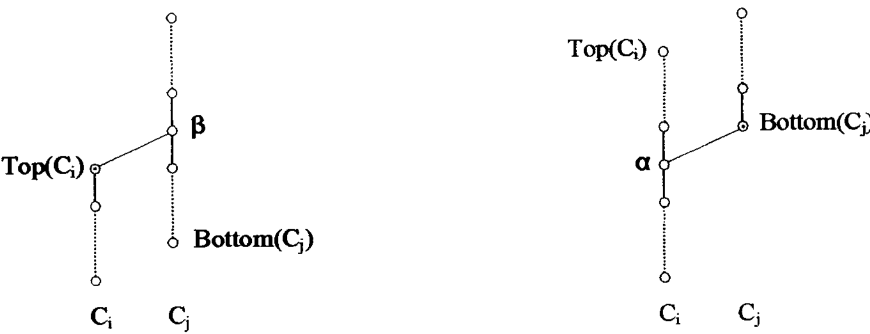


Figure 5.5: Links from "Top" for N-free **Figure 5.6:** Links to "Bottom" for N-free

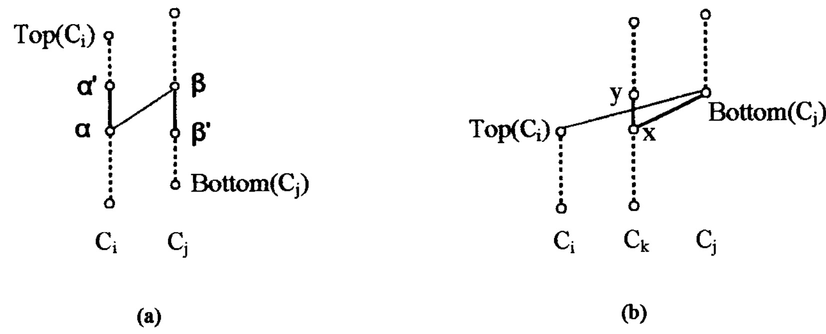


Figure 5.7: The forbidden links between two chains

5.4.4. "Unicity of Links from Top and Bottom" Lemma

In the preceding subsection, we proved that only special type of LR edges stand in the LR drawing. One extremity of this special edge is either the "Bottom" or the "Top" element of one chain. Moreover, we also notice that both elements Bottom and Top are only linked "once" to the remaining left chains and the remaining right chains, respectively. We formally define this property in this next lemma:

Lemma 5.8 Let P be a finite N-free ordered set which is also X-cycle-free.

Let L be a greedy linear extension of P where $L = C_1 \oplus C_2 \oplus C_3 \oplus \dots \oplus C_n$ and where C_i are the chains. Then, in our LR Drawing:

i. *Bottom*(C_i) can “receive” at most “one” link from any left chain C_j where $1 \leq j < i \leq n$ (*Figure 5.8*)

ii. *Top*(C_i) can “send” at most “one” link to any right chain C_j where $1 \leq i < j \leq n$. (*Figure 5.9*)

PROOF :

- The *Figure 5.8* is a reference to the following proof, which proves (i).

Let x be an element of C_j and y an element in C_k such that $1 \leq k < j < i \leq n$, $x \prec Bottom(C_i)$ and $y \prec Bottom(C_i)$. According to *Lemma 5.7*, neither x nor y could be the top of its respective chain. Let z be an element in C_j that covers x. Clearly, the subset {x, y, *Bottom*(C_i), z} will form a forbidden quadruple. Moreover, since z covers x and *Bottom*(C_i) covers y, this situation contradicts *Lemma 5.6*.

- The same argument in (i) can be used to prove (ii). The *Figure 5.9* is a reference to the analogy.

The following results are direct consequences of the preceding lemmas.

Corollary 5.1 *Let P be a ordered set which is N-Free and X-cycle-Free .The maximum number of LR links between two different chains is less or equal to 2.*

Corollary 5.2 *Let P be a ordered set which is N-Free and X-cycle-Free. Let* $\delta(P, n)$ *be the overall Number of Left-to-Right links of the LR Drawing of P of n greedy chains. Then,* $\delta(P, n) \leq 2\ (n - 1)$.

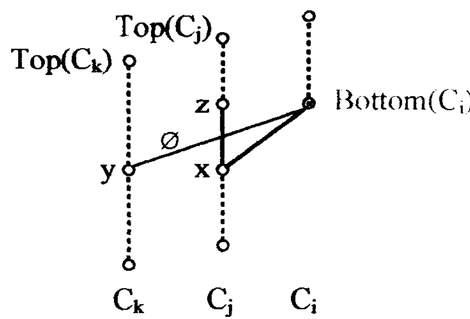


Figure 5.8: Unicity of Link towards "Bottom" of any chain

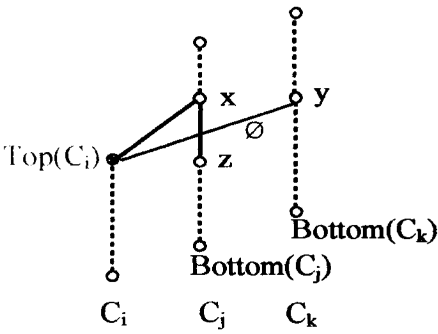


Figure 5.9: Unicity of Link from "Top" of any chain

5.5. The Chains Interchange technique

The chain interchanging is a systematic technique that allows us to manipulate chains of a greedy the linear extension without affecting its basic property of being greedy. It only works for N-free ordered sets and does not affect the number of chains in the decomposition. This operation has been introduced by *Rival* [Riv83] to successfully prove that any greedy linear extension is optimal for the jump number problem. Moreover, every optimal linear extension for the jump number problem is actually a greedy linear extension [RZ86].

This technique is very crucial to our approach since it will allow us to navigate among the greedy linear extension in order to find the "appropriate" one that could be used as the underlying structure for our upward drawing.

5.5.1. The technique of interchanging two consecutive chains

Let P be an N-Free ordered set and let $L = C_1 \oplus C_2 \oplus C_3 \oplus ... \oplus C_n$ be a greedy linear extension of P. For every index i such that $0 \leq i < n$, there are at most two covering relations between the chains C_i and C_{i+1}, that is, $(Top(C_i) \prec u)$ for some $u \neq Bottom(C_{i+1})$ in C_{i+1} and $(v \prec Bottom(C_{i+1}))$ for some $v \neq Top(C_i)$ in C_i. Let

$C'_i = \{x \text{ in } C_i: x \leq v\} \cup \{x \text{ in } C'_{i+1}: x < u\}$ and

$C'_{i+1} = \{x \text{ in } C_i: x > v\} \cup \{x \text{ in } C'_{i+1}: x \geq u\}$

We transform the greedy linear extension $L = C_1 \oplus ... \oplus C_i \oplus C_{i+1} \oplus ... \oplus C_n$ into another greedy linear extension $L' = C_1 \oplus ... \oplus C'_i \oplus C'_{i+1} \oplus ... \oplus C_n$. We denote this operation by $IC(L, i) = L'$ (see *Figure 5.10*).

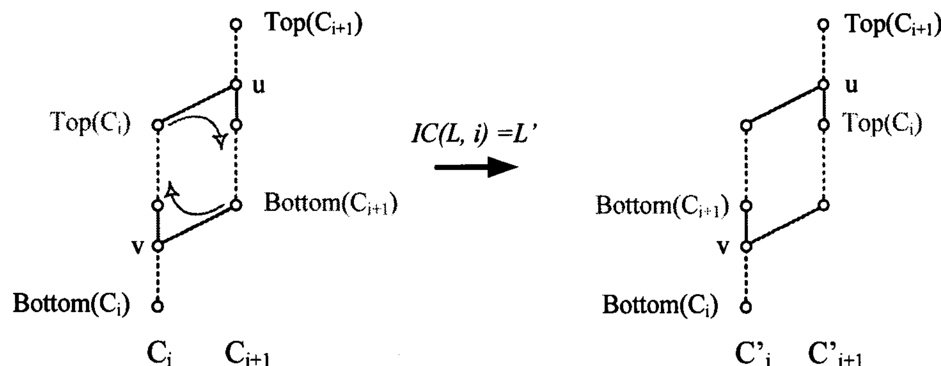


Figure 5.10: The chains interchange between two consecutive chains

Notice that any of the four subsets in both definitions of C'_i and C'_{i+1} could be empty depending of the configuration between the chains C_i and C_{i+1} (*Figure 5.11*).

It is also important to discern that the operation of interchanging chains does not necessarily work if the two chains are not consecutive. The example of *Figure 5.12* is sufficient to show its unfeasibility.

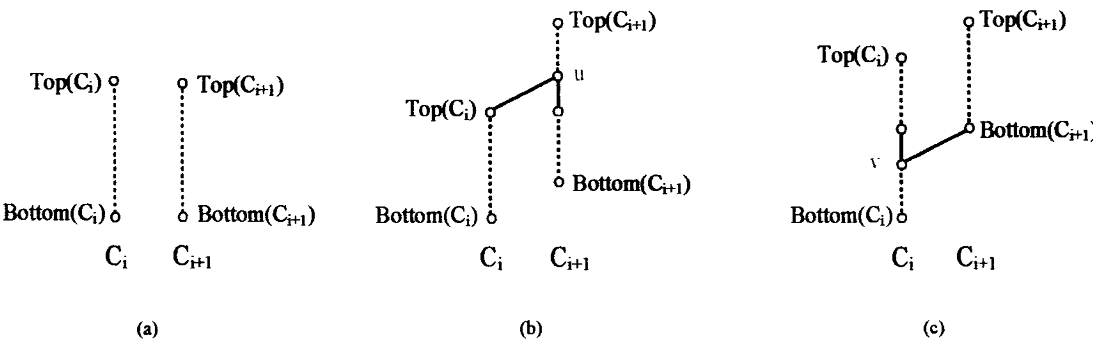


Figure 5.11: The remaining possible configuration between two consecutive chains.

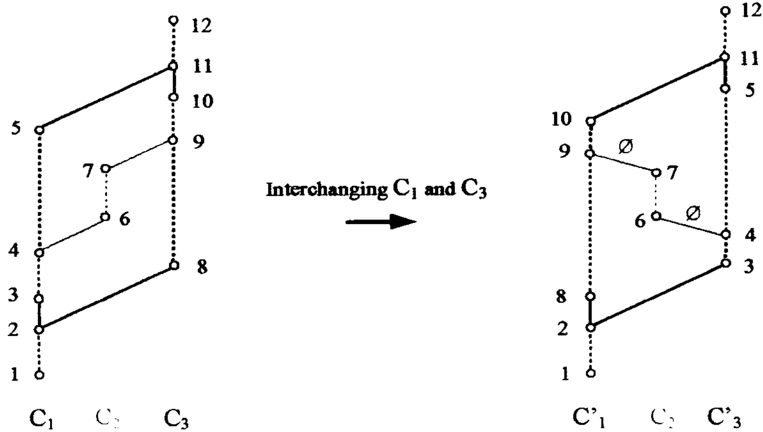


Figure 5.12: An example of configuration where a chain interchanging is directly undoable.

Notice that if L is a greedy linear extension then L' is also a greedy linear extension. Moreover, L and L' have the same number of chains. In fact, for any pair L and L' of greedy linear extensions, L' can be obtained from L by a sequence of chains interchanges operations of type $IC(L, i)$.

Corollary 5.3 *Let P be a finite ordered set which N-Free and X-cycle-Free.*

Let L be a greedy linear extension of P where $L = C_1 \oplus C_2 \oplus C_3 \oplus \ldots \oplus C_n$, *and let* L' $= IC(L, i)$ *for some* $0 \leq i < n$.

i. *L' is a greedy linear extension of P.*

ii. *Let u, v* are *two elements* in $C_i \cup C_{i+1}$ and u || *v in P. Then u<v in L if and only if v<u in L'.*

Proof : We saw that the jump-number of *P* is preserved by the interchanging techniques since *L* and *L'* have the same number of chains. Consequently, the new linear extension after interchanging the chains is also jump-optimal since the *Theorem 5.1* stipulates that all linear extensions which are jump-optimal are necessarily greedy. It proves that the new linear extension *L'* is also greedy which proves (i). Assertion (ii) is directly proved by the definition of the C'_i and C'_{i+1}.

Chapter 6

LR Drawing on N-Free

Most of the graph drawing algorithms generally propose global techniques to improve the visual aspect of the drawing. In general, their idea is to propose a systematic algorithm which is applied to the overall picture in order to satisfy some general aesthetic criteria such as planarity or symmetry. Our approach is quite different in the sense that we start with a specific chain decomposition, improve the drawing locally by concentrating on the neighborhood of a specific chain, and then recursively expand the same technique to both sides of that chain.

In this chapter, we present our main drawing algorithm which is specifically applied to the class of N-free and X-free ordered sets. The main aesthetic criterion is to reduce the overall number of crossings.

6.1. The Interchange-Chains Algorithm

The first basic technique to manipulate the drawing is a process that allows us to control some elements positions of the ordered set. The *Interchange-Chains* algorithm is the implementation version of the *Chain Interchange* technique previously defined by the function *IC(L, i)* and depicted by the *Figure 5.10*. The *Algorithm 6.1* shows its pseudo-code which highlights the elements exchange between C_i and C_{i+1}. We use two temporary variable chains C'_i and C'_{i+1}, initially empty, which will contain the updated values of the new chains C_i and C_{i+1}. Variables u and v are used to reference involved elements which link both chains. To validate that u belongs to C_{i+1}, it takes at most $length(C_{i+1})-1$ comparisons. Dually, the same case applies to v with $length(C_i)-1$ evaluations. The elements exchange between consecutive chains depends on the data structure type. If transfer of values is required then it takes at most $(length(C_i) + length(C_{i+1}))$ operations, which is $O(height(P))$.

Interchange-Chains (L: LinearExtension, i: index of a Chain)
Input: A linear extension L such that $L = C_1 \oplus \ldots \oplus C_i \oplus C_{i+1} \oplus \ldots \oplus C_n$
An index i which indicates that C_i will be interchanged with C_{i+1}
An implicit ordered set P which is N-Free and X-cycle-Free
Output: A new L where C_i and C_{i+1} are interchanged.
Begin
C'_i, C'_{i+1} : Temporary chains to retain the new contents of C_i and C_{i+1}
u, u', v, v' : Temporary references to some elements of P.

Let u be the element that covers $Top(C_i)$ in P
if $u \in C_{i+1}$
Let u' be the element that is covered by u in C_{i+1} (i.e. $u' \prec u$)
else
Let u' represents $Top(C_{i+1})$.
Note that $\{u,...,Top(C_{i+1})\} = \varnothing$
Let v be the element that is covered by $Bottom(C_j)$ in P
if $v \in C_i$
Let v' be the element that covers v in C_i (i.e. $v \prec v'$)
else
Let v' represents Bottom(C_i).
Note that $\{Bottom(C_i),...,v\} = \varnothing$
$C'_i := \{Bottom(C_i),...,v\} \cup \{Bottom(C_{i+1}),...,u'\}$ as ordered.
$C'_{i+1} := \{v',...,Top(C_i)\} \cup \{u,...,Top(C_{i+1})\}$ as ordered.
Replace C_i by C'_i into L.
Replace C_{i+1} by C'_{i+1} into L.
End

Algorithm 6.1: The Chain-Interchanging algorithm of two consecutive chains.

However, if only updated references are enough to satisfy the new configuration (as in a linked list) then $O(1)$ operations is sufficient. The java code to fulfill the chains interchange algorithm can be consulted in the Appendix section, at page 112.

6.1.1. Execution samples

The *Figure 6.1* shows an execution of the algorithm applied on an arbitrary linear extension L highlighting special cases of links between concerned chains.

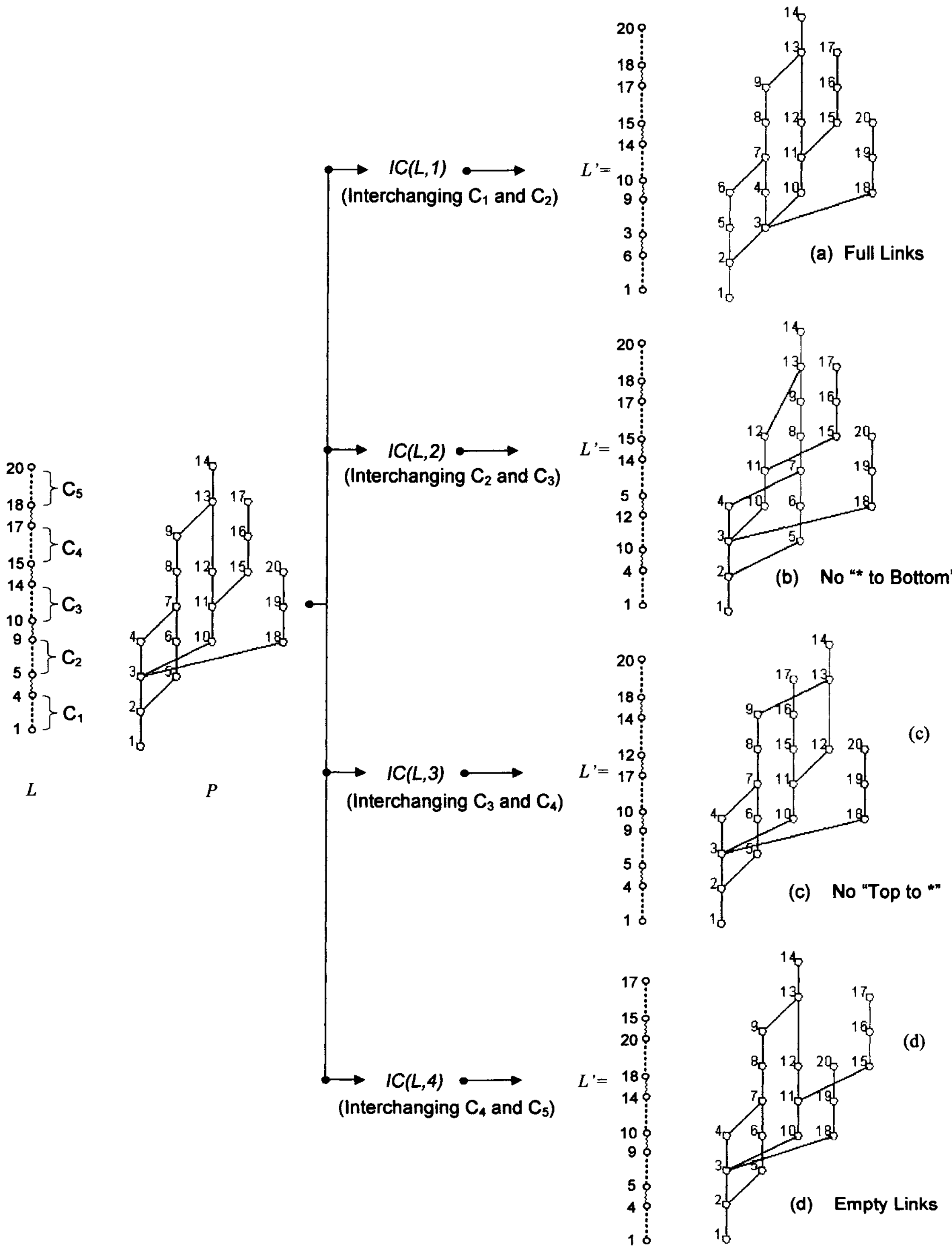


Figure 6.1: Samples of the Chains Interchange highlighting the 4 different cases configuration between two consecutive chains

6.2. Local Reorganization around a Chain

Contrary to many conventional approaches, the goal is not to directly arrange globally the upward drawing. Instead, we first approach the problem within a local improvement of certain parts of the drawing and then recursively generalize this local enhancement. At first, we only concentrate on one local branch of the drawing which is the *chain* itself. The first question raised is what should be the best local neighborhood representation of that chain. So initially, the difficulty was analyzed from an "artistic" point of view. Then, we conclude that the configuration displayed in the *Figure 6.2(a)* is the most credible representation due to its symmetry and its absence of crossings. As displayed, the configuration follows one rule: "the more a chain is linked by a smaller element, the more it is closer". The process is to select any chain C_i and to arrange the neighbors of C_i following the configuration of the proposed picture. (The term "neighbors" is designated to the chains which are linked to elements of C_i except $Bottom(C_i)$ and $Top(C_i)$). The accomplishment can be split in two independent processes: the reorganization of the left side of C_i depicted by the *Figure 6.2 (b)* and the C_i right-side reorganization (*Figure 6.2 (c)*).

Furthermore, we also place C_i 's neighbors as close as C_i. In other words, we place the left neighbors of C_i following a consecutive decreasing ranking from the index *i* to the index *i-n* (see the indexing of each left neighbor chain on *Figure 6.2(b)* assuming that C_i has *n* neighbors). Conversely, the right neighbors of C_i will have an increasing ranking from *i* to m (*Figure 6.2(c)*).

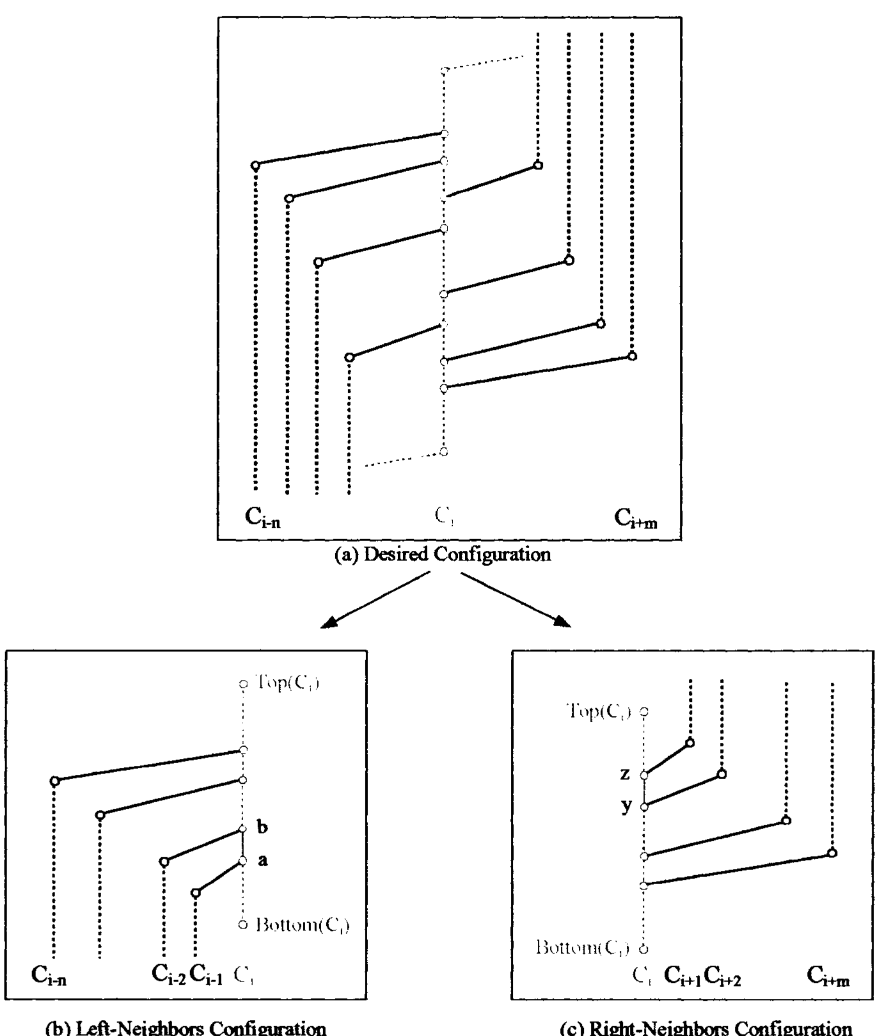


Figure 6.2: The proposed chain neighborhood after the local improvement

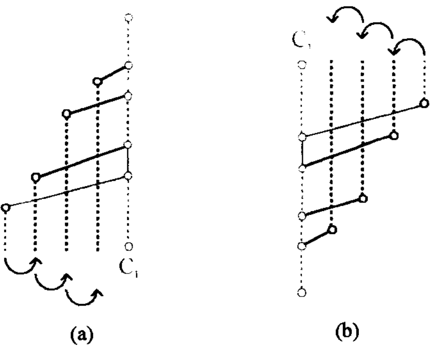


Figure 6.3: The worst case configuration of the chain neighborhood

6.2.1. The left neighbors chains restructuring

The left-neighbors chains have to be ordered as displayed by the *Figure 6.2(b)*. To accomplish this proposition, our suggestion is to "pull" successively each left neighbor C_k until it reaches its "targeted" place. We start the process by pulling the neighbor chain which is linked by the smaller element in C_i. The general formula for the position is as follows: if C_k is linked by the "d^{th} element of C_i having left-link" then the targeted place of C_k is the position $(i-d)$. However, the place exchange between C_k and the chain actually located at position $(i-d)$ cannot be accomplished immediately if $(i-d)$ and k are not consecutive (as we already noticed by the Interchange chain technique). Consequently, to fulfill the pulling, we require $((i-d)-k)$ *Chains Interchange* from the index k to the index $(i-d)$.

Formally, let $L = C_1 \oplus C_2 \oplus \dots \oplus C_i \oplus \dots \oplus C_n$ be a greedy linear extension of P, and let $i<n$. We define the set *Left-Links* (L, C_i) as the of set elements x in P, such that x=Top(C_k) for some $k < i$ and x is covered by some element in C_i. The reorganization of the left-neighborhood of a chain C_i in a greedy linear extension can be established as follows:

Proposition 6.1 Let P be a finite ordered set which is N-Free and X-cycle-Free.
Let $L = C_1 \oplus \dots \oplus C_{i-1} \oplus C_i \oplus \dots \oplus C_n$ be a greedy linear extension of P, and let $i \leq n$.
There exists another greedy linear extension $L' = C'_1 \oplus \dots C'_{i-1} \oplus C_i \oplus \dots \oplus C_n$ of P such that:

1. *Left-Links* (L, C_i) = *Left-Links* (L', C_i)
2. If $Top(C'_k) < u$ and $Top(C'_j) < v$ for some elements u, v in C_i and $u \leq v$, then $j<k$.
3. All chains C'_k in L' where $Top(C'_k) \in$ *Left-Links* (L', C_i) are consecutive just at the left of C_i. That is, if j is the smallest index such that $Top(C'_j) \in$ *Left-Links* (L', C_i) then $Top(C'_k) \in$ *Left-Links* (L', C_i) for every k where $j \leq k < i$

PROOF **:** The algorithm that construct L' consists of performing a series of Chains Interchange operations in order to "move" successively each left-neighbors C_k of C_i one by one to the right until it reaches its "targeted" place. The algorithm starts with the chain C_k which is linked by the smallest element in C_i. At the end, the more a chain is linked by a smaller element, the more it is closer.
Let k be the smallest value such that $Top(C_k) < u$ for some u in C_i. If k = i-1 then we are done since $Top(C_k)$ will be the only element in *Left-Links* (L', C_i) and thus $L'=L$.
Assume that k <i-1and let $IC(L, k) = L_1 = C_1 \oplus \ldots \oplus C'_k \oplus C'_{k-1} \oplus \ldots C_i \oplus \ldots \oplus C_n$
According to the definition of the IC operation, $C'_k = \{x \text{ in } C_k\text{: } x \le v\} \cup \{x \text{ in } C'_{k+1}\text{: } x < u\}$ and $C'_{k+1} = \{x \text{ in } C_k\text{: } x > v\} \cup \{x \text{ in } C'_{k+1} : x \ge v\}$, where $Top(C_k) < u$ for some $u \neq Bottom(C_{k+1})$ in C_{k+1} and $v < Bottom(C_{k+1})$ for some $v \neq Top(C_k)$ in C_k. Because of the unicity of the links that $Top(C_k)$ could have (according to *Lemma 5.8*), and since $Top(C_k)$ is linked to C_i then $Top(C'_{k+1}) = Top(C_k)$ and $Top(C'_k) = Top(C_{k+1})$.
With the single operation $IC(L, k)$ we constructed a new greedy linear extension where the chain with a top $Top(C_k)$ is now C'_{k+1} and therefore closer by 1 to C_i. Performing the operation as many times as it is needed, we can get $Top(C_k)$ to be the top of the chain just beside the Chain C_i.

Recursively, we bring each of the chains with a Top in *Left-Links* (L', C_i) to its wanted position within the greedy linear extension.

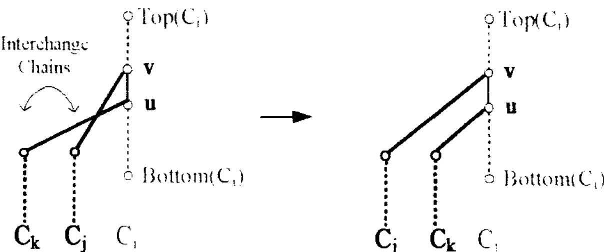


Figure 6.4: The basic technique of improving the left-links of a chain

The *Algorithm 6.2* displays the left-links reorganization of a chain C_i. Its java implementation can be consulted in the Appendix Section at page 114.

```
Arrange-Left-Chains (L: LinearExtension, i: index of a Chain)
Input: A linear extension L such that L = C1 ⊕ ... ⊕ Ci-1 ⊕ Ci ⊕... ⊕ Cn
       An index i which indicates that "left links" of Ci will be arranged.
       An implicit ordered set P which is N-Free and X-cycle-Free.
Output: a new implicit L inducing a new drawing of P
Begin
   start, end : Temporary index values

   Assume Ci = {Bottom(Ci), a, b, ..., Top(Ci)}
   end := i − 1    (End index of pulling)
   For each element e = a, b, ..., Top(Ci)
      For each Ck such that Top(Ck) ≺ e
         For each start = k, k + 1,..., end
            Interchange-Chains (L, start)
         end := end − 1
End
```

Algorithm 6.2: The Arranging-Left-Chains algorithm

The best case is that every left-neighbors chain is fortunately arranged as shown by the *Figure 6.2(b)* and no reorganization is required. In the worst case, the left-neighbors are ordered in the opposite way as "stairs" shown by the *Figure 6.3(a).* In this case, let's assume C_k is the farthest left-linked chain. Then, C_k needs $((i-1)-k)$ *Chains-Interchange* to be sited, C_{k+1} require $((i-2)-(k+1))$ if C_{k+1} is also a left-neighbor; C_{k+2} require $((i-3)-(k+2))$, and so on. Hence, the total number of *Chains-Interchange* operations in the worst case is in $O((i-k)^2)$ such that i,k are the index of the concerned chain and the farthest left-neighbor chain, respectively. Of course, both $k < i$ and $i-k < width(P)$ hold. Consequently, the time complexity is $O((width(P))^2)$ in the worst case.

Note that the three "for" loops does not imply that the algorithm has a cubic time complexity. The first two "for" statement only loops with the same number of left links. The reason is that the top of a chain is only linked once and consequently, each left-chain is only pulled "once" during its entire interchanging process. An example of execution for an arbitrary index i is displayed by the *Figure 6.5.*

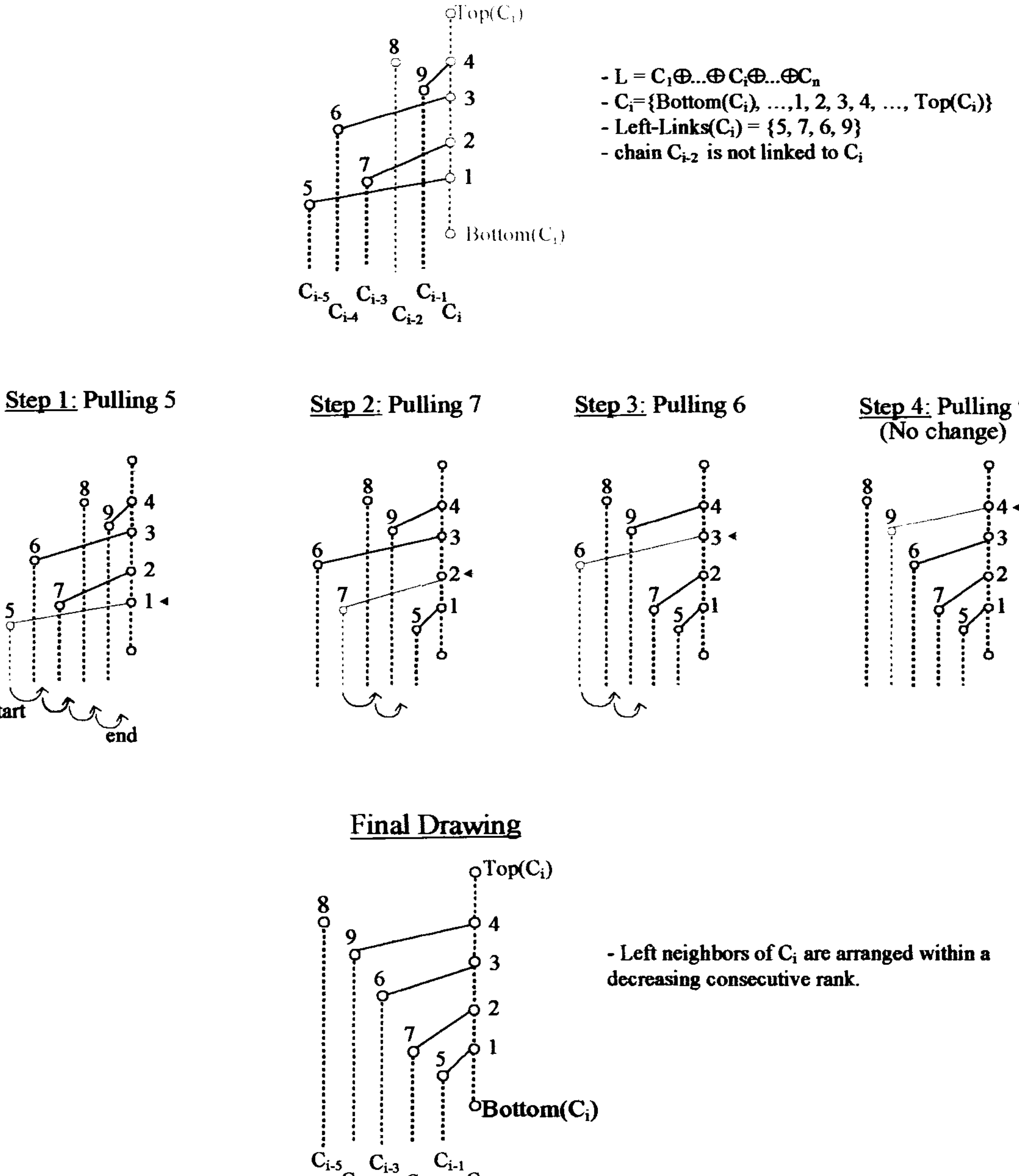


Figure 6.5: An execution sample of the Arrange-Left-Chains algorithm

6.2.2. The right neighbors chains restructuring

The right case configuration is intuitively derived from the left-case one. The right-neighbors chains have to be ordered as displayed by the *Figure 6.2(c)*. Similarly

to the left case, the algorithm also needs to "pull" successively each right neighbor C_k till it reaches its "targeted" place. The general formula for the position is as follows: if C_k is linked by the "d^{th} element of C_i having right-link" then the targeted place of C_k is the position $(i+d)$. Of course, an immediate chains-exchange is not feasible. Consequently, $((i-d)-k)$ *Chains Interchange* from the index k to the index $(i-d)$.

The formal definition of the desired configuration is also established similar to the left case. Formally, let $L = C_1 \oplus C_2 \oplus \ldots \oplus C_i \oplus \ldots \oplus C_n$ be a greedy linear extension of *P*, and let i≥1. We define the set *Right-Links (L, C_i)* as the of set elements *x* in P, such that *x*=Bottom(C_k) for some k>i and *x* is covering some element in C_i. The reorganization of the right-neighborhood of a chain C_i in a greedy linear extension can be established as follows:

Proposition 6.2 Let P be a finite ordered set which is *N-Free* and *X-cycle-Free*. Let $L = C_1 \oplus \ldots \oplus C_i \oplus C_{i+1} \oplus \ldots \oplus C_n$ be a greedy linear extension of *P*, and let i≥1. There exists another greedy linear extension $L' = C'_1 \oplus \ldots C'_i \oplus C_{i+1} \oplus \ldots \oplus C_n$ of *P* such that:

1. *Right-Links (L, C_i) = Right-Links (L', C_i)*
2. If $Bottom(C'_k) > v$ and $Bottom(C'_j) > u$ for some elements *u*, *v* in C_i and $u \leq v$, then $k<j$.
3. All chains C'_k in *L'* where $Bottom(C'_k) \in$ *Right-Links (L', C_i)* are consecutive just at the right of C_i. That is, if *j* is the greatest index such that $Bottom(C'_j) \in$ *Right-Links (L', C_i)* then $Bottom(C'_k) \in$ *Right-Links (L', C_i)* for every *k* where $i<k\leq j$

Therefore, the algorithm is dually established as like the left case. The *Algorithm 6.3* is the pseudo code and the java implementation can be consulted in the Appendix Section at page 114.

Arrange-Right-Chains (L: LinearExtension, i: index of a Chain)
Input: A linear extension L such that $L = C_1 \oplus \dots \oplus C_i \oplus C_{i+1} \oplus \dots \oplus C_n$
An index i which indicates that "right links" of C_i will be arranged.
An implicit ordered set P which is N-Free and X-cycle-Free.
Output: a new implicit L inducing a new drawing of P
Begin
Assume $C_i = \{\text{Bottom}(C_i), \dots, y, z, \text{Top}(C_i)\}$
$End := i + 1$ (End index of pulling)
For each element $e = z, y, \dots$ Bottom(C_i)
For each C_k **such that** $e \prec Bottom(C_k)$
For $start = k, k-1, \dots, End$
Interchange-Chains (L, $start$-1)
$End := End + 1$
End

Algorithm 6.3: The Arranging-Right-Chains algorithm

The best case is that every right-neighbors chain is already arranged as the desired configuration and no reorganization is required. In the worst case, the right-neighbors are ordered in the opposite way shown by the *Figure 6.3(b).* The time complexity of this algorithm is dually established following the left case. We conclude that the total number of *Chains-Interchange* operations in the worst case is in $O\left((k-i)^2\right)$ such that k, i are the index of the farthest right-neighbor chain and the concerned chain, respectively. Of course, both $i < k$ and $k - i < width(P)$ hold so the time complexity is $O((width(P))^2)$ in the worst case

An example of the algorithm's execution for an arbitrary index i can be induced by the left-case example (*Figure 6.5*) by just inversing the order.

6.2.3. The local picture enhancement algorithm

Finally, we propose the *Algorithm 6.4* that organizes the neighborhood of one particular chain C_i. Its aim is to merge the precedent techniques of enhancing the left and right neighborhood. The *Arrange-Left-Chains* (*Algorithm 6.2*) is called if the left neighbors exist, that is C_i is not the first chain in the drawing. Conversely, the *Arrange-Right-Chains* (*Algorithm 6.3*) is called if C_i is not the last chain in the drawing. The java code can be consulted in the Appendix section at page 114.

```
Arrange-Chain (L: LinearExtension, i: index of a Chain)
Input: A linear extension L such that L = C1 ⊕ ... ⊕ Ci-1 ⊕ Ci ⊕... ⊕ Cn
       An index i which indicates that "left links" of Ci will be arranged.
       An implicit ordered set P.
Output: a new implicit L inducing a new drawing of P
Begin
    If i ≠ 1 then
        Arrange-Left-Chains(L, Ci)
    If i ≠ n then
        Arrange-Right-Chains(L, Ci)
End
```

Algorithm 6.4: The Arranging-Chain algorithm

The time complexity is induced from the analysis done in the previous section. If C_i has n left-neighbors and m right-neighbors, then the arrangement of the picture around C_i will involve $O(n^2+m^2)$ interchanging. Of course, $n+m \leq width(P)$. Hence, we can conclude that this improvement has a time complexity of $O((width(P))^2)$ in the worst case.

6.2.4. Crossings and Limitations.

We now analyze the limits of our reorganization technique in terms of numbers of crossings with respect of the chain C_i.

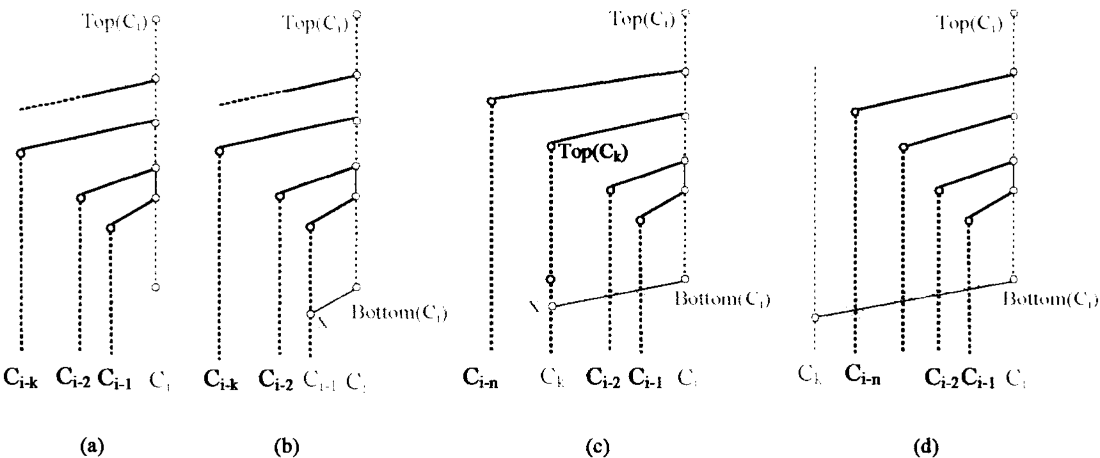


Figure 6.6: The crossing cases for the left-neighbors chains.

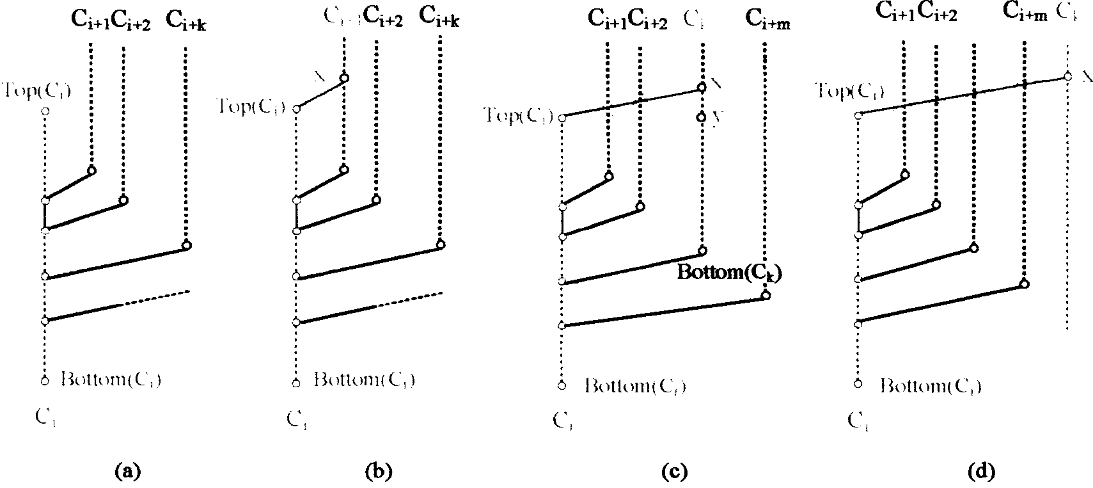


Figure 6.7: The crossing cases for the right-neighbors chains

a. Edges "directly" linked with C_i.

Let's analyze the left picture of C_i with respect to the number of crossings. Assume that the left-neighbors are well ordered as proposed. The crossing may take

place depending on the "type of edge" coming towards $Bottom(C_i)$. The possible situations are shown in *Figure 6.6* where we can distinguish 3 major cases:

- $Bottom(C_i)$ is free (*Figure 6.6(a)*); that is $Bottom(C_i)$ is a minimal element of *P*. Crossing does not occur.
- $Bottom(C_i)$ is linked to one left-neighbor. There are two sub cases:
  - The best scenario is that $Bottom(C_i)$ is linked to its closest neighbor (*Figure 6.6 (b)*); that is $x \prec Bottom(C_i)$ such that $x \in C_{i-1}$. Crossing does not occur.
  - $Bottom(C_i)$ is linked to a left-neighbor C_k such that $i-n \leq k < i-1$ where n is the total number of the left-neighbors (*Figure 6.6 (c)*) . In this case, there will be $(i-k-1)$ crossings.
- $Bottom(C_i)$ is linked to external chain C_k (*Figure 6.6(d)*). Thus, $k < i-n$ and all the left-neighbors chains is crossed. There will be at least n chains crossed and at most $(i-2)$.

An analysis of the right-neighbors is done symmetrically. The *Figure 6.7* shows the different cases. Here, the element $Top(C_i)$ plays the special role in comparison with the left-case's $Bottom(C_i)$. Once again, crossing is avoided if $Top(C_i)$ is a maximal element of *P* (*(a)*) or linked with the closest right-neighbor C_{i+1} (*Figure 6.7(b)*). If C_i is linked with a right-neighbor C_k then there will be $(k-i-1)$ crossings (*Figure 6.7(c)*). Finally, in the worst case, $Bottom(C_i)$ is linked with an external chain C_k such that $k > i+m$ where m the total number of right neighbors of C_i. In this case, all right chains will be crossed and hence, the minimum number of chains which are crossed is m and the maximum is $(width(P)-i-1)$.

In many cases, many crossing of this type is mostly unavoidable. For instance, consider the previous scenario presented by *Figure 6.7 (c)* where $Top(C_i)$ is linked with a right-neighbor C_k through the element *x*. Moreover, Assume C_i is linked with

left-neighbor C_{i-y} through an element z. Attempt to solve the crossings is illustrated by the *Figure 6.8*:

- The first solution (*Figure 6.8(b)*) is to pull C_k closer to C_i by consecutively interchanging C_k with the chains located between indexes $(k-1)$ and $(i+1)$ in decreasing order. Unfortunately, the new C_k, denoted $*C_k$, will cross all left-to-right edges linking C_i and its right-links greater than a. Hence, the crossing number remains invariant.
- The second solution (*Figure 6.8(c)*) is to directly flip the chains C_k with C_i. Even if they are not consecutive, let's assume that the flipping is not violating the greediness of L. Hence, the old crossing is now solved. However, the left-to-right edge ($Top(C_i) \prec z$) is now crossing the new chain $*C_i$. Moreover, several new crossings may appear, depending on the number of right links of $*C_k$ greater than c and smaller than x.

Thus, these type of crossings given rise by $Bottom(C_i)$ and $Top(C_i)$ are the "limitations" of our technique. However, we are confident that the proposed configuration, as introduced in *Figure 6.2(c),* constitutes a "good compromise" over the presence of this category of crossings.

(a) Drawing after the local improvement technique

(b) Solution #1

(c) Solution #2

Figure 6.8: The dilemma on solving the crossing around a particular chain

b. Edges "indirectly" linked with C_i.

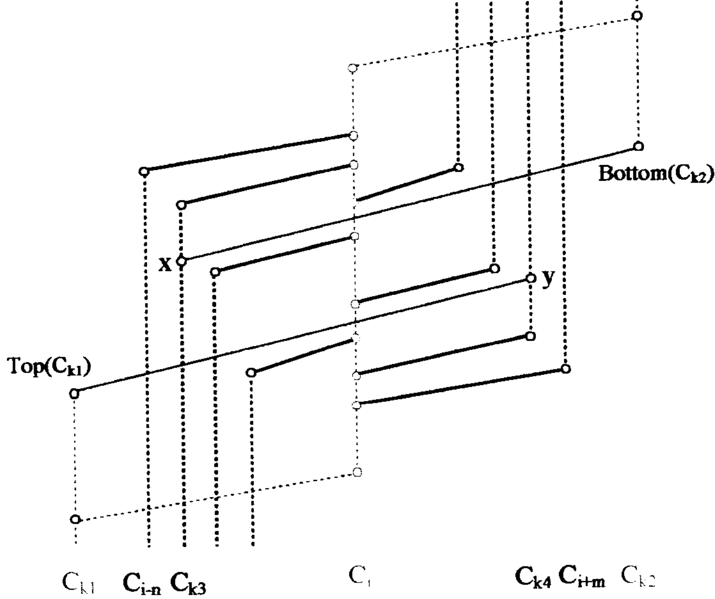


Figure 6.9: The worst case configuration of the chain neighborhood

Let's now consider the crossing involved with the neighbors of C_i. This is the list of the possible scenario:

- The left neighbors are linked between themselves. For instance, in the *Figure 6.6(d)*, if *Bottom*(C_{i-2}) is linked with C_{i-n}, then all C_k included between them will be crossed (i.e. $i-n<k<i-2$). However, the resolution of this crossing is more related to the resolution of the right neighborhood of C_{i-n}.
- The right neighbors are linked between themselves. For instance, in the *Figure 6.7(d)*, if *Top*(C_{i+2}) is linked with C_{i+m}, then all C_k included between them will be crossed (i.e. $i+2<k<i+m$). However, the resolution of this crossing is more related to the resolution of the left neighborhood of C_{i+m}.
- The left-neighbors cannot be linked immediately with the right neighbor since every left-neighbor's *Top* and every right-neighbor's *Bottom* are all linked with C_i.
- The only type of crossing left, illustrated by the *Figure 6.9*, is considered as the worst configuration. *Bottom*(C_i) is linked with an extern chain C_{k1}, and so is

Top(C_i) with an extern chain C_{k2}. Then, *Top*(C_{k1}) can be linked with any chain on its right (except C_i), say C_{k3}. Conversely, C_{k2} can be linked with any chain on its left (except C_i), say C_{k4}. Consequently, the LR edge ($Top(C_{k1}) \prec$ y) may cross every vertical chain located between C_{k1} and C_{k4}. The same situation applies with the LR edge (x $\prec Bottom(C_{k2})$) which may cross every vertical chain sited between C_{k3} and C_{k2}. The number of crossings varies according to the respective location of the elements x on C_{k3} and y on C_{k4}. At most, this value is $(k4-k1-1)+(k3-k2-1)$. Once again, this type of crossing is another limitation of our techniques.

One of the main advantages of this technique is C_i's independence over the remaining chains which are not linked to it. The reason is that *Arrange-Chain* can be called on any chain of *P* and will only act on chains linked to it. For instance, if we are particularly interested in one specific chain, this technique will allow us to clarify its neighborhood and temporarily ignore the remaining chains. On visualization software, this routine can also be easily integrated as a small functionality that bring local enhancement on a specific chain of which the user has a specific interest.

6.3. Overall View Improvement

Keeping in mind the limitations mentioned previously, we now investigate the global improvement strategy. We present the idea of "how to use" the precedent techniques in order to acquire a better visual perception of the overall drawing.

6.3.1. Difficulties and Problematic

In the ideal case, each chain of the drawing has its neighbors well organized. Unfortunately, it is not just a matter of calling the "*Arrange-Chain*" method for each chain of the drawing in an arbitrary order. The *Figure 6.10* shows the magnitude of the combinatorial call order of the method. It shows three different tentative of improving the overall view of an ordered set by successively calling the technique *Arrange-Chain* on each chain. For instance, the *Figure 6.10 (a)* shows the ideal

picture but the remaining two do not succeed to produce it. Hence, the first problem raised was about the invocation order of the *Arrange-Chain* method and its combinatorial issue.

However, we realize the problem is more than combinatorial. Indeed, considering the *Figure 6.10 (c)*, the ideal picture is obtained just after the second call. Consequently, the ideal picture may be satisfied by only calling *Arrange-Chain* on the appropriate chain and which will avoid extra unnecessary operations. The ideal picture (3rd picture) is even again destroyed by the call of another *Arrange-Chain* on the chain C_1 (last picture). This is a major problem because it means a call of *Arrange-Chain* can deteriorate another chain which is already arranged.

Moreover, concerning this same example of *Figure 6.10 (c)*, the final picture produces one crossing. But, a last recall of the *Arrange-Chain* on chain C_2 will provide the ideal drawing as in *Figure 6.10 (a)*. In other words, even if the ideal picture is not obtained after calling *Arrange-Chain* on each chain, it can be fulfilled by an extra-call. Hence, this case demonstrates another problem which is the decision of when to stop calling the *Arrange-Chain* method.

In addition, consider the first two pictures of the *Figure 6.10 (b)*. As we can see, both drawing are identical. Indeed, the call *Arrange-Chain* on the chain C_2 did not influence on the drawing. This situation is problematic since the call could be avoided to not perform extra operations.

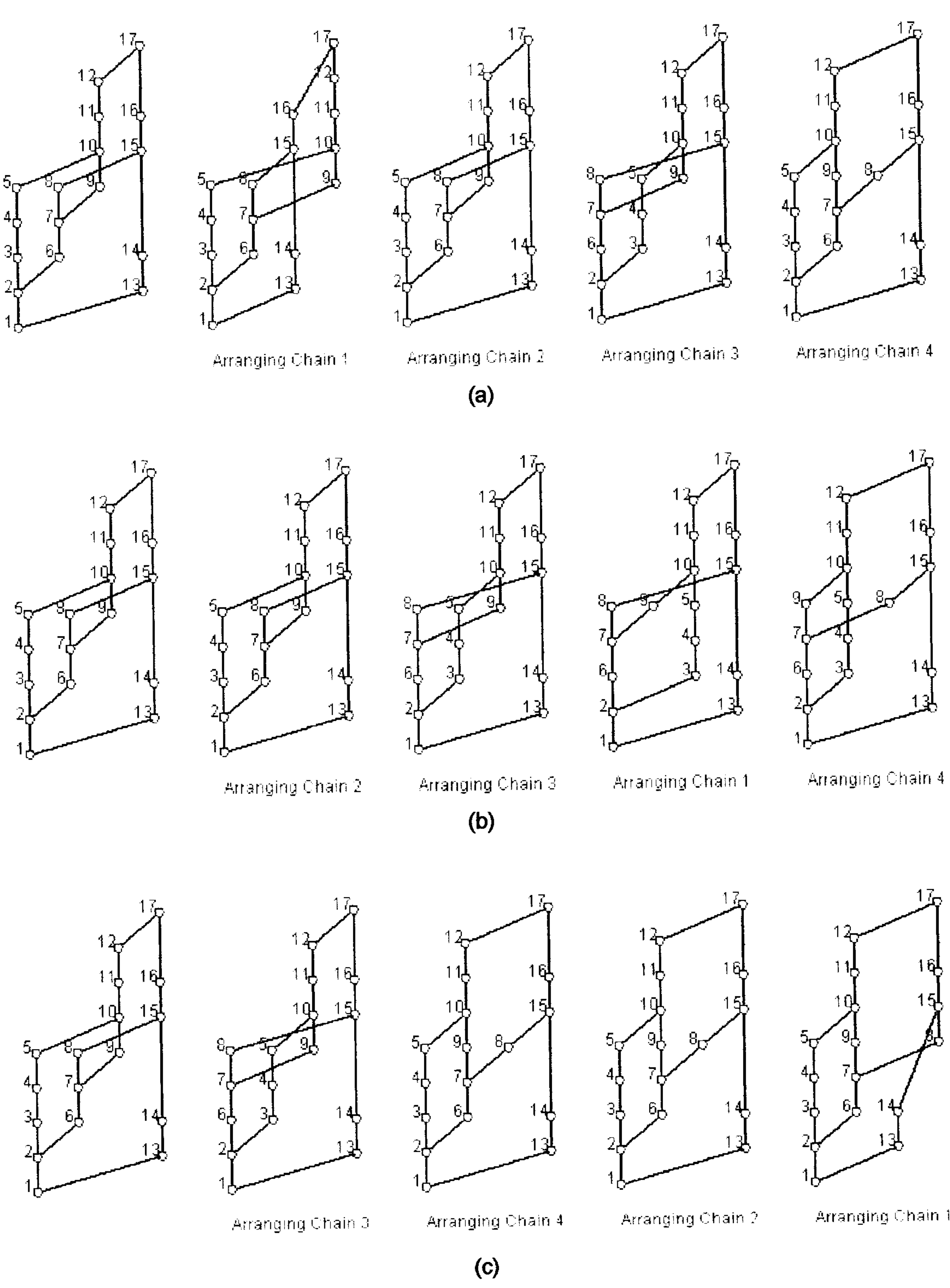


Figure 6.10: Example of tentative picture improvement by successive arrangement of each chain.

6.3.2. The global picture enhancement algorithm

One of the main issues presented previously is that the call of *Arrange-Chain* on a chain C_i may deteriorate the neighborhood of a chain C_j which is already arranged. However, two properties are fundamental: the call of *Arrange-Left-Chain* on a chain C_j will never conflict with the left-neighborhood of a chain C_i already arranged as long as C_j is on the left of C_i (i.e. $j < i$). Conversely, the *Arrange-Right-Chain* on C_j will not deteriorate the right-neighborhood of C_i as long as $j > i$.

a. *The heaviest chain idea*

Accordingly, our idea is to use these properties after finding an ideal index k that splits our drawing in two parts composed by the left-picture and the right-picture. Such k should have the most connections in terms of left-neighbors and right-neighbors. This choice has been made on the motivations that a chain C_k which owns lots of neighbors eventually creates lots of crossing around it. Based on this reason, firstly our technique computes the heaviest C_k.Then it arranges the left-neighborhood "decreasingly" from the index k to the *first chain* and the right-neighborhood "increasingly" from the index k to the *last chain.* The index k of the heaviest chain C_k can be computed beforehand during the initial drawing of *P*. Otherwise, it can be fulfilled by another independent function which, firstly selects every chain from *L one by one,* secondly calculates the number of its left-neighbors and its right-neighbors and thirdly computes the maximum.

b. *The algorithm and its time complexity*

The *Algorithm 6.5* implements this idea. The java-code can be consulted in the Appendix section, at page 116. Since the calculation of the number of neighbors requires at most $O(height(P))$ operations for one chain, $O(width(P)*height(P))$ operations are then needed to find the index k. The subroutines *Arrange-Left-Chains* and *Arrange-Right-Chains* being in $O((width(P))^2)$ for each chain, the algorithm has a time complexity of $O((width(P))^3)$.

```
Arrange-All-Chains (L: a linear-extension)
Input: A linear extension L = C1 ⊕ ... ⊕ Ci-1 ⊕ Ci ⊕ Ci+1 ⊕... ⊕ Cn
        An implicit ordered set P.
Output: a new implicit L inducing a new drawing of P
Begin
    Find the heaviest index k that split the initial drawing.
    For i = k, k-1,...,2
        Arrange-Left-Chains (L, Ci)
    For i = k, k+1,...,n-1
        Arrange-Right-Chains (L, Ci)
End
```

Algorithm 6.5: The Arranging-all-Chains algorithm.

6.4. The N-Free drawing algorithm

The drawing algorithm for N-Free ordered sets is a refinement of the general algorithm presented in *Chapter 4* which incorporate the global enhancement technique (*Arrange-All-Chains* algorithm).

The first step which remains unchanged from the general algorithm (*Algorithm 4.2*) is to generate an arbitrary linear extension L of P. The second step, which did not exist in the general algorithm, is to arrange the chains stored in L to satisfy the needed configuration. The last step is to place the chains on the screen. The difference between strategies of placing the elements against the general order algorithm resides on the sense that only Bottom and Top elements are necessarily the only links that stands between any two pairs of chains. Consequently, assume a chain C_i = {*Bottom*(C_i), a, b, ..., *Top*(C_i)}. The y-coordinate of *Bottom*(C_i) is the y-coordinate of its unique immediate predecessor increased by 1 or 0 if *Bottom*(C_i) is a minimal in P. Since the remaining elements: a, b, ..., *Top*(C_i) can be only linked by Top of any left chains of C_i, their y-coordinate depends on the set of Top-elements that they covered (instead of computing coordinate of all predecessors elements suggested by the general algorithm). In the pseudo-code, we use a variable called *Unlinked-Max* which keeps track of the set of these Top-elements.

```
NFree-Drawing-Algorithm(P: ordered set)
Input: an ordered set P which is N-Free and X-cycle Free
Output: a drawing of P in a 2D Cartesian plan
Begin
   Unlinked-Max: Set of maximum element of chain which are not yet linked.
   Actual-y-Coordinate: Temporary variable to track the actual y coordinate
   L: the greedy linear extension of P

   Step1:  L:= Generate_A_Greedy_Linear_Extension(P)

   Step2:  Arrange-All-Chains(L)

   Step3:  Draw chains of L on screen according to both properties that:
             1) (Top to *) and (Bottom to *) are the only left-to-right edges which stand
             2) these links are unique "from Top" or "to Bottom"
   Unlinked-Max:= {}
   Assume L = C1 ⊕ C2 ⊕ ... ⊕ Cn
   For each chain Ci  where i=1, 2… n
       Assume Ci = {Bottom(Ci), a, b, …, Top(Ci)}
       if Bottom(Ci)  covers an element α then
           y-coordinate of Bottom(Ci) := (y-coordinate of α) + 1
       else
           y-coordinate of Bottom(Ci) := 0
       Place Bottom(Ci) at the position x=i  and y= y-coordinate of Bottom(Ci)
       if α exists then
           Draw edge (α ≺ Bottom(Ci))                    // link (Bottom to *)
       Actual-y-Coordinate := y-coordinate of Bottom(Ci)
       For each element e of Ci  where e=a, b, … Top(Ci)
           if e covers some sets of elements X such that X ⊆ Unlinked-Max then
               y-coordinate of e := Maximum (y-coordinate of elements in X) + 1
               Remove X from Unlinked-Max
           else
               y-coordinate of e := Actual-y-Coordinate + 1
           Place e at the position x=i  and y= y-coordinate of e
           Draw edge (el ≺ e)  where el ≺ e  in Ci            // vertical edge
           Draw edges (x ≺ e) for each x∈ X                   // link (Top to *)
           Actual-y-Coordinate = y-coordinate of e
       Add Top(Ci) to Unlinked-Max
End
```

Algorithm 6.6: The LR-drawing algorithm of N-Free X-cycle free ordered sets.

The *Algorithm 6.6* shows the pseudo code. *Step 1* was already studied to run with O(n**width*(*P*)) operations. *Step 2* requires $O((width(P))^3)$ operations. Even though comparisons are decreased, *Step 3* still has an upper bound of $O((width(P))^2)$ operations. We conclude that the time complexity of the algorithm is $O((width(P))^3)$.

6.4.1. Example of Execution and Comparisons

The *Figure 6.11* shows an execution of the *Arranging-All-Chains* technique (*Algorithm 6.5*) applied on an arbitrary greedy *L* of *P*. Thus, the resulting linear extension *L* of the *Step 1* of the *Algorithm 6.6* is represented by the picture (a). The result of the *Step 2* and the assignation of coordinates of all elements of *P* are represented by the picture (c).

This example of *P* contains 79 elements where its initial drawing has 29 crossings. The first stage of the algorithm execution is to compute the heaviest chain, C_7, which has 10 neighbors. The following steps are to improve the left picture on the left of C_7 (*Figure 6.11(b)*), followed by the improving of the right picture of C_7 (*Figure 6.11 (c)*). This last picture represents the end of execution of our algorithm. As we can see, the crossing represented by edges linking fourth chain ($3 \prec 4$) and first chain {1, 2, 5, 6} shows the limitations previously discussed. A slight improvement of the visual perception, without decreasing the number of crossing, can be seen in *Figure 6.12(a)*. In addition, the crossing between the edges ($52 \prec 61$) and ($69 \prec 70$) can be eliminated by interchanging last two chains. (See *Figure 6.12(b)* as a resulting picture). However, these two manipulations were not automatically processed but can be achieved interactively.

Thus, the picture that our algorithm proposes is the *Figure 6.12 (b)* which presents two crossings and satisfying our proposed neighborhood configuration. As noticed, the final picture is aesthetically better than the initial one.

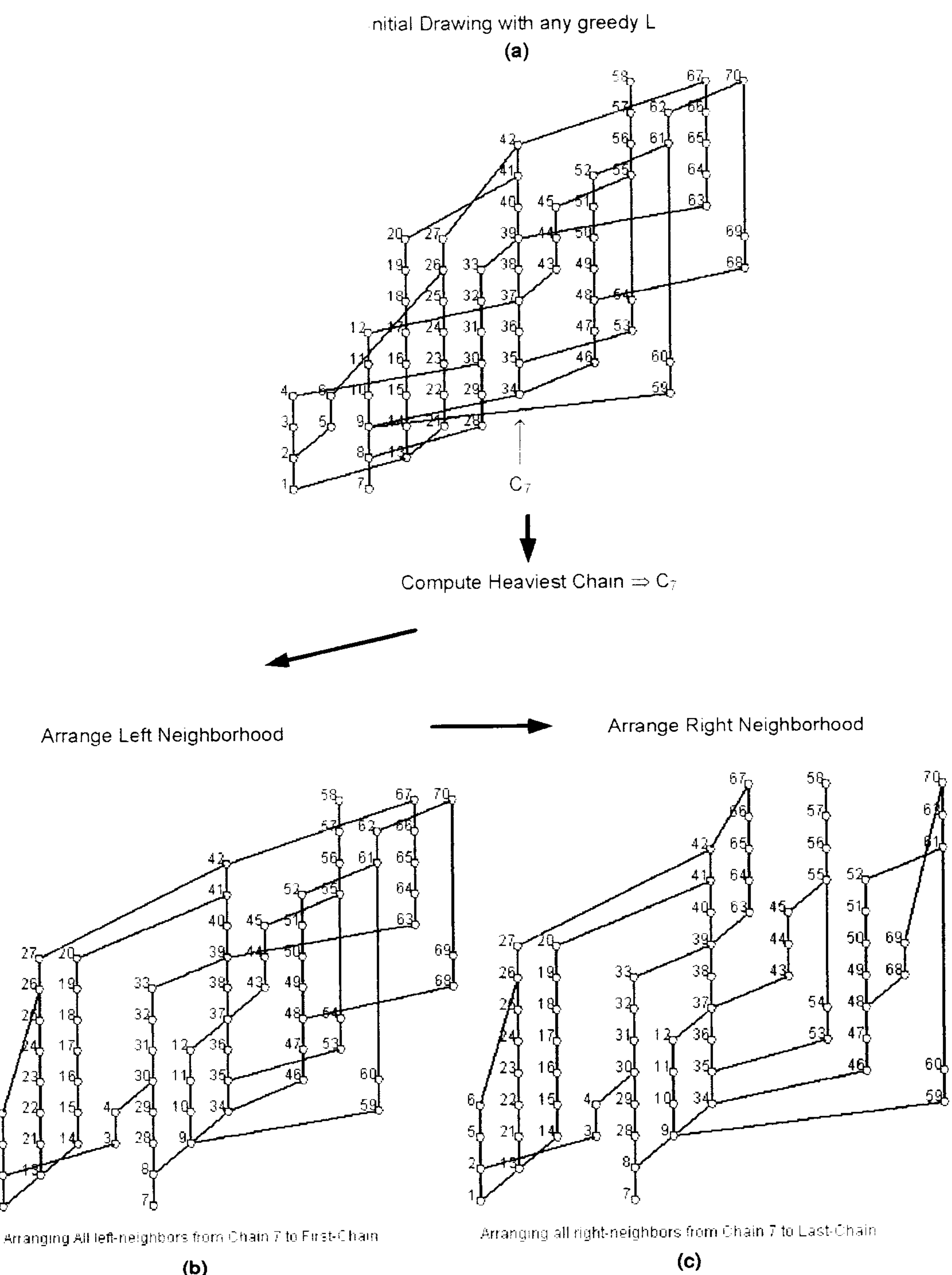
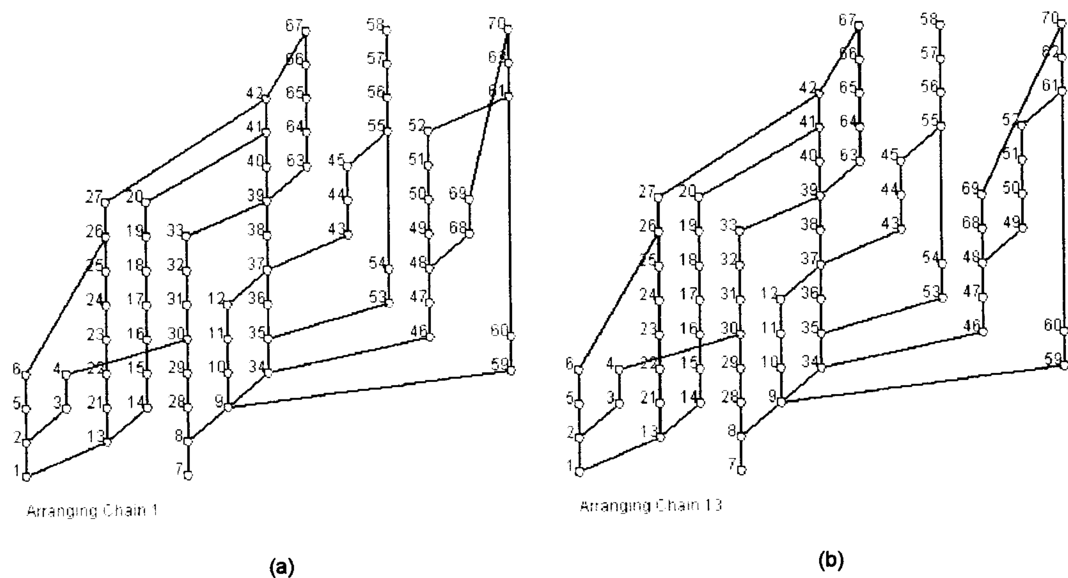


Figure 6.11: A sample execution of the Arrange-All-Chains algorithm.



(a) (b)

Figure 6.12: Sample of additional operations improving the picture

As a title of comparison, the picture of the same ordered set, presented above and produced by similar systems using different algorithms, can be seen in the Appendix section at page 117. The top-right picture is produced by the software *Lattice Draw* using a *Force-Directed* approach. The bottom-left and bottom-right pictures are produced by the software *AGD-GraphWin* using a *k-level Hierarchy* strategy. Explanation about these software and their approach were described in *Chapter 3*. Finally, the top-left is the embedding produced by our system. The ordered set being N-Free, we can notice the clear improvement brought by our drawing.

Additional pages of the same Appendix also show different comparisons of other ordered set pictures.

6.4.2. Planarity, Symmetry and Space used.

In the preceding subsection, we presented a sort of configuration of a good picture around a specific chain C_i. We presented respective algorithms of reorganizing its left-neighbors and right-neighbors and the limitations in terms of number of crossings. With respect to these limitations, we presented an algorithm that

fulfills an overall improvement of the drawing. Even if our algorithm do not guarantee a planar embedding of a planar ordered set, the configuration that we are trying to fulfill clearly shows considerable reduction of crossings presence. This result is convincing since we can declare the next proposition regardless of the situation:

Proposition 6.3: Testing Planarity and Crossing-minimization for an N-Free ordered set diagram is as hard as the general ordered set.

PROOF : We know that testing upward planarity for a general ordered set is NP-Complete. In addition, crossing minimization is NP-Hard. Since a general ordered set can be converted into an N-Free in a polynomial time, it is expected that testing planarity and minimizing crossing of an N-Free ordered set are as hard as for a general ordered set.

However, some attempts of clearing up the planarity for a subclass of N-Free which is the X-Cycle free were experienced and this next lemma is one of our results:

Lemma 6.1 Let P be an ordered set which is N-Free and X-Cycle-Free. If $Width(P) \leq 3$ then Upward Planarity of the LR-Drawing of P is always satisfied.

PROOF : The maximum number of links being finite (here equal to 4), an exhaustive representation of all possible cases of links between the chains were established. Clearly, for each of them, planarity is always satisfied by chains interchanging.

We also note that our algorithm do not satisfy a strict symmetry of the drawing. However, the configuration of drawing (left neighbors and right neighbors) demonstrates symmetrical aesthetics. In addition, an attempt to satisfy the symmetry can be accomplished by alternating the chains by positioning them to the left and then to the right (instead of going from left to right). But in this case, our drawing will not satisfy the flow of comparability which is our main motivation.

We also recall that our drawing of P is a result of the chain decomposition coming from a greedy linear extension L of P. P being N-Free, L is guaranteed to be

jump-optimal and consequently, P is decomposed into the minimum number of chains. Our drawing improvement, even requiring transformation of L and is performed by interchanging the chains, also guaranteeing the same number of chains. Thus, the width of our drawing remains unchanged which is then equal to $width(P)$. We also evoke that the next element that covers another element is directly placed above its Y-Coordinate. The height of our drawing is then equal to $height(P)$. Accordingly, the space used can always be embedded into a rectangle of which its area is $width(P) \times height(P)$. This value slightly increases if edges co-linearity appears and needs to be addressed. Hence, by setting the upper-bound of $width(P)$ and $height(P)$ with the cardinality of P, the space embedding has a complexity of $O(n^2)$.

Chapter 7

Implementation

In this chapter, we discuss the implementation part involved during the development of our algorithms and the creation of an ordered set viewer called the *LR-FlowOrder*. Even though our aim was not to embark on the building of a large scale software, we still approached all drawing functionalities as possible features of visualization software. The goal was to have the most complete implementation to minimize the required in incorporating it into real applications. We used an Object Oriented design where each entity involved (ordered set, Chain, Element, Edge...) is an object. All coding was done with the Java programming language not only because it is an object oriented language, but also because of its vast library of functionality over the graphics components. In our research, we benefited from existing packages such as Java2D over the graphics, Swing over the User Interface and the package java.util.* for different functionalities involving such items as Vector and Hash-table ...

Firstly, we will discuss a preprocessing work which leads us to the creation of a database of ordered sets that we used later on for testing. Then, we will discuss about the implementation of the algorithm itself and the data structure used. Finally, we will analyze the classical "time versus space" ratio.

7.1. Testing and ordered sets database

The first stage of our research involved storing a decent collection of ordered sets that was to be used later for testing our algorithm and for verifying its effectiveness. The creation of this database is divided into two major tasks: the creation of a general ordered set database from some existing graph database and the extraction of N-Free ordered sets from the general ordered set.

7.1.1. Graph database to ordered set database.

As is known, an ordered set can be represented by its diagram which is in itself a "special" graph. Thus, a vertex of a graph can be an element of an ordered set. Conversely, the presence of an edge in a graph can indicate the existence of a relationship in the order. However, a usual graph is not necessarily an ordered set since edge in that graph has to reflect order restrictions such as reflexivity, transitivity and antisymmetry. Nevertheless, the first restriction of reflexivity was ignored in order to increase the size of our database.

Three different graph databases were used to produce our own:

- The first database comes from researchers the University of Brown research lab. The database has 4,128 random graphs and produced 546 ordered sets.
- The second graph database comes from the SIVALab of "University of Naples" in Italy. It is a vast collection of graphs (approximately 200,000) of different kinds and sizes. It was generated for the purpose of establishing a common test ground for benchmarking graph matching algorithms. The extraction produced 3,239 ordered sets.
- The third database comes from a private company that owns data that which reflect properties of an ordered set. The data was in a format that was already valid and, hence no extraction was required. The database contained 10,930 ordered sets.

In the first two databases, the general strategy was to: a) understand the format of the graph; b) create a small program validating the order c) extract the proper graph; d) finally output into our desired format. Since it was a preprocessing task, at that time we did not yet know the optimum format for storage. Consequently, we choose the most standard format: a matrix with binary values where if the position at the line l and column c is *true* means that $l \leq c$ in P. This matrix is commonly defined as an "Adjacency matrix" with directed edges (see *Figure 7.1(b)*). The matrix is saved into a simple ASCII text file with a significant filename which clearly mentions some information about the ordered set. To simplify the testing, all ordered sets were also split into different folders depending on their size.

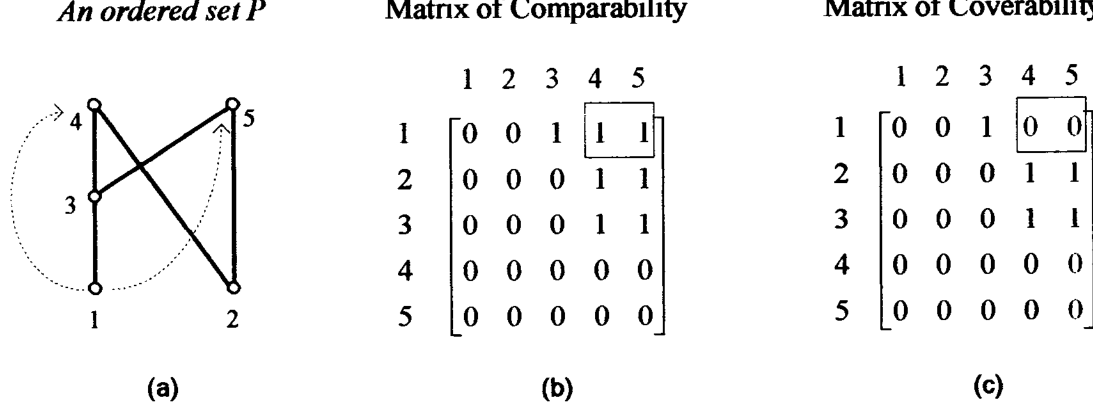


Figure 7.1: Sample of comparability and coverability matrix of an ordered set P.

7.1.2. Extraction of N-Free ordered sets.

After the database was created, a simple program to extract the N-Free ordered sets were fulfilled. Since we only investigated the X-Cycle-Free N-Free, the extracted ordered sets were split into two groups. The first group contained natural N-Free ordered sets and the second one contained N-Free which are also X-Cycle-Free.

7.1.3. Particular Testing

The ordered sets database was useful in verifying if our algorithm worked and produced acceptable drawings. However, in many instances, some special and precise test cases are needed. As a consequence, we had to produce a framework and a program to quickly generate an input ordered set in accordance with our needs.

7.2. Algorithm Implementation and Data Structure

7.2.1. Matrix of Comparability vs Matrix of Coverability.

As previously mentioned, the data structure used during the extraction of the database is a square matrix which validates the comparison between two distinct elements (*Figure 7.1(b)*). On the other hand, the Hasse Diagram is a rendering via the covering and we formulated another version of the same data structure which reflects

the coverability. Hence, instead of using a matrix of comparability, we opted to store the ordered set into a matrix of coverability.

We note that transforming a coverability matrix into a comparability matrix adds a transitive relation. This situation can be solved by the "transitive closure" algorithm of *Floyd-Warshall*. Since we are interested in the inverse process, which is the "transitive reduction", we developed an algorithm that was inspired from the Floyd-Warshall strategy. *Algorithm 7.1* below presents its java-code and has $O(n^3)$ time complexity, similar to the original work that inspired it.

```
* the algorithm is inspired from the Warshall Algorithm which find the
* the transitive closure. Initially, the variable matrixOfCoveringPairs
* is initially a matrix of comparability. At the end of process, it
* will be a matrix of Coverability.
* @author Livaniaina Rakotomalala
* @version 1.1
* @param none
* @see class OrderedSet
*/
public void transitiveReduction() {
   int n = matrixOfCoveringPairs.getNumberOfNodes();

   for(int k=0; k<n; k++)
       for(int i=0; i<n; i++)
           for(int j=0; j<n; j++)
              if(matrixOfCoveringPairs.getElem(i,j) &&
                 matrixOfCoveringPairs.getElem(i,k) &&
                 matrixOfCoveringPairs.getElem(k,j))
                 matrixOfCoveringPairs.setElem(i,j,false);
}
```

Algorithm 7.1: The Transitive-Reduction algorithm in Java

An execution of this algorithm is illustrated by the *Figure 7.1*. The picture *(b)* is comparability matrix of the ordered set P represented by the diagram in *(a)*. The edges represented by arrows are the transitive edges. Finally, the picture *(c)* is the output coverability matrix after applying the transitive reduction to *(b)*.

7.2.2. The java classes Architecture.

Since we used an object oriented approach, each entity involved in our system is represented by an object. Naturally, the principal object is the class *OrderedSet*. We also denote the classes *Element*, *Edge*, *RelationMatrix*, *Chain* and *LinearExtension*. Another package also contains the classes required for the User Interface and some needed utilities. *Table 7.1* below presents a snap shot of the main class. The major methods involved can be found in the *Appendix*.

```
public class OrderedSet {
  private String name;
  private int nbElements, width;
  // data structure
  private RelationMatrix matrixOfCoveringPairs;
  private Hashtable setOfUpperCover = new Hashtable();
  private Hashtable setOfLowerCover = new Hashtable();
  // references
  private Element[] element;
  private Chain[] chains;
  private Vector linearExtension;

  ...

  public OrderedSet(String myName, RelationMatrix myRelationMatrix) {
    name = myName;
    matrixOfCoveringPairs = myRelationMatrix;
    transitiveReduction();
    nbElements = myRelationMatrix.getNumberOfNodes();
    putCoveringIntoHashTable();
    processLinearExtension();
  }
  ...

  public void transitiveReduction() {...}

  private void putCoveringIntoHashTable() {...}

  public void processLinearExtension() {...}

  public void draw(Graphics g) {...}

  public void flipChains(Graphics g, int chainIndex1, int chainIndex2) {...}

  public void arrangeChain(Graphics g, int chainIndex) {...}

  public void arrangeAllChains(Graphics g, int chainIndex) {...}

  ...
}
```

Table 7.1: Template of the principal class Ordered Set.

7.2.3. The N-free ordered set viewer: The LR-Flow Order.

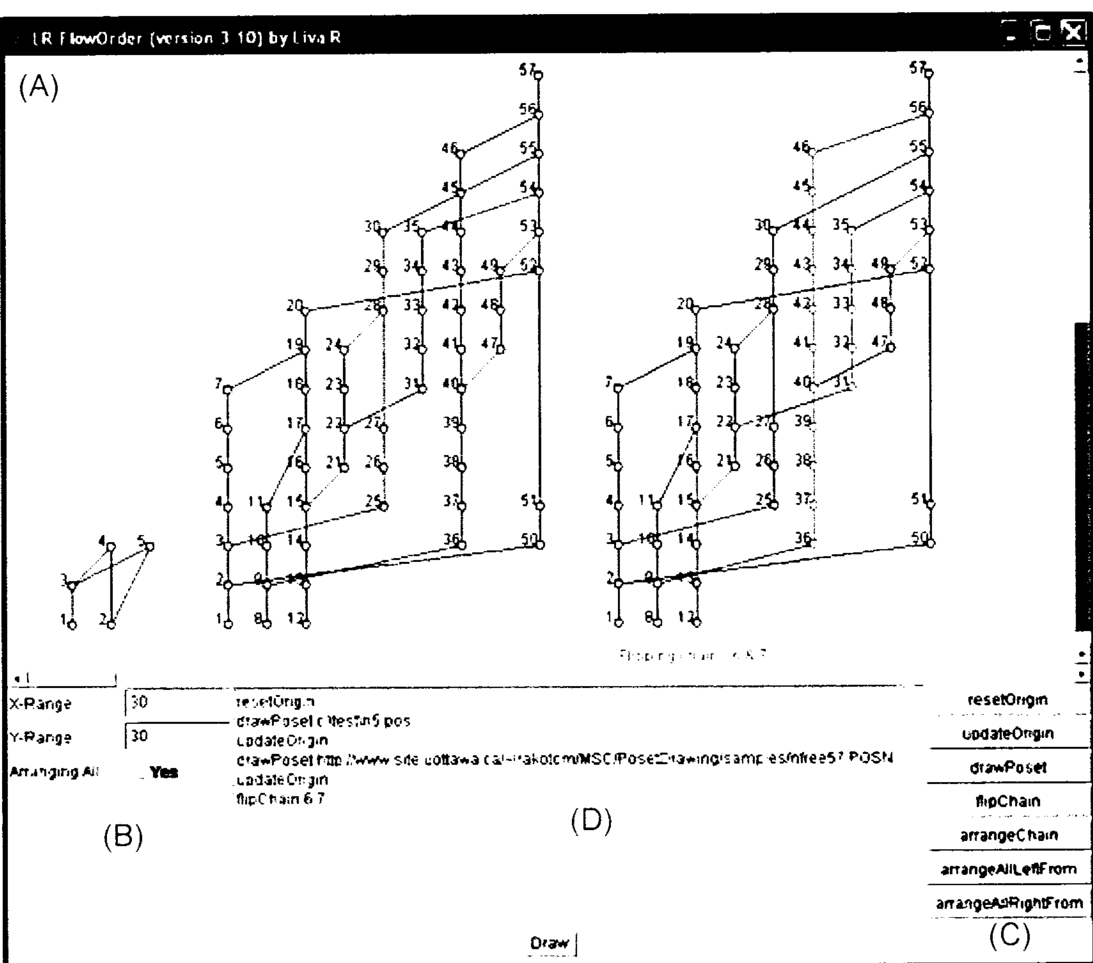


Figure 7.2: The ordered set Viewer GUI

The LR-FlowOrder is an ordered set viewer that implements all the algorithms previously presented. It can simultaneously draw many ordered sets according to the LR strategy. It also presents all enhancement features investigated for N-Free ordered sets, including the local improvement and the global enhancement. The User Interface (*Figure 7.2*) was designed such that different algorithms can be tested quickly. The main advantage of this design is that each algorithm can be tested independently (see *Figure 7.3*). The user can thereby directly interact directly with the algorithms and can view their corresponding effects on the drawing. The GUI is composed of four parts:

- The section (A) is the panel designed for the drawing. It contains both horizontal and vertical scroll-bars so that many drawings of different ordered sets can be embedded simultaneously one after another.
- In section (B), we introduce two options. The first option, X-range and Y-range, are the ranges for the coordinates of the elements and are expressed in pixels. By changing the values, the user can zoom-in the picture to focus on details or zoom-out to concentrate on the larger picture. The second option, a check-Box, is clicked if the user wants to quickly get the final ordered set picture (*Arrange-All-Chains* method).
- Section (C) contains all commands that can be applied to the drawing. Note that each command, except *resetOrigin* and *updateOrigin* executes the algorithm that has the same title. The titles of these commands and their functions are as follows:

    - *resetOrigin* sets the Coordinate Origin to the bottom left of the viewable screen. *updateOrigin* initially updates the coordinate Origin to a new origin which is the bottom right coordinate of the precedent picture. If this command is omitted, the ordered set is drawn on the actual origin. (This can create overlapped pictures)
    - *drawPoset <Filename>* draws the actual ordered set represented by the name <Filename> which is an address of a text file representing a <comparability matrix> or a <coverability matrix>. Note that the file can be a URL address.
    - *flipChain <IndexChain1> <IndexChain2>* flip the chain IndexChain1 with the chain IndexChain2 onto the linear Extensions according to the Chains-Interchange algorithm.
    - *arrangeChain <IndexChain>* arranges the ordered set picture with respect to chain IndexChain.
    - *arrangeAllLeftFrom <IndexChain>* arranges the ordered set picture from right to left starting with chain IndexChain. Conversely, *arrangeAllRightFrom <IndexChain>* arranges the ordered set picture from left to right.
- The section (D) is a command-line version of the section (C) and is useful for executing previously prepared batch commands.

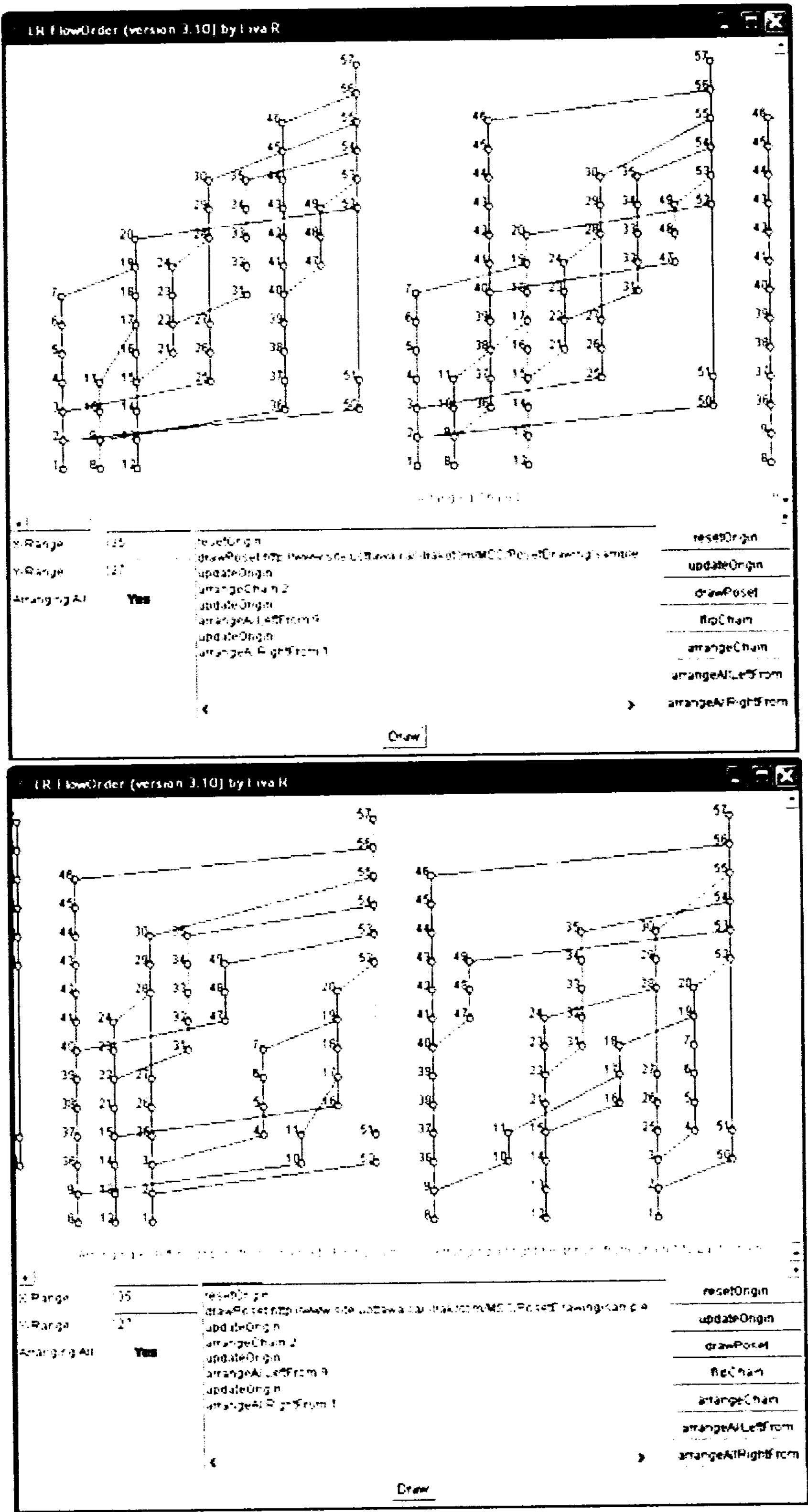


Figure 7.3: An execution sample of the ordered set viewer.

7.3. "Time versus Space" Discussion

The advantages of using a coverability matrix instead of a comparability matrix have previously been explained regardless of these advantages. However, during the implementation, the real data structure used is a Hash-table. All existing relations were saved into a "set of Upper/Lower Covering". The data structure can be seen as a classical graph Adjacency-List, however the elements and its covered/covering elements can be accessed in $O(1)$ instead of navigating into the list. An extra $O(n^2)$ time is needed to preprocess the coverability matrix and produce the Hashtable. The *Algorithm 7.2* shows this preprocessing in java.

```
/*
 * This function transforms the covering matrix into a hashtable.
 * The set of lower cover (upper cover) is saved into a hashtable
 * where the key is the element and the value is a list of all its
 * covering (respectively covered) elements.
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param none
 * @see class OrderedSet
 */
private void putCoveringIntoHashTable() {
  Vector[] listOfUpperCover = new Vector[nbElements];
  Vector[] listOfLowerCover = new Vector[nbElements];

  for (int i=  ; i<nbElements ; i++) {
     listOfUpperCover[i] = new Vector();
     listOfLowerCover[i] = new Vector();
  }
  for (int i=  ; i<nbElements ; i++) {
     for (int j=  ; j<nbElements ; j++) {
        if(matrixOfCoveringPairs.getElem(i,j)==true) {
           listOfUpperCover[i].add(element[j]);
           listOfLowerCover[j].add(element[i]);
        }
     }
  }
  for (int i=  ; i<nbElements ; i++) {
     setOfUpperCover.put(element[i], listOfUpperCover[i]);
     setOfLowerCover.put(element[i], listOfLowerCover[i]);
  }
}
```

Algorithm 7.2: The CoverabilityMatrix-To-HashTable preprocess in Java.

Figure 7.4 shows the hash-table, the set of upper-covering and the set of lower-covering, of the same example presented in the *Figure 1.1*.

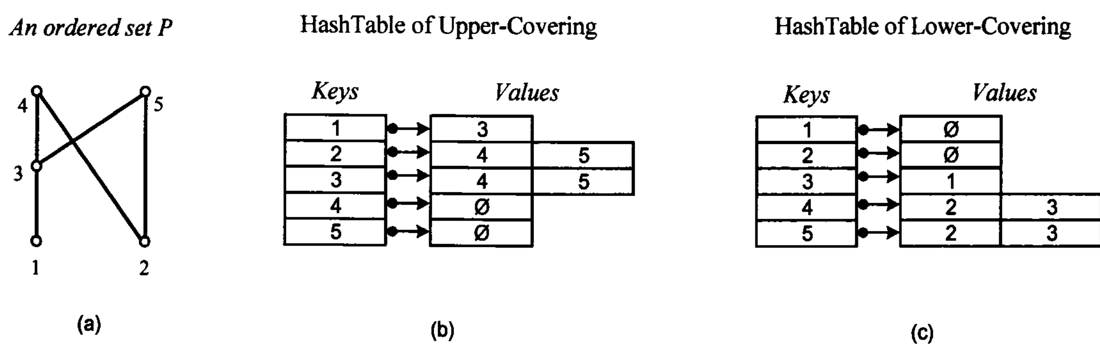


Figure 7.4: The set of Upper-Covering and Lower-Covering into an HashTable

Both *Table 7.2* and *Table 7.3* displays the discussion of how the ratio "time versus space" depends on the data structure used to represent *P*.

Data Structure (*n* represents the number of elements)	Space complexity		Time Complexity of Initial Drawing	
	Needed	Extra	Preprocessing	Needed
Comparability Matrix	$O(n^2)$	$\varnothing$	$\varnothing$	$O(n^4)$
Coverability Matrix	$O(n^2)$	$\varnothing$	$O(n^2)$	$O(n^3)$
HashTable (Upper/Lower Covering)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$

Table 7.2: The ratio time/space on the initial drawing of *P*

For any data structure used, a $O(n^2)$ space is needed to represent *P*. (square matrix). In our case, a hash-table was used to store the set of lower/upper covering and it required in the worst case an extra $O(n^2)$ space to represent the table. If the coverability matrix is used then an extra $O(n^2)$ time is required to produce the coverability matrix from the comparability matrix (transitive reduction). In our case, the hash-table again requires another $O(n^2)$ time to load the table from the coverability matrix (see *Algorithm 7.2*).

Data Structure (*n* represents the number of elements)	Basic Technique	Local Improvement on 1 chain	Global Improvement (**FINAL DRAWING)**
	Chains-Interchange	*Arrange-Chain*	*Arrange-All-Chains*
Ordered set in a Comparability matrix	$O(n^2)$	$O(n^4)$	$O(n^5)$
Ordered set in a Coverability matrix + chains stored in a vector)	$O(n)$	$O(n^3)$	$O(n^4)$
Ordered set in a Coverability matrix + chains stored in a list	$O(1)$	$O(n^2)$	$O(n^3)$

Table 7.3: The ratio time/space on the drawing improvement after the initial picture.

To analyze the final picture after the improvement techniques illustrated by the *Table 7.3*, we first start by the basic routine used by all of our methods. The *Chains-Interchange* technique requires normally $O(n)$ time if the chains are stored in a vector, more precisely O(*height*(*P*)), due to the elements exchange into the array. However, the same operation requires only $O(1)$ if the chains is stored in a list. Consequently, the local improvement of one chain can be fulfilled in $O(n^2)$ time (and more precisely $O((width(P))^2)$), while $O(n^3)$ is needed for a classical coverability matrix. Finally, we can see that the final picture produced by our algorithm involve $O(n^3)$ operations or more precisely $O((width(P))^3)$.

We recall that the Arrange-All-Chains algorithm only rearranges the ranking of elements into the linear extension but do not render the ordered set yet. To fulfill the drawing of *P*, the use of an hashtable were only motivated by the use of the sets of Upper-Covering and Lower-Covering. For instance, the element which is covered by the Bottom of any chain (or the element which covers the Top) is accessed within a single operation.

Chapter 8

Conclusion

8.1. Achievements

Conventional ideas regarding Upward Drawing exist and among them, the most famous are the Hierarchical Layering method, the Force-Directed tactic and the Divide-And-Conquer strategy. In this thesis, we presented a new approach, called the *LR-Drawing*, which uses ordered sets theory and takes advantage of its linear extension. Initially, we first introduced a *new aesthetic metric* where edges can only have bottom-to-top and left-to-right directions. This configuration demonstrates a *flow of comparability* between elements which is useful for quickly confirming the relationship between any two distinct elements. The LR-Drawing algorithm first generates a *greedy linear extension L* which induces a decomposition of *P* into greedy chains. Chains are then placed vertically from left to right in the drawing. The use of chain greediness also brings a maximization of elements that are ranked on the same vertical position. These two new aspects: *comparability flow* and *vertical relations maximization* of the diagram are among of our achievement.

We also presented a subclass of an ordered set, called N-Free, which is treated by our algorithm in an interesting manner. First, we proposed an efficient algorithm (in terms of additional space expenses) to convert any general ordered set into an N-Free one. This achievement is important because the algorithm can be used to draw any general ordered set by primarily converting it to N-Free. Second, we showed that comparisons are decreased during the drawing of this class due to the particularity of the existing links between the chains. However, our major contribution is the formulation of different techniques that improve the drawing of this class. Initially, we proposed some special chain configurations that bring better visual perception and which showed some crossing reduction. Then, we proposed

some techniques to fulfill these configurations via chain interchanging into the greedy linear extension. Other improvement techniques can be used to improve the drawing locally and the overall picture globally. These improvement techniques can be called interactively or systematically to any initial drawing. On a grid, our initial drawing is fulfilled with operations $O(width(P)^2)$. Our global improvement strategy, which carries out our final picture, has $O(width(P)^3)$ time complexity and the rectangle embedding in our picture uses $O(width(P)*height(P))$ area, which is $O(n^2)$ for a large bounding.

We also developed a visualization software package called *the LR-FlowOrder* which automatically draws any ordered set by the use of our algorithms and our picture improvement techniques.

One limitation of the techniques is that they do not guarantee a minimum number of crossings. However, keeping in mind that both Upward-Planarity and Crossing-Minimization are open problems, our algorithm still satisfactorily fulfills the main aim of reducing the number of crossings.

8.2. Future work

The first major work that follows this thesis is the development of a heuristic technique that "minimizes" the crossing (*Arrange-All-Chains* function). Several approaches can be used. One approach is to study the combinatorial aspects of the possible embedding as explained in the last subsection of the previous chapter.

Since we only present a drawing improvement for the N-Free ordered set, a study of this enhancement for a general ordered set can be scrutinized. Indeed, we only presented a general algorithm to draw a general ordered set without drawing improvement. In the same concern, the stretch factor of the LR-Drawing should also be investigated and the resulted picture should be compared to the one from a general upward drawing algorithm. In other words, the question is to know if there is another class, other than N-Free, of which the LR-Drawing produce respectable area.

Also, the N-Free converter presented in this thesis is seen as more efficient than the naïve algorithm in terms of additional space expenses. However, its possible

optimality with respect to the minimum number of dummy elements added can be investigated. If the case fails, a development of a space-optimal algorithm will also be of an interest.

The co-linearity issue, which is not inspected in this thesis due to its graphical specificity, should also be investigated. It has to be inspected in close conjunction with the drawing improvement. The reason for this is that the chain interchanging into the linear extension gives rise to a new drawing of P. The question to be answered is if co-linearity should be solved during the transformation or separately at the end of the drawing process.

Another aspect of the future work does not deal with the drawing but with the theoretical aspect of the planarity of an ordered set. We showed that planarity of an N-Free X-cycle-Free ordered set is always satisfied if $width(P) \leq 3$. A characterization of the non-planarity of the case $width(P) = 4$, induced from case $width(P) = 3$, would be interesting to identify. That may help to characterize ordered sets of this class which are planar. Since any ordered set can be converted into that class, it may bring us closer to the planarity of the general ordered set.

Another future work, and surely innovative, is the use of the "edge elimination" strategy to the LR-Drawing to redefine the concept of an ordered set drawing. Since the vertical relations are implicitly induced, edges joining elements within the same chain can be eliminated. Consequently, the drawing will still contain all elements but only LR edges linking different chains. All aesthetic criteria, especially the concept of crossing, can be redefined with this new concept of drawing. For instance, the planarity of the ordered set will end up deciding if there is a rendering of the LR edges without crossings. Since we have fewer edges without loss of information, not only this new concept brings a less-loaded picture (thus bringing more readability) but most importantly, the complexity of solving aesthetic criteria is considerably reduced.

Bibliography

[AS92] A. Aeschlimann , J. Schmid, Drawing orders using less ink, Order 9, 5-13, 1992

[BBLM96] P.Bertolazzi, G. Di Battista, G, Liotta ,C. Mannino, *Upward drawings of triconnected digraphs*, Algorithmica, 6(12), 476-497, 1996

[BCD+94] P. Bertolazzi, R. F. Cohen, G. G. Di Battista, R. Tamassia, I. G. Tollis, *How to draw a Series-Parralel Digraph*, Internat. J.Comput. Geom. Appl., 4, 385-402, 1994.

[BDBD02] Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo. Quasi-upward planarity. Algorithmica, 32:474-506, 2002.

[BDG97] L. Bianco, P.Dell'Olmo, S.Giordani, *An optimal Algorithm to Find the Jump Number of Partially ordered sets*, Computational Optimization and Applications, 8, 197-210, 1997.

[BTMT98] P.Bertolazzi, G. Di Battista, C. Mannino, R. Tamassia, *Optimal Upward Planarity Testing of Single-Source Digraphs*, SIAM J. Comput., 27, no. 1, 132-169, 1998.

[CGKT97] T. Chan, M. T. Goodrich, S. R. Kosaraju, R. Tamassia, *Optimizing Area and Aspect Ratio in Straight-Line Orthogonal Tree Drawings*, Lect. Notes Comput. Sci. 63-75, Springer-verlag, 1997.

[DN88] G. Di Battista, E. Nardelli, *Hierarchies and planarity theory*, IEEE Trans. Systems Man Cybernet. 18 (6) , 1035-1046, 1988.

[DETT99] G. G.Di Battista, P. Eades, R. Tamassia, I. G. Tollis, *Graph Drawing: Algorithms of visualization of graphs*, Prentice Hall, 1999.

[DTT92] G. G. Di Battista, R. Tamassia, I. G. Tollis, *Area requirement and Symmetry Display of Planar Upward Drawings*, Discrete Compu. Geom., 7, 381, 401, 1992.

[EW94] P. Eades, S. Whitesides, *Edge crossings in Drawings of Bipartite Graphs*, Algorithmica, 11, 379-403, 1994.

[Far97] J.D. Farley, *Chain Decomposition theorems for ordered sets and Other Musings*, DIMACS Series in Discr. Math. and Theor. Comput. Sci., 34, 3-13, 1997

[Fre04] R. Freese, *Automated Lattice Drawing*, Lecture Notes in Artificial Intelligence, 2961, 112-127, 2004.

[FT99] P.C. Fishburn, W.T. Trotter, *Geometric Containment Orders: A Survey*, Order 15, 167-182, 1999.

[GGT96] A. Garg, M. T. Goodrich, R. Tamassia, *Planar upward tree drawings with optimal area*, Internat. J. Comput. Geom. Appl., 6, 333-356, 1996.

[GT04] M.T. Goodrich, R. Tamassia, *Data Structures and Algorithms in Java*, 3rd Edition, John Wiley & Sons Inc, 2004.

[GT95] A. Garg, R. Tamassia, *Upward Planarity Testing*, Order: A Journal on the Theory of ordered sets and Its Applications, 12, 109-133, 1995.

[HT74] J. Hopcroft, R. E. Tarjan, *Efficient Planarity Testing*, J. ACM, 21, 4, 549-568, 1974

[Joh82] D.S. Johnson. *The NP-Completeness column: An ongoing guide*, Journal of Algorithms, vol 3 p 97, 1982.

[Jou95] G. V. Jourdan, *L'analyse d'exécutions réparties en utilisant la théorie de l'ordre*, phd Thesis, Université de Rennes 1, 1995.

[Lin94] C. Lin, *The Crossing Number of ordered sets*, Order 11, 169-193, 1994.

[Riv83] I. Rival, *Optimal linear extensions by interchanging chains*, Proc. Amer. Math. Soc 89, 387-394, 1983.

[Riv84] I. Rival, *Graphs and Order: The role of Graphs in the Theory of ordered sets and Its Applications*, D.Reidel Publishing Company, 1984.

[Riv86] I. Rival, *Stories about Order and the letter N (en)*, Contemporary Mathematics, Volume 57, 1986.

[RZ86] I. Rival, N. Zaguia, *Constructing greedy linear extensions by interchanging chains*, Order 3, 107-121, 1986.

[See98] J. Seemann, *Extending the Sugiyama Algorithm for Drawing UML Class Diagrams*, Lect. Notes in Comput. Sci., volume 1353, 415-424, 1998.

[STT81] K. Sugiyama, S. Tagawa, M. Toda, *Methods for visual understanding of hierarchical systems,* IEEE Trans. Syst. Man. Cybern., 11, 109-125, 1981.

[Tro92] W. T. Trotter, *Combinatorics and Partially ordered sets*, The Johns Hopkins Series University Press, 1992.

Appendix 1:

The Greedy Linear Extension Algorithm in Java

```
/*
 * This function implements the algorithm which generates a greedy
 * linear extension and decomposes it into chains.
 * The algorithm uses a Divide and Conquer strategy (recursive).
 * The   function also uses some extra methods:
 *  - isCoveringAnotherElement(i,min,M) which check if the element i is
 * covering another element than min into the matrix of coverability M[][]
 *  - updateArray(M) which update the matrix of coverability M[][]
 *  - findAMinimum(M) which select an arbitrary minimum into M[][]
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param C[] an array of vectors containing the resulted chains
 * @param chainIndex actual index of a chain, initialized with 0
 * @param min an index telling the actual minimum in the matrix
 * @param min an index telling the actual minimum in the matrix
 * @see public void processLinearExtension() for the call of this algorithm
 * @see class OrderedSet
 */
public void greedyLinearExtension(Vector C[], int chainIndex, int min,
                                  int M[][]) {
   boolean findASuccessorMinimum = false;
   int newMin = -1;

   C[chainIndex].add(element[min]);

   if(! isMaximal(min, M)) {
      for(int i=0; i<nbElements && !findASuccessorMinimum; i++) {
         if(i!=min && M[min][i]==1 && !isCoveringAnotherElement(i, min,M){
            findASuccessorMinimum = true;
            newMin = i;
         }
      }
   }
   updateArray(M, min);

   if(findASuccessorMinimum) {
      greedyLinearExtension(C, chainIndex, newMin, M);
   }
   else {
      chainIndex++;
      newMin = findAMinimum(M);
      if(newMin!= -1)
        greedyLinearExtension(C, chainIndex, newMin, M);
      else
        width= chainIndex;
   }
}

/*
 * This function generate the initial call to the algorithm of greedy
 * linear extension.
 * - the variable chains is an array of Vector where each element of the
 *   vector is an element of P.
```

```
* - the variable linearExtension is a Vector where each element is a chain
* - the variable matrixOfCoveringPairs is 2D square matrix containing
* the covering relation of the ordered set where 1 represent the presence
* of a cover
* the method use a function:
*  - findAMinimum(M) which select an arbitrary minimum into M[][]
*
* @author Livaniaina Rakotomalala
* @version 3.10
* @param none
* @see class OrderedSet
*/
public void processLinearExtension() {
   int M[][] = new int[nbElements][nbElements];  // temporary array
   for (int i = 0; i<nbElements ; i++) {
      for (int j = 0; j<nbElements ; j++) {
         if(matrixOfCoveringPairs.getElem(i,j)==true)
           M[i][j] = 1;
         else
           M[i][j] = 0;
      }
   }
   chains = new Chain[nbElements];
   for(int i=0 ; i<nbElements ; i++) {
        chains[i] = new Chain(i+1);
   }
   // call of the algorithm
   greedyLinearExtension(chains, 0, findAMinimum(M), M);
   linearExtension = new Vector();
   for (int i=0; i<width; i++) {
      linearExtension.add(chains[i]);
   }
}


/*
 * - UTILITY used by the method <greedyLinearExtension(...)>
 * - find the actual minimum available in the ordered set
 * - the next minimum taken is the one with the smallest label which
 * does not have a lower cover.
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 */
public int findAMinimum(int M[][]) {
  boolean done=false;
  boolean goodMin=false;
  int min=-1;

  for(int i=0; i<nbElements && !done; i++) {
     boolean stop=false;
     goodMin=false;
     for(int j=0 ; j<nbElements && !stop ; j++) {
       if(M[j][i]==1) {
         stop = true;
       }
       if(M[j][i]==0 && !goodMin)
          goodMin=true;
       }
       if(stop==false && goodMin==true) {
          done=true;
```

```
          min = i;
        }
      }
      return min;
    }

  /*
   * - UTILITY used by the method <greedyLinearExtension(...)>
   * - check if element of a ordered set become maximal
   * @author Livaniaina Rakotomalala
   * @version 3.10
   */
   public boolean isMaximal(int e, int M[][]) {
     boolean result = true;
         for(int j=0; j<nbElements; j++) {
        if (M[e][j] == 1) {
          result = false;
        }
     }
     return result;
   }

  /*
   * - UTILITY used by the method <greedyLinearExtension(...)>
   * - check if the element <e> is covering another element except <exceptE>
   * @author Livaniaina Rakotomalala
   * @version 3.10
   */
   public boolean isCoveringAnotherElement(int e, int exceptE, int M[][]) {
     boolean result = false;
     boolean stop = false;

     for(int j=0; j<nbElements && !stop; j++) {
       if(j!= exceptE && M[j][e] ==1) {
          stop = true;
          result = true;
       }
     }
     return result;
   }


  /*
   * - UTILITY used by the method <greedyLinearExtension(...)>
   * - assign the value -1 to all elements covering and covered by <min>
   *
   */
   public void updateArray(int M[][], int min) {
     for(int i=0; i<nbElements; i++)
       M[min][i] = M[i][min] = -1;
   }
```

Appendix 2:

The Implementation of the LR-Drawing algorithm in Java

```
/*
  * the implementation of the LR-Drawing algorithm
  * - the variable setOfLowerCover is a set containing all lower cover
  * of any element
  * the method use a function:
  *  - processChainCoordinate which give 2D Cartesian coordinate to
  *    all elements of the chain
  * @author Livaniaina Rakotomalala
  * @version 3.10
  * @param g the graphic
  * @see class OrderedSet
  */
  public void draw(Graphics g) {
    processLinearExtension();
    processChainCoordinate();
    // drawing edges
    for(int i=0; i< element.length ; i++) {
      Vector itsListOfLowerCover = (Vector) setOfLowerCover.get(element[i]);
      for(int j=0; j<itsListOfLowerCover.size() ; j++) {
         Element elemCovered = (Element) itsListOfLowerCover.get(j);
         Edge edge = new Edge (elemCovered, element[i]);
         edge.draw(g);
      }
    }
    //drawing elements
    for(int i=0; i< element.length ; i++) {
      element[i].draw(g);
    }
  }

 /*
  * the method use a function:
  *  - processChainCoordinate which give 2D Cartesian coordinate to
  * all elements of the chain
  * @author Livaniaina Rakotomalala
  * @version 3.10
  * @param none
  * @see class OrderedSet
  */
  public void processChainCoordinate() {
    double actual_x = Coordinate.origin.getXCoordinate();
    double actual_y = Coordinate.origin.getYCoordinate();
    double maxY = actual_y;

    for(int k=0 ; k<chains.length ; k++) {     // k chains
       if(k==0) {
         for(int elementInChain= 0; elementInChain<chains[0].size() ;
             elementInChain++) {
             Element elem = (Element) chains[0].get(elementInChain);
             Coordinate itsCoordinate = new Coordinate (actual_x, actual_y);
             elem.setCoordinate(itsCoordinate);
             actual_y += Coordinate.X_RANGE;
```

```
        }
      }
      else {
        actual_y = Coordinate.origin.getYCoordinate();
        for(int elementInChain=  ; elementInChain<chains[k].size() ;
            elementInChain++) {
            Element elem = (Element) chains[k].get(elementInChain);
            Vector itsListOfLowerCover =(Vector) setOfLowerCover.get(elem);
            Coordinate itsCoordinate;

            if(itsListOfLowerCover.size()== ) {
               itsCoordinate = new Coordinate (actual_x, actual_y);
               actual_y += Coordinate.X_RANGE;
            } else {
               maxY = actual_y;
               for (int itsCoveredElements= ;
                    itsCoveredElements<itsListOfLowerCover.size();
                    itsCoveredElements++ ) {
                  Element elemCovered = (Element)
                  itsListOfLowerCover.get(itsCoveredElements);
                  if(elemCovered.getCoordinate().getYCoordinate()>= maxY) {
                     maxY = elemCovered.getCoordinate().getYCoordinate() +
                            Coordinate.X_RANGE;
                  }
               }
               actual_y = maxY;
               itsCoordinate = new Coordinate (actual_x, actual_y);
            }
            elem.setCoordinate(itsCoordinate);
        }
      }
      actual_x += Coordinate.Y_RANGE;
    }
  }
```

Appendix 3:

The Interchange-Chains Algorithm in Java

```
/*
 * Flip 2 different N-free chains, consecutive or not.
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param g a reference to the drawing space
 * @param chainIndex1 the index of the first chain in the actual drawing
 * @param chainIndex2 the index of the second chain in the actual drawing
 * @see class OrderedSet
 */
public void flipChains(Graphics g, int chainIndex1, int chainIndex2) {
   Element eCoveredByMin =null;
   Element eCoveringMax =null;
   int indexBefore, indexAfter;
   Chain newBefore, newAfter;

   if (chains[chainIndex1].rank<chains[chainIndex2].rank) {
      indexBefore = chains[chainIndex1].rank-1;
      indexAfter = chains[chainIndex2].rank-1;
   } else {
      indexBefore = chains[chainIndex2].rank-1;
      indexAfter = chains[chainIndex1].rank-1;
   }

   Vector ListOfUpperCoverOfMax = (Vector)
          setOfUpperCover.get(chains[indexBefore].getMax());
   Vector ListOfLowerCoverOfMin = (Vector)
          setOfLowerCover.get(chains[indexAfter].getMin());

   // finding the element in the "right chain" who covers the Top of the
  //  left chain
   if(ListOfUpperCoverOfMax.size()==1 &&
       chains[indexAfter].contains(ListOfUpperCoverOfMax.get(0)) ) {
       eCoveringMax = (Element) chains[indexAfter].get(
       chains[indexAfter].indexOf(ListOfUpperCoverOfMax.get(0)));
   }
   // finding the element in the "left chain" who is covered by the Bottom
   // of the right chain
   if(ListOfLowerCoverOfMin.size()==1 &&
        chains[indexBefore].contains(ListOfLowerCoverOfMin.get(0)) ) {
        eCoveredByMin = (Element) chains[indexBefore].get(
        chains[indexBefore].indexOf(ListOfLowerCoverOfMin.get(0)));
   }

   // best case: No LR links between both chains
   if(eCoveredByMin==null && eCoveringMax==null) {
       newBefore = (Chain) chains[indexAfter].clone();
       newAfter = (Chain) chains[indexBefore].clone();
   }
   // case : No "*-to-bottom" Links
   else if (eCoveredByMin==null){
       int indexOfECoveringMax = chains[indexAfter].indexOf(eCoveringMax);
       newBefore = new Chain();
       for(int i=0 ; i<indexOfECoveringMax ; i++) {
```

```java
        newBefore.add( chains[indexAfter].get(i));
      }
      newAfter = (Chain) chains[indexBefore].clone();
      for(int i=indexOfECoveringMax ; i<chains[indexAfter].size() ; i++){
         newAfter.add( chains[indexAfter].get(i));
      }
  // case : No "top-to-*" Links
  } else if (eCoveringMax==null){
      int indexOfECoveredbyMin =
          chains[indexBefore].indexOf(eCoveredByMin);
      newBefore = (Chain) chains[indexAfter].clone();
      for(int i=0 ; i<= indexOfECoveredbyMin ; i++) {
        newBefore.insertElementAt( chains[indexBefore].get(i), i);
      }
      newAfter = new Chain();
      for(int i=indexOfECoveredbyMin+1 ; i<chains[indexBefore].size() ;
          i++) {
         newAfter.add(chains[indexBefore].get(i));
      }
  // case : general case
  } else {
      newBefore = new Chain();
      newAfter = new Chain();
      int indexOfECoveredbyMin =
          chains[indexBefore].indexOf(eCoveredByMin);
      int indexOfECoveringMax = chains[indexAfter].indexOf(eCoveringMax);

      // newBefore
      for(int i=0 ; i<= indexOfECoveredbyMin ; i++) {
        newBefore.add((Element) chains[indexBefore].get(i));
      }
      for(int i=0 ; i< indexOfECoveringMax ; i++) {
        newBefore.add((Element) chains[indexAfter].get(i));
      }
      // newAfter
      for(int i=indexOfECoveredbyMin+1; i<chains[indexBefore].size();
          i++){
        newAfter.add((Element) chains[indexBefore].get(i));
      }
      for(int i=indexOfECoveringMax ; i<chains[indexAfter].size(); i++) {
        newAfter.add((Element) chains[indexAfter].get(i));
      }
  }
  // updating the rank and the name
  newBefore.setName(chains[indexBefore].getName());
  newBefore.setRank(chains[indexBefore].getRank());
  newAfter.setName(chains[indexAfter].getName());
  newAfter.setRank(chains[indexAfter].getRank());
  chains[indexBefore] = newBefore;
  chains[indexAfter] = newAfter;
}
```

Appendix 4:

The Arranging-Chain Algorithm in Java

```
/*
  * Arrange the aesthetic of the ordered set according to the chain
<chainIndex>.
  * call 2 functions:
  * arrangeLeftChains() : Arrange the chains located on the left.
  * arrangeRightChains(): Arrange the chains located on the right.
  *
  * @author Livaniaina Rakotomalala
  * @version 3.10
  * @param g a reference to the drawing space
  * @param chainIndex the index of chain whose neighbors will be arranged
  * @see class OrderedSet
  */
  public void arrangeChain(Graphics g, int chainIndex) {
    if(chainIndex !=  ) {
       arrangeLeftChains (g, chainIndex);
    }
    if(chainIndex !=width) {
      arrangeRightChains(g, chainIndex);
    }
  }


 /*
  * Arrange the chains on the left: The chains linked to the
  * least element of the chain <chainIndex> are closer.
  *
  * @author Livaniaina Rakotomalala
  * @version 3.10
  * @param g a reference to the drawing space
  * @param chainIndex the index of chain whose left neighbors will
  * be arranged
  * @see class OrderedSet
  */
  public void arrangeLeftChains(Graphics g, int chainIndex) {
    int end = chainIndex- ;
    int start;

    for(int i= ; i<chains[chainIndex].size() ; i++) {
       Element elem = (Element) chains[chainIndex].get(i);
       Vector itsListOfLowerCover = (Vector) setOfLowerCover.get(elem);
       for (int j= ; j<itsListOfLowerCover.size(); j++ ) {
          Element elemCovered = (Element) itsListOfLowerCover.get(j);
          int chainContainerIndex = getChainContainer(elemCovered);
          if(chainContainerIndex != chainIndex &&
             chainContainerIndex<chainIndex) {
             start = chainContainerIndex;
             for(int k=start; k<end; k++ ) {
                flipChains(g, k, k+1);
             }
             end--;
          }
       }
    }
```

```
}

/**
 * Arrange the chains on the right: The chains linked to the
 * greater elements of the chain <chainIndex> are closer.
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param g a reference to the drawing space
 * @param chainIndex the index of chain whose right neighbors will be
 * arranged
 * @see class OrderedSet
 */
public void arrangeRightChains(Graphics g, int chainIndex) {
  int end = chainIndex+1;
  int start;

  for(int i= chains[chainIndex].size()-1; i>=0 ; i--) {
     Element elem = (Element) chains[chainIndex].get(i);
     Vector itsListOfUpperCover = (Vector) setOfUpperCover.get(elem);
     for (int j=0 ; j<itsListOfUpperCover.size(); j++ ) {
        Element elemCovering = (Element) itsListOfUpperCover.get(j);
        int chainContainerIndex = getChainContainer(elemCovering);
        if(chainContainerIndex != chainIndex &&
           chainContainerIndex>chainIndex) {
           start = chainContainerIndex;
           for(int k=start; k>end; k-- ) {
              flipChains(g, k, k-1);
           }
           end++;
        }
     }
  }
}
```

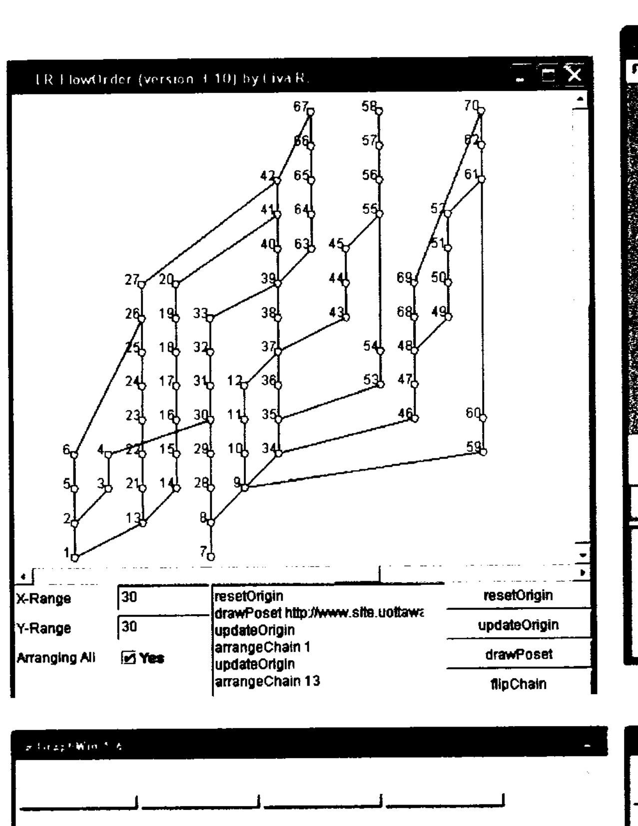
Appendix 5:

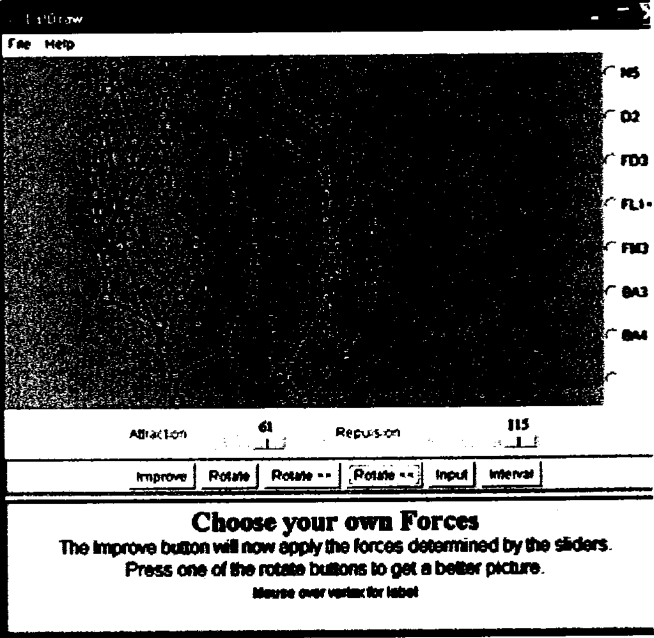
The Arranging-All-Chains Algorithm in Java

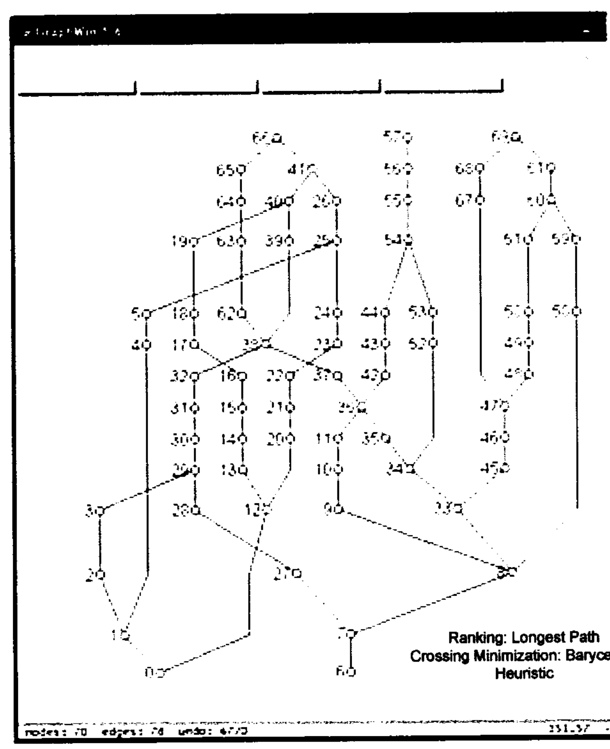
```
/*
 * Arrange All chains in order to produce an overall improvement view.
 * call 3 functions
 * - getPreferredChain() : select a chain where the improvement
 *                         is applied.
 * - arrangeAllLeftChains() : arrange all its left chains
 * - arrangeAllRightChains(): arrange all its right chains
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param g a reference to the drawing space
 * on its left will be arranged
 * @see class OrderedSet
 */
public void arrangeAllChains(Graphics g) {
   int indexHeaviestChain;

   indexHeaviestChain= getPreferredChain();
   arrangeAllLeftChains(g, indexHeaviestChain);
   arrangeAllRightChains(g, indexHeaviestChain);
}

/*
 * Arrange Left view from chains[chainIndex]..<--..chains[1]
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param g a reference to the drawing space
 * @param chainIndex the index of chain whose ALL chains located
 * on its left will be arranged
 * @see class OrderedSet
 */
public void arrangeAllLeftChains(Graphics g, int chainIndex) {
  for(int i=chainIndex; i>0; i--) {
    arrangeLeftChains(g, i);
  }
}

/*
 * Arrange right view from chains[chainIndex]..-->..chains[width-2]
 *
 * @author Livaniaina Rakotomalala
 * @version 3.10
 * @param g a reference to the drawing space
 * @param chainIndex the index of chain whose ALL chains located
 * on its right will be arranged.
 * @see class OrderedSet
 */
public void arrangeAllRightChains(Graphics g, int chainIndex) {
  for(int i=chainIndex; i<width-1; i++) {
    arrangeRightChains(g, i);
  }
}
```

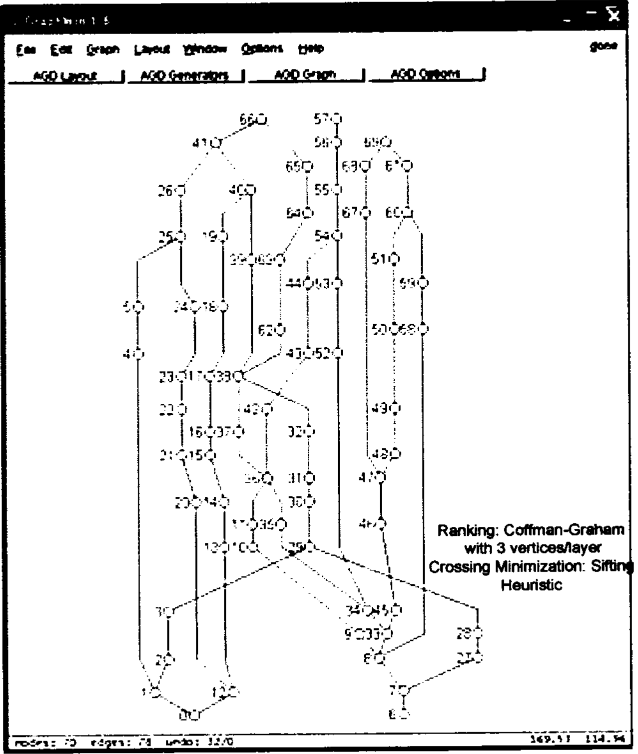
Appendix 6: LR-Drawing pictures versus Existing Systems.

LR-Drawing pictures versus Existing Systems.

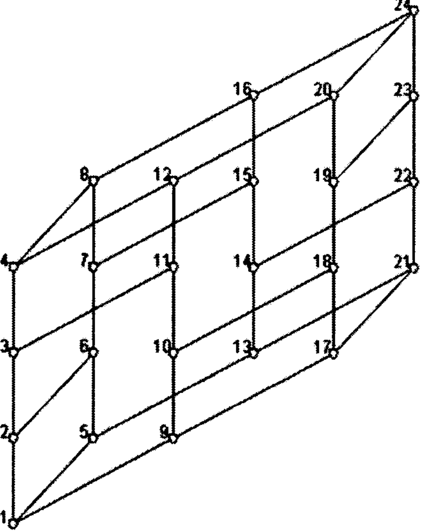




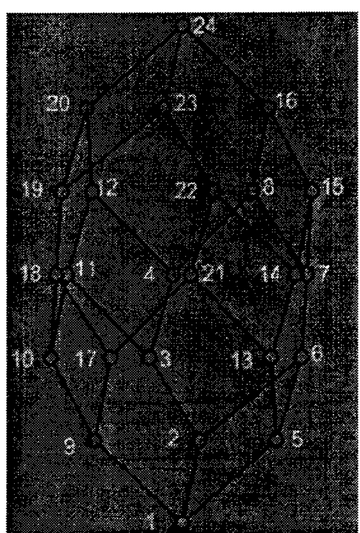




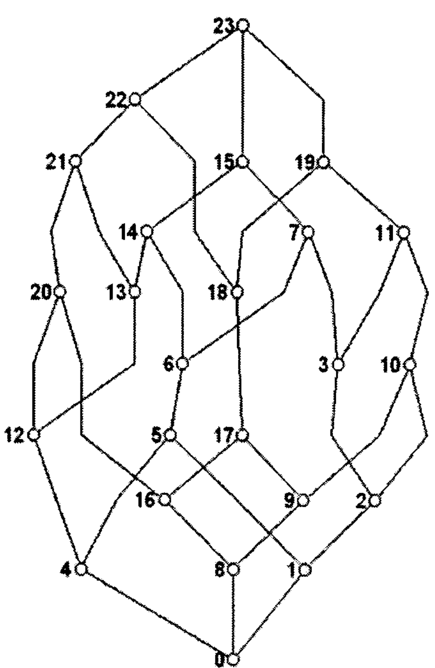
LR-Flow Order (LR Upward)



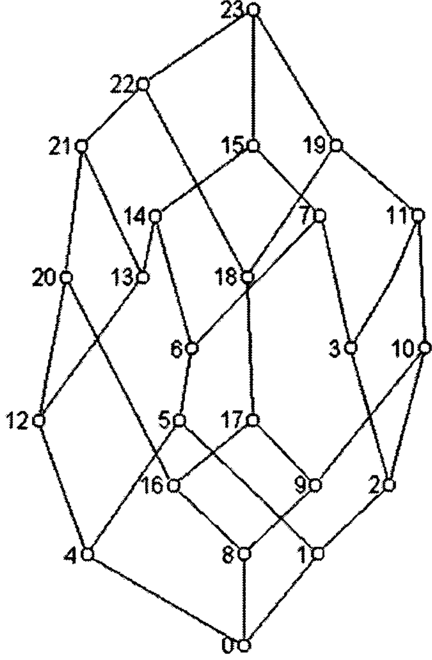
LatDraw (Forces Directed)



GraphWin (Hierarchical)

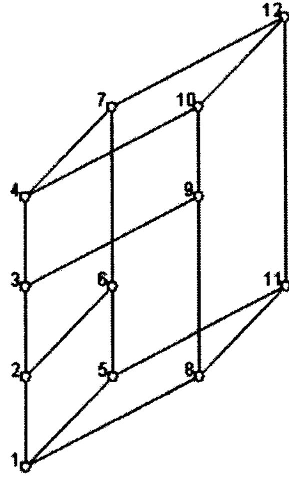


Ranking: Coffman-Graham with 3 vertices/layer
Crossing Minimization: Sifting Heuristic

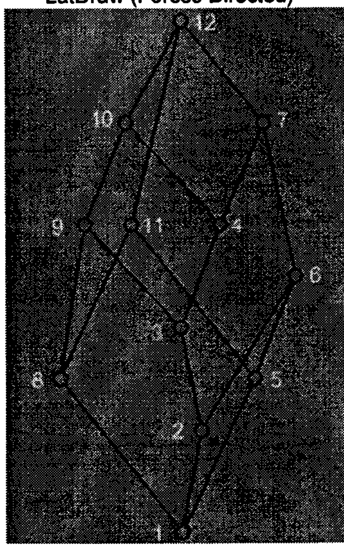


Bend on edges removed

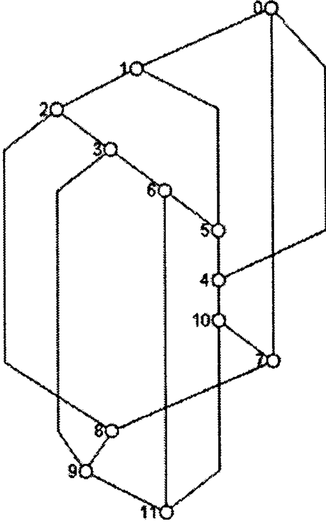
LR-Flow Order (LR Upward)



LatDraw (Forces Directed)

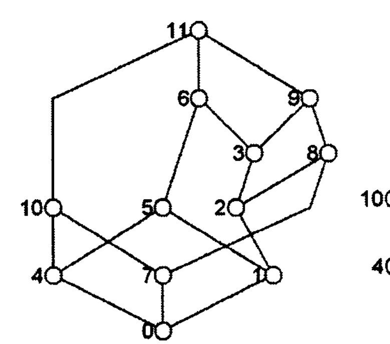


GraphWin (Hierarchical)

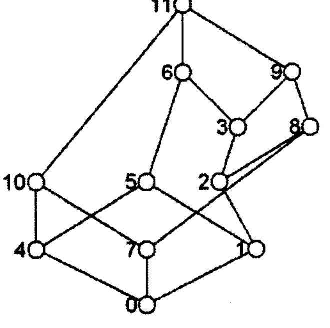


Ranking: DFS
Crossing Minimization: Sifting Heuristic
(Downward Picture)

GraphWin (Hierarchical)



Ranking: Coffman-Graham with 3vertices/layer
Crossing Minimization: Barycenter Heuristic



Bends removed