TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

Three-Dimensional Drawing of Lattice-Like Structures

conducted at the Institute of Information Systems
Knowledge-Based Systems Group
at the Vienna University of Technology

under supervision of
A.o.Univ.Prof. Dipl.-Ing. Dr.rer.nat. Uwe Egly

by

Bakk.techn. Georg Ziegler
Leidesdorfg. 19/7
A-1190 Wien

Date

Signature

Acknowledgements

I would like to thank Dr. Uwe Egly from the Vienna University of Technology for his support and inspiration during the course of this thesis. I am grateful to Dr. Helmut Leder from the University of Vienna for helpful suggestions on literature to use for the psychological section.
Furthermore, I want to thank Dipl.-Ing. Andreas Zugaj for providing source code and binaries of the `cgol` program. He also helped me with interesting insights and discussions.
I want to thank Tim Dwyer for making WilmaScope available as open source software, to be used by the public. It proved an ideal starting point to create an application for rendering three-dimensional drawings of lattices.
Last, I want to express my deepest gratitude to my parents for supporting me during my studies and my friend Lisa.

Abstract

Hasse Diagrams are a way to draw lattices in an easy-to-understand way. For complicated lattices with lots of edges, the conventional 2D approach usually is not adequate as the diagram is too cluttered. Too many edges and edge crossings make the picture unpleasing and hard to comprehend. In this thesis, we describe a way to model lattices in 3D and a Java program which implements this approach. Furthermore, this program acts as a wrapper for the `cgol` program proposed in the master thesis of A. Zugaj. The program `cgol` is a theorem prover for ortholattices. A proof search with `cgol` results either in a proof in a sequent-style calculus or an ortholattice which acts as a counter-example for the given formula to prove. This "counter lattice" is taken and transformed into a 3D Hasse Diagram. Such a Hasse Diagram depicts the transitive reduct of a lattice. As it cannot be guaranteed that all input represent a transitively reduced lattice, an approach which automatically calculates the transitive reduct of the input lattice is chosen.

This thesis describes some of the graph theoretic and algorithmic foundations of graph drawing. Besides the fundamental aspects, psychological and physiological implications of graph understanding are also taken into account. It describes a program which is devised according to the theoretic guidelines. This program acts as a wrapper for the program `cgol` to simplify user-interaction. Our implementation is then compared to two other prominent state-of-the-art lattice drawing programs.

Contents

1 Introduction and Motivation

"One picture is worth ten thousand words."

According to [25], this famous quotation was coined by F. R. Barnard—an advertising manager—in 1927. What was true then, namely that information is much better conveyed in a picture than in a long text, is still true today. It is not only applicable in the field of advertising, where a quick impression counts. There is psychological evidence that the same principle can be employed in many other areas as well. In [16] we can find a survey on the eye movement of people reading Shakespearian sonnets. It was found out that the reader spends 300 milliseconds for one fixation of the eyes and 35 milliseconds to advance to the next fixation. That yields about 150 words per minute if one fixation takes in about 1.2 words and if we take into account some degree of regressive movement while reading. As we will see later on we are not dealing with natural language texts here, but numbers and parentheses and we can safely assume the same or an even longer duration for deciphering one number. In [16] we can also find a reference to research on the eye movement of viewers when scanning pictures. Viewers have a general impression of the whole picture within the first seconds of exposure. During this relatively short time period, the eyes search for particularly interesting points in the picture where there is a "dark spot", a gradient fall indicating an edge or the like. Then the eyes move along edges to adjacent interesting spots. Mixed with this systematic scanning of the pictures is some chaotic eye movement which is similar to regressive movement when reading text and which cannot be controlled by the viewer.

This indicates that relational information can be much better conveyed if represented as a picture and not as a written text, as the picture can be decoded much quicker than the text. The time estimates in the above paragraph only deal with the proper reading, that is transferring the letters on the page to the brain in a certain way. For the information to become actually useful, a process of understanding the meaning is necessary. Furthermore,

if dealing with lattices, a representation of the lattice has to be constructed in the mind of the reader. This representation most likely will be that of a picture showing a Hasse Diagram of the lattice.
To ease the understanding of relational information occurring in lattice theory, H. Hasse used diagrams effectively. Later, this kind of diagram was named after him. At first, other theorists were skeptical of the diagrams and did not accept them for a proof. Later on, around the 1970s, they became an essential part of several proofs making them much more understandable.

When studying a logic, it is often desirable to have a calculus at hand, with which one can prove "valid" formulas of the semantics under consideration. With help of "theorem provers" (often called solvers), one can automate the search for a proof for a certain formula. If it is not possible to prove the formula, then there has to exist a counter-example which provides evidence that the formula is not true in every case. Most modern solvers only have two kinds of output after they have processed a formula, either "valid" or "invalid". Moreover, depending on the logic and the input formula, the solver might not terminate. In case the formula is valid, most solvers output a proof. In case the formula is not valid, then one is usually interested in a counter-example, indicating why the formula is not valid. Most solvers are not able to produce such a counter-example.
In the field of orthologic (sometimes called minimal quantum logic for historical reasons), a new solver—`cgol`—was devised by A. Zugaj as described in [32]. The solver is based on results from the investigation of proof search in [7] and the construction of counter-examples in [6]. This solver not only outputs whether a formula is valid or not, it can also construct a proof if the formula is valid and a counter-example if the formula is not valid. The counter-examples have the form of a semi-lattice. One can construct a lattice from this semi-lattice very easily which has the same properties as the semi-lattice and thus still serves as a counter-example for the formula. This lattice is represented as textual output and, as such, it is not very attractive.

Many people have tried to automatically draw lattices with a textual representation as input. These attempts either did not yield satisfactory results for several reasons as we will see later on, or it was very complicated to get the textual representation first and then use it to get the drawing. The program described in Section 5 aims to overcome the drawbacks of unpleasing drawings by exploiting the advances in 3D hard- and software to create a three-dimensional model of the lattice, but also trying to find a better layout algorithm suitable for three dimensions. To ease the handling of formulas, the program interfaces directly with `cgol` and thus hides many of the textual files which the user had to work with. To use the functionality of `cgol` in the drawing program, the solver was adapted to work as a dynamic library. This integration of `cgol` into the drawing program allows the user to interactively manipulate the lattice. It also allows user-guided optimization of the lattice by interactively adding non-logical axioms. This optimization can dramatically reduce the size of these lattices.This library is only needed when dealing with orthologic formulas. It is not needed for simple rendering of input lattices. The drawing program does not only layout the lattice it gets as input, but it also performs some computations and optimizations that make it easier for the user to comprehend the structure of the lattice. One example of such an optimization is the transitive reduction of the lattice. This can reduce the number of edges in the diagram and therefore greatly improve the readability of the picture.

The thesis is organized as follows. In Section 2, the theoretical foundation for later chapters is laid. Important terms like that of posets, lattices and chains are defined. A short introduction to graph theory and Hasse Diagrams as far as they are needed for this thesis can also be found here. At the end of this section, we discuss efficient algorithms for lattices.
Section 3 deals with the physiological and psychological aspects related to three-dimensional perception and aesthetics for graph layouts. Different aspects are discussed which lead to three major layout algorithms implemented

in WilmaScope. Each of these algorithms satisfies the stated requirements to some degree. A detailled description of the layout algorithms follows in Section 4.

Section 5 deals with the implementation details of the program which was developed in the course of this thesis according to the insights from the previous sections. First, the use of the program is explained and then illustrated with screenshots in order to show the functionality of the program. The rest of this section is devoted to the technical details of the drawing program and the development process. As the developed prototype proved not to be sufficient, an alternative approach was explored. This alternative in the form of the WilmaScope program proved to be a good choice for the reasons given in Subsection 5.5 where a description of this program can also be found.

Section 6 explores the analogies and differences of the chosen approach to the two most important drawing programs in this context, which are the LatDraw program from Ralph Freese and the graphviz package developed by Emden R. Gansner and others.

2 Theoretical Background

In this section, we introduce the theoretical background from lattice theory, graph theory and algorithms for drawing graphs. In the later sections, it will be necessary to use these definitions and theorems to justify the approach that was chosen, as well as to support the understanding of the algorithms.

2.1 Lattice Theory

A *partially ordered set* (or poset) is an ordered pair $(P, \leq)$ where P is a set and $\leq$ is a partial order relation on P. This partial order relation is a subset of $P \times P$ and is written as $a \leq b$, where $a, b \in P$. It satisfies the following properties

P1	$a \leq a$	reflexivity,
P2	$a \leq b$ and $b \leq a$ implies $a = b$	antisymmetry and
P3	$a \leq b$ and $b \leq c$ implies $a \leq c$	transitivity.

The equality used in P2 is defined as follows:

$$a = b \iff a \leq b \wedge b \leq a.$$

This partial order relation is the reflexive and transitive closure of the cover relation $\prec$. This cover relation satisfies P2 and, for all $a, c \in P$, P4 given below.

P4 $\quad a \prec c \iff$ for any $b \in P$, $a \leq b \leq c$ implies $a = b$ or $b = c$.

Another order relation $<$ can be defined on P. This relation does not satisfy P1 and P4, but P2 and P3.
An *upper bound* of a subset X of a poset P is an element c of P such that $x \leq c$ for all elements x of X. Similarly, a *lower bound* d of X is an element d of P for which it holds that $d \leq x$ for all elements x of X. The supremum of X, $\mathit{sup}(X)$, is an upper bound of X where any other upper bound u of X is greater than or equal to $\mathit{sup}(X)$. The infimum of X, $\mathit{inf}(X)$, is a lower bound of X such that any other lower bound of X is less than or equal to $\mathit{inf}(X)$.

The supremum or infimum of arbitrary sets need not to exist. *Lattices* are those posets, for which the supremum and the infimum exist for any two-element subset. *Complete lattices* are lattices for which, the supremum and infimum of arbitrary subsets exist.

A *bounded poset* is a poset which contains a smallest element (zero element) denoted by $\mathbf{0}$ and a largest (or unit) element denoted by $\mathbf{1}$. Elements $a, b \in P$ are called *comparable*, if $a \leq b$ or $b \leq a$ holds. They are *incomparable*, if they are not comparable. A *chain* is a subset of a lattice, where any two elements are comparable. In other words, a chain is a totally ordered subset of a lattice. Since a chain satisfies all the requirements for a lattice, it is also a lattice. An *anti-chain* is a subset of a poset, where any two elements are incomparable. A poset is called *chain finite*, if all chains in the poset are finite. The *length* of a chain is defined as its cardinality. The length of a lattice is equal to the length of the longest chain in it. A chain C which satisfies the following three conditions is called a *chain from a to b*.

C1 The element a is the lower bound of C.

C2 The element b is the upper bound of C.

C3 The length of C is less than or equal to the length of all chains satisfying C1 and C2.

In the following, the operations $\wedge$ and $\vee$ denote the meet and join, respectively. A lattice L ist called *upper semimodular* if $a \wedge b \prec a$ implies $b \prec a \vee b$ for all elements a, b in L. L is called *lower semimodular* if $b \prec a \vee b$ implies $a \wedge b \prec a$. If L is chain finite and is upper and lower semimodular, it is called *modular*.

A lattice L satisfying the *distributive law* given in Equation (1) for all elements $x, y, z \in L$ is called a *distributive lattice.*

$$\begin{aligned} x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) \\ x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) \end{aligned} \tag{1}$$

A *rank* can be assigned to each element x in a lattice. This rank of x, $rank(x)$, is defined as the length of the chain from $\mathbf{0}$ to x.

An *interval between a and b*, [a, b], is a subset I of a lattice which satisfies the following conditions.

I1 The element a is the least element of I.

I2 The element b is the greatest element of I.

I3 For all other elements x in I, it holds that $a \leq x$ and $x \leq b$.

Another way to define I is to view it as the union of all chains from a to b.

An *orthoposet* is a bounded poset with a unary *orthocomplementation* operation ' satisfying the following three conditions for all $x, y \in P$.

O1 If $x \leq y$ holds, then $y' \leq x'$ holds.

O2 The equality $x'' = x$ holds.

O3 The supremum $x \vee x'$ and the infimum $x \wedge x'$ exist and the equations $x \vee x' = \mathbf{1}$ and $x \wedge x' = \mathbf{0}$ hold.

An *ortholattice* is an orthoposet which is also a lattice. Particularly interesting are ortholattices where the distributive law as given in Equation (1) holds. One example of such a distributive ortholattice is a Boolean Algebra. Note that classical propositional logic is defined as the logic of distributive ortholattices.

Here, the definition of an (ortho-)lattice as a special poset is particularly interesting, as the drawing has to reflect that ordering. There is another possibility to define a lattice by equations.
An *equational class* consists of all algebras satisfying a given set of equations. In the case of lattices, the equational class $\mathcal{L}$ of lattices consists of algebras $(L; \wedge, \vee)$ where L is a set and $\wedge$ and $\vee$ are the binary operations *meet* and *join*. The set of equations these algebras have to satisfy includes the following associative, commutative and absorption laws.

$$\begin{array}{ccl} a \vee (b \vee c) = (a \vee b) \vee c & a \wedge (b \wedge c) = (a \wedge b) \wedge c & \text{associative law} \\ a \vee b = b \vee a & a \wedge b = b \wedge a & \text{commutative law} \\ a \vee (a \wedge b) = a & a \wedge (a \vee b) = a & \text{absorption law} \end{array}$$

From these equations, another important law can be derived for $\mathcal{L}$. From the absorption law, the *idempotent laws* $a = a \wedge (a \vee (a \wedge b)) = a \wedge a$ and $a = a \vee (a \wedge (a \vee b)) = a \vee a$ follow.

A *linear extension* of a poset P is a total order on P containing $\leq$. This linear extension is also referred to as topological sorting. The proof that every poset also has a linear extension can be found in [9].
A *principal filter* or *filter* for short, of an element x in a lattice is the union of all chains from that element x to the top element. Recall that chains are just special sets. This union in general is not a chain. A *reduced principal filter of an element* x is just the principal filter of x not containing x itself.

To illustrate the above definitions, a few examples are given. These examples use Figure 1 as a reference. For brevity, a letter is assigned to each vertex which can be found to the left of each vertex.
In the given structure $\mathbf{a} \leq \mathbf{f}$ and $\mathbf{a} \prec \mathbf{b}$ hold, but $\mathbf{a} \prec \mathbf{f}$ does not hold. The reason for this is that **b** and **d** directly cover **a**, but either element **b** or **d** lies between **a** and **f**. All those elements connected with a single line or a path of lines are comparable. Elements **b** and **c** are an example of two incomparable elements. An exemplary chain is the set {**a**, **b**, **f**, **h**}. This chain has length 4. The sets {**b**, **g**} and {**c**, **f**} are examples for anti-chains. The structure is a bounded poset, since it contains vertices **a** and **h** as its respective smallest and largest elements. A linear extension of this structure might look like {**a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**}. The principal filter of **c** is the set {**c**, **e**, **g**, **h**}, the reduced filter just {**e**, **g**, **h**}.
Figure 1 can also be used as an example of an ortholattice if the orthocomplementation operation is interpreted as the set-theoretic complement. Indeed, all the requirements for the orthocomplementation operation O1 to O3 also

hold for the set-theoretic complement as discussed below.
Let us consider the elements **b** and **f**. For these two, the relation $\mathbf{b} \leq \mathbf{f}$ holds in this structure. To satisfy O1, **f**' $\leq$ **b**' should hold. The set-theoretic complement of set **f** is the set {2} and the complement of **b** is {2, 3}. These sets are labelled **c** and **g**, respectively, and indeed $\mathbf{c} \leq \mathbf{g}$ holds in this structure. To show that O2 holds, let us consider element **b**. Above we have established that **b**' is **g**. Now, what we need is **g**' to obtain **b**". The complement of set **g** is the set {1} which is **b**, thus **b** = **b**" holds and O2 is satisfied. To see that O3 holds, again consider **b** and **b**' = **g**. Remember that **b** $\vee$ **b**' can also be written as $\{1\} \cup \{2, 3\}$ which is equal to {1, 2, 3} = **h**. Now, **h** is the largest element in the structure, denoted by **1** in O3. The other equation in O3, $x \wedge x' = \mathbf{0}$, can be proved similarly. $\{1\} \cap \{2, 3\} = \{\}$ which is **a**, the smallest element in the structure. This smallest element is denoted by **0** in O3. To verify that O1 to O3 hold for all other elements, a similar argumentation can be used. This is left to the reader.

2.1.1 The Chain Condition of Jordan and Dedekind

In [15], the procedure Jodeh is devised which computes the transitive reduct of an acyclic graph in linear time. A prerequisite for this procedure is that the graph has to satisfy the Jordan-Dedekind Chain Condition (JD for short), which is given below. The definition is taken from [28].

> For all elements a, b [of a lattice] with $a < b$, all maximal chains of the interval $[a, b]$ have the same length, that is, either all chains between a and b are inifinte or the lengths of all chains equal the same finite number.

It is essential to note that only finite lattices can be processed with the program proposed in Section 5. It is also important to note that finite lattices always satisfy the Jordan-Dedekind chain condition. The proof can be sketched as follows. By Theorem 1.9.9 in [28], every lattice of finite length is upper semimodular. Together with Theorem 1.9.1, which states that any

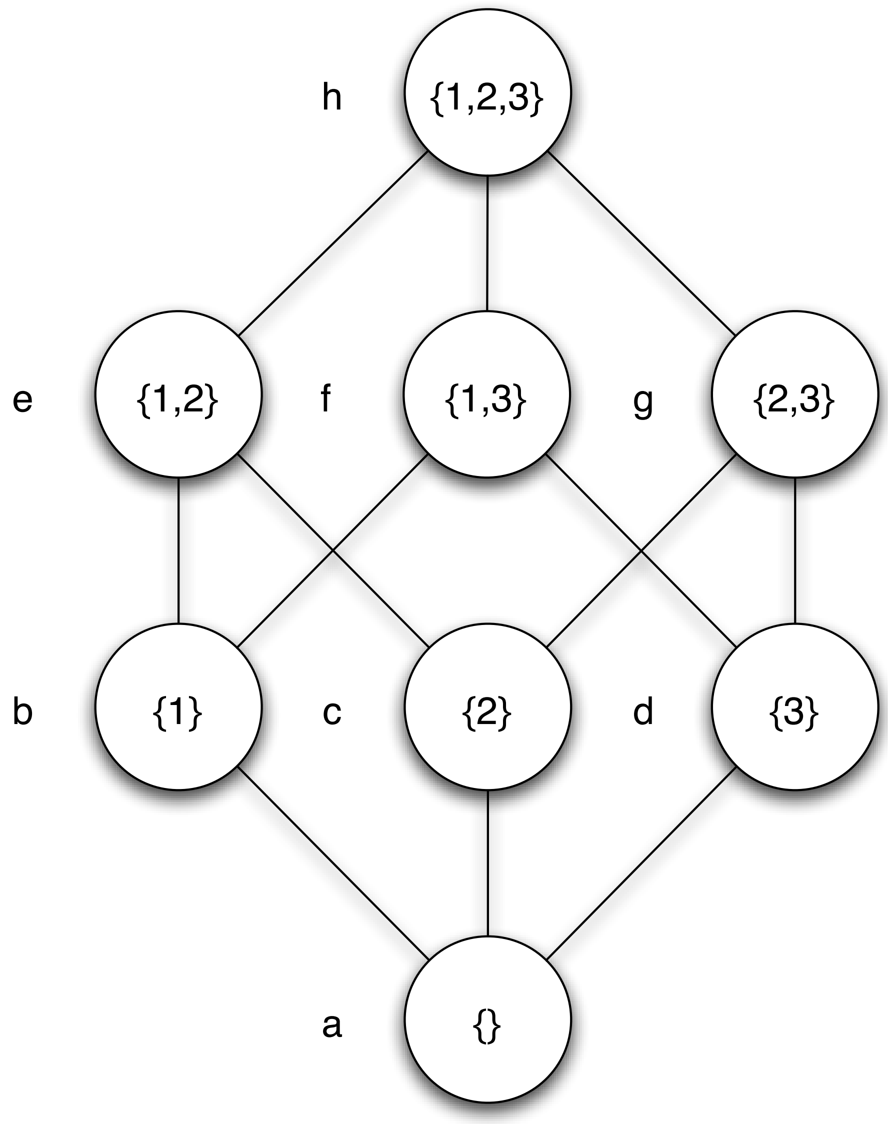


Figure 1: The Lattice of Subsets of a Set with Three Elements.

upper or lower semimodular lattice satisfies JD, we can conclude that every finite lattice satisfies JD and the procedure Jodeh can be used.
This procedure gives an upper bound on the time complexity for computing the transitive reduct. The algorithm proposed in Subsection 2.4.2 is an adapted version of the Jodeh procedure. For performance reasons, it does not operate on sets, as Jodeh does, but more suitable data structures are used. In Subsection 2.4.2, it will be shown that Jodeh and the adapted algorithm both have the same upper bound on time complexity.

2.2 Graph Theory

A *graph* G is an ordered pair $\langle V, E \rangle$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. If G is an *undirected graph*, then, for each element $e \in E$, it holds that, if $e = (v_1, v_2)$, then E also has to contain $e' = (v_2, v_1)$. A *directed graph* need not to have both e and e' in its set of edges. Two vertices $v_1, v_2 \in V$ are called *adjacent* iff (v_1, v_2) is in E. An edge e and a vertex v are *incident* iff $e = (v, v_1)$ or $e = (v_1, v)$. A *representation* of a graph is a drawing, where each vertex is depicted as a dot, circle or sphere and each edge is depicted as a line or cylinder connecting the two vertices it is incident to. A *planar* graph is a graph which has a representation where there are no intersections between edges. A path P between u and w of length k is a $(k+1)$-element subset $\{v_0, v_1, ..., v_k\}$ of V satisfying the following three conditions

G1 $u = v_0$,

G2 $w = v_k$, and

G3 for all i with $0 \leq i < k$, there is an edge (v_i, v_{i+1}) in E.

The graph theoretical *distance* between vertices u and w is the length of the shortest path between u and w. A *cycle of length* k is a path where $v_0 = v_k$. An *acyclic graph* is a graph not containing any cycles. The above definitions of path, cycle and acyclic graph in this form also hold for directed graphs.

The transitive closure G^* of a graph $G = \langle V, E \rangle$ is an ordered pair $\langle V, E^* \rangle$, where E^* is a set of edges for which the following two conditions hold.

T1 $E \subseteq E^*$

T2 If (v_1, v_2) and (v_2, v_3) are in E^*, then also (v_1, v_3) is in E^*.

The transitive reduct G^- of a graph $G = \langle V, E \rangle$ is an ordered pair $\langle V, E^- \rangle$ where E^- is the smallest set of edges such that the transitive closure of G^- and the transitive closure of G are the same.

A *weighted graph* G is an ordered 4-tuple $\langle V, E, \Gamma, \Delta \rangle$. V and E are defined as above, Γ and Δ are functions. Γ is defined as $\Gamma : V \to \mathbb{R}$ which assigns a weight to each vertex. The function Δ is defined as $\Delta : E \to \mathbb{R}$ and assigns a weight to each edge.

2.3 Hasse Diagrams

The Hasse Diagram is named after H. Hasse. He was the first who extensively used a certain diagram to display lattices. Ortholattices are an extension of the concept of lattices. This kind of diagram can also be used to depict ortholattices. Hasse Diagrams are directed acyclic graphs. Usually, the arrows which would indicate the direction are omitted, since the orientation of any edge is known to be from bottom to top as indicated in the conditions H4 and H5 below.
This kind of diagram is one way to graphically represent a poset. It conforms to the criteria H1,...,H5 given below.

H1 The diagram is organized in layers.

H2 Each element is represented by a circle.

H3 The bottom-most elements $b_1, b_2, ...$ are drawn on the lowest layer.

H4 If an element x covers an element y ($y \prec x$), then x is drawn in the layer directly above the layer of y.

H5 If an element x covers an element y, then the circles for x and y are connected with a (straight) line from x to y.

An example of such a Hasse Diagram is given in Figure 1. The relation depicted is set inclusion. Though this definition was originally intended for two-dimensional drawings, it can be adopted for three-dimensional drawings.

2.4 Algorithms on Lattices

In this subsection, we introduce some of the most important algorithms on lattices which are important later on. All these algorithms are used in the program and their worst case time-complexity will be discussed by analyzing pseudo code descriptions. When analyzing the time complexity of the algorithms the following notation will be used. The number of vertices in a graph $|V|$ will be given as n, the number of edges $|E|$ as m and the length of a lattice as l. For simplicity, we will only refer to lattices in the following although all of the facts mentioned in this subsection also hold for ortholattices unless otherwise mentioned.

2.4.1 Computation of Levels, Linear Extension and Principal Filters

As it was pointed out in Subsection 2.1, a rank can be assigned to each element v in a lattice. This rank of v will also be referred to as the level of v. To compute the levels of all elements in the lattice by traversing the graph just once and to get the linear extension of the elements at the same time, the Breadth First Search algorithm (BFS for short) is used. BFS is designed to traverse a (directed) graph from a starting vertex and then visiting all adjacent vertices before choosing a new starting vertex among the vertices visited in the previous step. Compared to the Depth First Search algorithm (DFS for short), it has the drawback of much more bookkeeping being involved in the process. Its big advantage is that all the vertices get the right level by traversing the graph exactly once. DFS would have to traverse certain

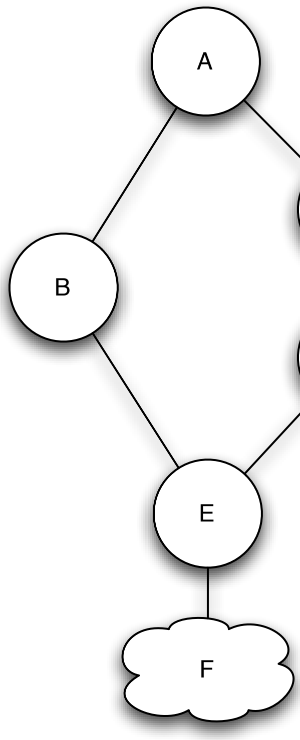


Figure 2: Example Graph for Comparing BFS, DFS and IDDFS.

subgraphs several times. Take the graph in Figure 2 as an example. The subgraph F rooted at vertex E and depicted as a cloud shape would have to be visited two times by DFS to guarantee that all vertices in F have the right rank. Let us assume that DFS always expands nodes from left to right and that a rank of zero is assigned to A. Then the rank of E changes from 2 to 3 when DFS visits it the second time. Accordingly, all the ranks in F need to be updated and the whole subgraph F has to be explored a second time. With BFS each edge is traversed only once.

A disadvantage of BFS is often the enormous space requirement because nearly the whole search space as to be recorded in the memory. For our application here, we always have the underlying graph in the main memory. Therefore, BFS can be preferred over DFS without disadvantages. Thus we

can guarantee a running time of $O(m)$.
If DFS is used, then a linear extension can also be computed in $O(m)$ by using a *visited* flag for each vertex. This flag is set in a vertex after all outgoing edges have been visited. If it is set, then DFS will not explore the subgraph rooted in this vertex again. The problem with this approach is the computation of levels which would require the whole subgraph rooted in a vertex v to be explored again every time the level of v changes. This would unnecessarily increase the running time.
Related to BFS and DFS is the approach of Iterative Deepening Depth First Search (IDDFS for short). The idea is to perform a DFS, but to bound the depth of maximal descent. If there are vertices which were not yet visited, increase the bound and perform another DFS. It can be used to traverse directed graphs and assign ranks to vertices, yet in this context it is not a sensible choice. If all vertices are visited up to a predetermined rank and then the whole procedure is repeated with an increased rank, the example graph in Figure 2 would be visited in the following manner. Let us assume that we want to determine the exact rank of F and we only count the vertices we visit on the way down and again that DFS expands vertices from left to right. Then DFS visits 8 vertices in the sequence A, B, E, F, C, D, E, F. Similarly, BFS also visits 8 vertices but in the sequence A, B, C, E, D, F, E, F. In contrast to the first two algorithms IDDFS performs far worse. The reason is that the vertices between A and F are visited more than once, because the rank up to which IDDFS descends is increased in each iteration. The total number of vertices that IDDFS visits is composed as follows. In the first iteration, IDDFS visits A, B, C, in the second A, B, E, C, D, in the third A, B, E, F, C, D, E and finally in the fourth A, B, E, F, C, D, E, F. This equals a total of 24 vertices. Since the whole graph has to be explored DFS has to be performed a number of times equal to the length of the longest path in the graph. This approach is of better use if only a small portion of a graph needs to explored. Thus BFS remains the best choice.
The pseudo code of the method used to compute the linear extension and

```
Vertex {
    int index = -1;
    int level = -1;
    int position = -1;
    Vertex[] cover = null;
    BitSet filter = null;
}
```

Figure 3: The `Vertex` Data Structure with Initial Values.

levels and principal filters of vertices in the program is given below in Algorithm 1.

Before discussing the details of the algorithm, a look at the data structure employed in the algorithms is helpful. The vertex data structure is shown in Figure 3. Each vertex holds the following data. It has an index to identify it and an associated level. This level is used to determine at which level of the Hasse Diagram this vertex is located. The position of a vertex v, later referred to as $position(v)$, is the integer index i, for which it holds that $extension[i] = v$. The principal filter of a vertex v, referred to as $filter(v)$, stores the characteristic function of v's principal filter. It is an array of bits with length equal to the total number of vertices. If a vertex w is contained in the principal filter of v, then the bit at the position $position(w)$ is set to one, otherwise it is zero. The values for index and cover are set during the processing of the input file. The values for level, position and filter are computed in Algorithm 1, which is described in the following.

The function `ComputeLevels` computes the level of each vertex using the auxiliary method `level_BFS`. We estimate the running time of `ComputeLevels`, which is essentially the runtime of `level_BFS(Set` S`, int` $level$`)`. The `foreach`-loop in lines 10 to 15 fills the set S^+ for the next recursive call of `level_BFS(Set` S^+`, int` $++level$`)`. If S^+ is empty, we have reached the top element and no further recursive calls are needed, otherwise we enter a new recursion. The

maximal recursion depth of all calls is equal to the length of the lattice. Each vertex is put into S^+ a number of times equal to the number of its incoming edges. If we have reached the top element, we put it into the linear extension, compute its principal filter and leave the recursion. Before leaving a recursion, all vertices in S, which are not put into the extension, are put into the extension and their filters are computed. Note that the i used in the `foreach`-loop in lines 19 to 25 is a global variable. This is the reason it is declared in line 1 outside the local scope of all functions.
The time complexity of the whole algorithm T is given in Equation (2) where t_i is the time taken for one recursion with input set S_i as given in Equation (3). E_i is the set of edges with starting vertices in S_i. The cardinality of the set S_i, $|S_i|$ will be given as n_i and the cardinality of the set E_i as m_i. The set of all vertices in S_i which are not contained in the linear extension is denoted by $\bar{n}_i$

$$T = \sum_{i=0}^{l} t_i \tag{2}$$

$$t_i = \underbrace{O(n_i + m_i)}_{\text{for}_1} + O(1) + \underbrace{O(\bar{n}_i + m_i)}_{\text{for}_2} \tag{3}$$

The first term labelled for_1 is derived from the `foreach`-loop on lines 10 to 15. The middle term represents the `if`-statement and the recursive call. The last term labelled for_2 is derived from the `foreach`-loop on lines 19 to 25. We will now estimate T with a worst-case analysis. We are dealing with directed acyclic graphs. This class of graphs can have at most $\sum_{i=0}^{n-1} i$ edges in a graph with n vertices, otherwise it contains a cycle. Therefore, the S_i have sizes $1, n-1, n-2, ..., 1$ since each vertex is put into S_i a number of times equal to the number of its incoming edges. In the worst case we start with S_0 containing one vertex. This vertex v_0 has an edge to all other vertices in the graph, which are put into S_1. In S_1, one vertex v_1 has to have an edge to all other vertices in the graph except for v_0. This observation can be generalized for all S_i with $i \geq 1$ such that $S_i = S_{i-1} \setminus \{v_{i-1}\}$.
Summing up only the first terms of Equation (3) yields $O(1 + \sum_{i=0}^{n-1} i + m)$

since each edge is explored exactly once. This can be rewritten as $O(1 + \frac{(n-1)n}{2} + m)$.
Summing up the middle terms yields $O(l)$ since it is done once in each recursive call. In the worst case, the whole graph is a directed chain and $l = n$, thus we can rewrite this estimate to $O(n)$.
The third term is different form the first in that the loop only considers vertices which are not yet in the extension. Thus, each vertex is handled only once and again, each edge is explored only once. This yields a total time of $O(n + m)$. This third term equals the time estimate of BFS as usually given in the literature.
Taking all three of the sums above together we obtain the time estimate for T as given in Equation (4). Taking into account that constants can be omitted in the O-notation and that m can also be estimated with $O(m) = O(n^2)$ we reach the result in Equation (5).

$$T = O(1 + \frac{(n-1)n}{2} + m) + O(n) + O(n + m) \tag{4}$$

$$T = O(n^2) \tag{5}$$

The increased cost compared to traditional BFS is caused by the computation of the levels. Traditionally, BFS is realized with some queue data structure to store the working set and each vertex is put into the queue only once. The problem with this approach is, that it does not compute the right levels. It computes the shortest path from the root vertex to each other vertex in the graph. In contrast, we are interested in the longest path from the root to each vertex. That is why we have to consider certain vertices several times, which increases the running time.

2.4.2 Computing the Transitive Reduct

The Hasse Diagram is a representation of the transitively reduced graph of a poset. Therefore, it is necessary to compute the transitive reduct of the input poset since it cannot be guaranteed that the input is of that form. Furthermore, this reduction can increase the readability of the graphs.

Input: A set S of starting vertices with no incoming edges, the total number of elements and a set E of directed edges not containing cycles.

Result: An array with a linear extension of the vertices with their levels and filters.

```
Vertex [] $extension = null$; int $i = -1$;
ComputeLevels(Set $S$, int $numElements$) {
    extension = new Vertex [numElements];
    $i = numElements - 1$;
    level_BFS($S$, $0$);
    **return** extension;
}
level_BFS(Set $S$, int $level$) {
    Set $S^+ = \{\}$ ;
    **foreach** $vertex\ v\ in\ S$ **do**
        $level(v) = level$; $position(v) = -1$;
        **foreach** $(v, w) \in E$ **do**
            $S^+ = S^+ \cup \{w\}$
        **end**
    **end**
    **if** $S^+ \mathrel{!}= \{\}$ **then**
        level_BFS($S^+$, $++level$);
    **end**
    **foreach** $vertex\ v\ in\ S\ where\ position(v) == -1$ **do**
        $extension[i] = v$; $position(v) = i$;
        $filter(v) =$ new BitSet (); $filter(v, i) = 1$; $i = i - 1$;
        **foreach** $(v, w) \in E$ **do**
            $filter(v) = filter(v) \vee filter(w)$;
        **end**
    **end**
}
```

Algorithm 1: Computation of Linear Extension and Levels Using BFS.

Each vertex v has references to two sets of vertices, namely the vertices which cover v and the vertices which are in the principal filter of v. The set of vertices denoted by `cover` in the vertex data structure in Figure 3 is initialized when the graph is read in. It cannot be guaranteed that the graph is transitively reduced, for which reason this array may contain vertices which do not directly cover v, but are in the principal filter of v. This array has to be reduced to contain only the vertices directly covering v.The set denoted by `filter` in the vertex data structure stores a set of vertices which are in the principal filter of v in the form of a `BitSet` representing the characteristic function of this set. With this information, the transitive reduct can be computed as follows. The pseudo code of the method used is given in Algorithm 2.

For each vertex v, the set of vertices in the filter of v but not directly covering v are computed in the following way. All the reduced principal filters of the vertices w—the filters of the vertices w not containing w itself—in the cover of v are joined. This operation is performed as a bitwise or of the `BitSets` storing the principal filter of each vertex w. This yields the filter of v where the vertices w are only contained, if they are not covering v. In other words, another vertex lies between v and w. If the bit at position w is set to 1 in this `BitSet`, then w it is not directly reachable from v, but from some other vertex. Therefore, w cannot directly cover v.

As can be seen this method explores each edge of the graph twice. Once in each `foreach`-loop starting at lines 5 and 10. At line 12, all the vertices in the cover of v which are reachable via an intermediate vertex are removed from the cover of v. As indicated above, the bit at position v_1 This process explores each edge a third time due to the kind of data structure used to represent the edges in a graph. The array storing references to the vertices covering v needs to be updated. This yields a total time of $O(m)$ for computing the transitive reduct of the input lattice. This is exactly the runtime estimate given by the Jodeh procedure in Subsection 2.1.1. The use of a `BitSet` to store the principal filters allows us to perform set union in constant time.

Since Jodeh is described using proper sets, an efficient encoding of the sets is needed. Therefore, `BitSets` are used to encode the characteristic function of these sets.

```
Input: A graph as a set $S$ of vertices with principal filters.
Result: Transitively reduced graph as a set of vertices.
ComputeTransitiveReduct(Set S) {
    foreach vertex v in S do
        /* new BitSets are initialized with zeros          */
        BitSet bs = new BitSet ();
        BitSet temp = new BitSet ();
        foreach vertex w adjacent to v do
            $temp = filter(w)$;
            $temp(position(w)) = 0$;
            $bs = bs \mid temp$; /* bitwise or of bs and temp  */
        end
        foreach vertex w adjacent to v do
            if bs(position(w)) == 1 then
                /* w reachable via an intermediate vertex   */
                $cover(v) = cover(v) - \{w\}$;
            end
        end
    end
}
```

Algorithm 2: The Transitive Reduct Method.

2.5 Description of Cgol

In this thesis, we describe and implement a system for lattice drawing with a theorem prover for ortholattices as a subsystem. This prover is called `cgol` and is described in full detail in [32]. In this subsection we give an introductory explanation of its basic ideas and goals.

The prover `cgol` is based on the paper [7] which presents an analysis of the search space for different search strategies of solvers for lattices. It is also based on [6], which deals with the construction of lattices from "saturated" sets of sequents.
In [7], different proof strategies for orthologic are considered. Orthologic is a logic defined over lattices where the orthocomplementation operation is defined. It is similar to classical propositional logic, with the difference that the distributive law does not hold in general for ortholattices. Also related to orthologic is orthomodular logic, a logic over ortholattices where the law of modularity holds. Now, `cgol` is an automated theorem prover for orthologic which performs proofs in a sequent style calculus. A sequent is an ordered pair $M \vdash N$ where M and N can be sequences, sets or multisets of formulas depending on the calculus. In the case of `cgol`, M and N are sets. An n-restricted sequent is a sequent $M \vdash N$, where M and N are sets and $|M| + |N| \leq n$ holds. Here, $|X|$ denotes the cardinality of the set X. A sequent style calculus or Gentzen system, as it is often called, uses axioms and inference rules to derive a proof for a given formula. Two sets of axioms can be distinguished which are given below.

- Logical Axioms are sequents which are known to be valid. Most important are sequents of the form $a \vdash a$, where a is an atom.

- Non-logical Axioms are sequents which are assumed to hold. Such non-logical axioms are not needed in the usual proof search, but can be used to find counter-examples of smaller sizes.

$$\frac{P}{C}\,\alpha$$

(a) Unary Inference Rule.

$$\frac{P \qquad Q}{C}\,\beta$$

(b) Binary Inference Rule.

Figure 4: Gentzen Style Inference Rules.

The inference rules in a Gentzen system can be of two forms as well. They are shown in Figure 4, where P, Q and C are sequents. P and Q are called premises and C is called the conclusion of the inference rule. There are rules for each logical connective and some structural rules. For details on these rules, please see [7] or [32].
In Gentzen systems one can distinguish three kinds of search strategies given below.

- First, there is backward search. It starts with the end sequent S and proceeds towards the axioms during the search. This strategy produces a tree proof for S with S as the root of the tree. All the inner nodes must be sequents which were derived by the application of inference rules on their predecessors. All leaf nodes must be (non-logical) axioms.

- Then there is forward search which is also known as Maslov's inverse method. It usually produces a sequence proof. A sequence proof of a formula S from a set of (non-logical) axioms A has the following form. It is a sequence of sequents $S_1, ..., S_n$ where $S = S_n$ and, for $1 \leq m \leq n$ and $k, l < m$, one of the following statements holds.

  1. S_m is a (non-logical) axiom in A.
  2. S_m is the conclusion of a unary inference rule with premise S_k.
  3. S_m is the conclusion of a binary inference rule with premises S_k, S_l.

  This approach implements a saturation where more complex formulas are built with the application of each inference rule. A crucial factor for the effectiveness of this approach is to keep the number of derived sequents as small as possible.

- Last there is a combined search strategy which is a combination of the two mentioned above. It produces a sequence proof where non-logical axioms are sought using short tree proofs. Weakening-free backward search is used to find proofs for α-subformulas of the original formula. If a proof is found, these subformulas are introduced as non-logical axioms to prove the original formula with forward search.

Egly and Tompits showed in [7] that the forward strategy can provide a polynomial decision procedure for orthologic. As `cgol` is based on the results of that paper, it implements this forward strategy. In the following, a short description of the benefits using forward search will be given.

Often simple forward search is considered slow and ineffective, but one has to take certain aspects into account to speed it up. The total search space can be reduced if only those sequents are derived as conclusions to inference rules, which contain only subformulas of the input sequent. For a further speed-up, we can even restrict those conclusions to only contain formulas in the right polarity. Deriving only those sequents which obey the above requirements can dramatically speed up the whole process. Unfortunately, to construct counter-examples the polarity restriction cannot be used in `cgol` so the process of constructing a counter-example is slightly slower than a simple proof with the polarity-restriction. An explanation why this is the case can be found in [32].

The interface between `cgol` depicted at the bottom of Figure 5 and the drawing program at the top is achieved with the help of four auxiliary files. A detailled description of these files can be found in Subsection 5.4.2. Here, only an overview of their use is given. The file to the very left labelled with `Input Formula` contains the formula for which `cgol` should find a proof. To find a proof, `cgol` uses a saturation approach as discussed above. This saturation is written to the file denoted by `Saturation File`. This file also contains the line `PROOF FOUND!` or `NO PROOF FOUND!` informing the user whether a proof

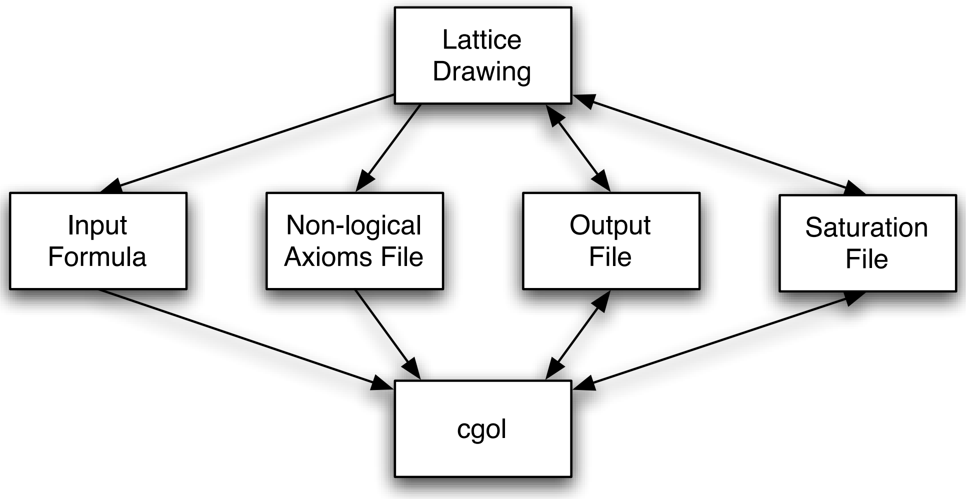


Figure 5: The Communication Between the Drawing Program and `cgol`.

was found or not. If no proof was found and the mode of operation without the polarity restriction was used, a counter-example to the formula is output to the file called `Output File`. This file is then processed by the drawing program as described in Subsection 5.4 to construct the graph corresponding to the lattice in the file and to render it to the screen. The last file not yet mentioned is called `Non-logical Axioms File`. It can be used to supply `cgol` with non-logical axioms to be used for reducing the size of the counter-example by finding equivalence classes. These equivalence classes reduce the size of the counter-example since only one representative of the whole class which may contain many elements needs to be rendered to the screen. This can also decrease the number of relations between elements and thus edges in the corresponding graph, which again makes it easier to understand the graph. The contents of this file are added by interacting with the rendered graph. By clicking on the displayed vertices a non-logical axiom can be added to this file. If it does not exist, it is created prior to adding non-logical axioms.

2.6 Concept Lattices

Concept lattices are an early form of ontologies which are important in the context of the semantic web. Such concept lattices as described in [13] or [14] are lattices which can also be rendered with our program. A *formal context* is a triple (G, M, I) where G is the set of *objects*, M is the set of *attributes* and $I \subseteq G \times M$ a relation over G and M. For a set $A \subseteq G$, we define the set of attributes common to all objects in A as $\bar{A} = \{m \in M | (g, m) \in I \text{ for all } g \in A\}$. For a set $B \subseteq M$ we define the set of objects having all the attributes in B as $\bar{B} = \{g \in G | (g, m) \in I \text{ for all } m \in M\}$.
A *formal concept* of a formal context (G, M, I) is a pair (A, B) with $A \subseteq G, B \subseteq M, \bar{A} = B$ and $\bar{B} = A$. In the following, formal context and context will be used synonymously for brevity, as well as formal concept and concept. If (A_1, B_1) and (A_2, B_2) are concepts of the same context, then (A_1, B_1) is called *subconcept* of (A_2, B_2), if $A_1 \subseteq A_2$ or equally $B_2 \subseteq B_1$. If (A_1, B_1) is a subconcept of (A_2, B_2), then (A_2, B_2) is a *superconcept* of (A_1, B_1). This subconcept-superconcept relation is a partial order relation on the set of all concepts in a given context. As stated in Subsection 2.1, a poset where the supremum and infimum of arbitrary subsets exist is a lattice. Ganter and Wille show in [13] that such a concept poset is indeed a lattice, which is called *concept lattice*. In a Hasse Diagram of such a concept lattice, the vertices are usually labelled with the concept and the connecting lines represent the subconcept-superconcept partial order relation. Therefore, we can use the same strategy for rendering ortholattices produced by `cgol` and concept lattices with the only difference that vertices are not labelled with orthologic formulas, but with concepts. An example of such a concept lattice taken from page 69 and the following pages of [13] can be seen in Figure 6. This example deals with properties of triangles.

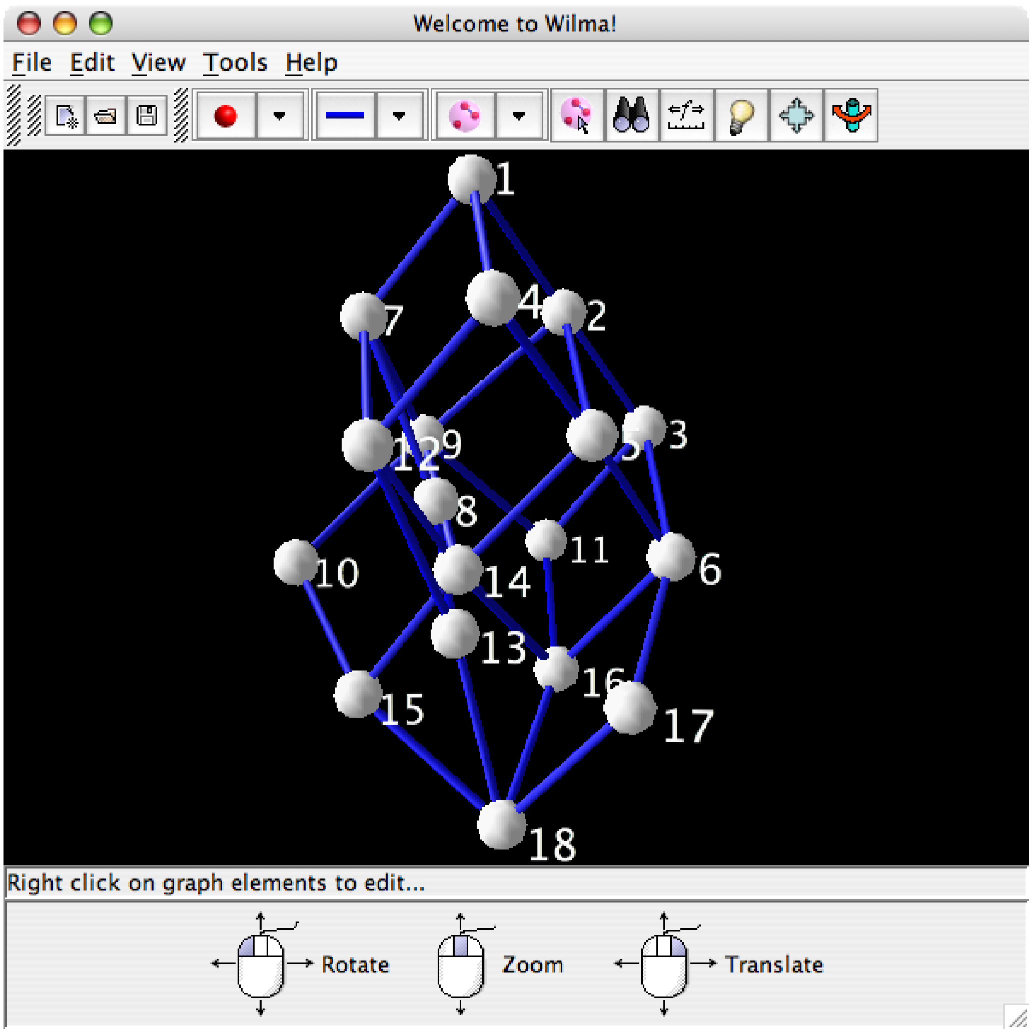


Figure 6: A Concept Lattice.

3 Psychological Aspects

The visual perception of images is organised in many simultaneous processes being performed in the retina and several portions of the brain. The reception and first preliminary processing of visual stimuli takes place in the retina. This retinal processing of isolated stimuli is not of great interest and thus its description is omitted. Since colors are of special interest, we have to investigate the use of colors based on some of these basics, though. What concerns us most is the composition of images from the stimuli and extraction of features from these images.
There are many theories concerning visual perception and there is evidence supporting all of these. The most important for this application is covered in short.
One theory which has to be mentioned here is the *Feature Theory.* It states that objects are recognised in such a manner that each object has a set of certain criteria which discriminates it from all other objects. During the process of object recognition, a check is performed which matches the properties of the perceived object to the set of properties ascribed to an object in memory. For this to happen, a process of feature extraction has to be performed prior to object recognition and classification. In this case, the user has to recognize the circles and edges in the diagram as distinct shapes. Though this theory is not the most accurate one, it provides an idea of what has to happen to extract a Hasse Diagram from a picture. Once the picture is split up into all the simple features, all the perceived features need to be grouped together to cope with all the information conveyed. This is where the next subsection comes into play.

3.1 Gestalt Psychology

According to [8], a process vital to perception is perceptual segregation. It separates parts of the visual information which belong together from each other and so enables the viewer to work with these seperate objects. Then, to

reduce complexity and find relations among these seperate objects, a process of grouping is performed according to [3]. How this grouping is performed depends on innate and learned factors.
The Gestaltist approach was one of the first to systematically study this process. This school consisting of the German psychologists Koffka, Köhler, Wertheimer and others proposed one important law concerning this perceptual segregation and grouping.
This *Law of Prägnanz* as found in [21] states:

> Of several geometrically possible organisations that one will actually occur which possesses the best, simplest and most stable shape.

This principle applied to the drawing of Hasse Diagrams is somewhat problematic. What is the best, simplest and most stable shape can be defined in many different and equally acceptable ways. Let us consider an example. With the analogy of the force-directed layout scheme described in Subsection 4.2, a definition of these terms becomes clearer. A *stable* shape is one that minized the total force of the system. The *simplest* form is that which exhibits the least crossing of edges and regular distances between vertices. Usually symmetric drawings are also perceived as simple. According to the above, the *best* drawing is one that satisfies all the above criteria.

3.2 Nice-looking Diagrams

If we are looking at pictures, we can distinguish two kinds of them—those which please the eye and those which do not. Of course, this classification is highly dependent on the person viewing the image. Yet, we know from research in the field of psychology that there are a number of generally applicable criteria which influence the perception of the image. These are the same for each viewer. Some of these criteria are listed in the following subsections. This enumeration is by no means exhaustive, but covers most of the issues which are relevant to the layout algorithm described in Subsection 4.2.

In [3], we can find a short summary of the history of aesthetics.
The first approaches to aesthetics date back to ancient Greece. As an example, we will first quote Aristotle here.

> “A master of any art avoids excess and defect but seeks the intermediate and chooses this.”

This observation can be seen to relate to layout algorithms in the following way. A *“master layout algoritm”* seeks to avoid excessively long or short edges, excessively acute or obtuse angles or an excessively irregular arrangement of vertices. What is missing here is the regular arrangement of vertices. A regular arrangement is that which we get when we find the intermediate. This is also the configuration proposed by the Gestaltists. The same line of thought can be found in Leibniz’s definition of *perfection.*

> “...to obtain as much variety as possible but with the greatest order that one can.”

Aesthetics were also examined from an information-theoretic point of view. Birkoff proposed a formula to measure the aestetic value M of objects.

$$M = \frac{O}{C} \tag{6}$$

O denotes the order and C the complexity of an object. The order of an object can be considered as a measure for the regularity or symmetry or any other property reflecting some sort of systematic arrangement. He also supplied formulas to estimate the order and complexity of objects like polygons, music and so on. Thus, the aesthetic value of an object is directly proportional to its order and inversely proportional to its complexity. Later, this equation was adapted by Gunzenhäuser where O and C are transformed into information-theoretic quantities. C is regarded as the uncertainty or entropy H and the order O is defined as redundancy $R = (H_{\max} - H)/H_{\max}$. Substituting this into Equation (6), we obtain Equation (7).

$$M = \frac{1}{H} - \frac{1}{H_{\max}} \tag{7}$$

This later adaption better reflects the findings of Eysenck and Davis which state that the aesthetic value reaches a maximum when Birkoff's M is neither too high nor too low.
Nice looking diagrams are not only pleasing for the eye, but they also simplify the comprehension of the information in the image. In the case of automatically layouted lattices, the intention is to convey the structure of the lattice as fast and easy as possible. Purchase shows in [24] that diagrams which exhibit a certain degree of symmetry, which have straight lines and few edge crossings are more readable than diagrams with less symmetry, curved lines and many crossing edges. Some of the featured properties are discussed in the following subsection.

3.3 Criteria for Nice-looking Diagrams

The following criteria and their impact on readability are taken from [24]. Some more information can be found in [3]. The latter is a survey of different studies on aesthetics and complexity in visual and acoustic arts.

3.3.1 Symmetry

This seems to be one of the most important criteria for eye-pleasing pictures. Therefore it is essential that the diagrams drawn by the program exhibit as much symmetry as possible. Lattices often have a high degree of inherent symmetry; thus, it is natural to reflect this symmetry in drawings of lattices. Symmetrical pictures are not only better-looking than asymmetrical ones, they were also found to help viewers in quicker understanding of the information displayed. Therefore, a layout scheme which favours symmetrical layouts over asymmetrical ones should be used.

3.3.2 Line Length

As it was implicit in the previous section, the length of the edges should be uniform or at least exhibit a small variation of length. To avoid extremes in line length and attain the goal of small variations in line length, the layering of the vertices should respect this requirement. Not only the layering should be determined in a way to achieve uniform edge length, but even more important, the vertices in each layer should be arranged in a way that promotes uniform edge length. Usually, drawings with a small variation in line length also exhibit a high degree of symmetry and vice versa. Since a small variation in line length and symmetry are both beneficial to aesthetic pleasure, this is a very nice property.

3.3.3 Edge Crossings

In [24], a study is mentioned which investigates the effect of edge crossings on the effectiveness with which users interpret given graphs. It turns out that the number of crossings has by far the greatest influence on the perceived complexity of a graph drawing. Graphs with many crossings have another negative property. When asked for the more aesthetically pleasing picture, test persons preferred drawings with fewer crossings over drawings with more crossings. This can be viewed to support the Gestaltist Law of Prägnanz quoted in Subsection 3.1. The test persons preferred the simpler arrangement with less crossings over the more complicated one with more crossings. This implies that a considerable effort has to be made to find a graph with as few edge crossings as possible. However, it is known that finding a layout with a minimum number of edge crossings is NP hard, which calls for a heuristic to be employed. One heuristic that proved to work relatively well is a force directed layout scheme like the Spring Embedder discussed in more detail in Subsection 4.2.

3.3.4 Slope Angles

In [24], the effect of slope angles is also investigated. The hypothesis there is that maximizing the minimum angle between edges leaving the nodes in a drawing increases the understandability of a graph. The data suggests that this aesthetic does not have a significant effect on the understandability.

3.3.5 Colours

Though this is not a criterion related directly to graph layout, it needs to be discussed in this context since it has an effect on the perceived aesthetic value of a drawing. The use of colours needs to be considered from a physiological and psychological point of view. We first discuss the physiological aspects. Some colours do not go well together, since the eye needs to adapt to the different wave-lengths that are refracted at different angles in the lens. Another reason why two colours should not be used together is that the contrast between two colours may be too small and it becomes difficult or impossible to distinguish between two seperate objects and they are perceived as just one object.

When choosing colours, one has to take into account psychological considerations and what has to be achieved with the colouring. Is is necessary to convey information or just to please the eye? Is it used to raise attention and focus the attention of the viewer on something special or to calm him or her down? All these aspects need careful consideration and the literature on this matter like [3] gives some hints on which colours to choose when. The colours red and orange were found to stimulate the viewers attention, excitation level and alertness. The colours on the blue-green end of the spectrum and pastel colours were found to have a calming effect on the viewer. In [18], we can also find a suggestion that bright and saturated colours are more pleasing than dull ones.

The assumption for the state of the user is that he or she is in a state of excitement and curiousity to see what the lattice will look like and to interact with it. There is no need to raise even more attention and so a calming color is chosen for the vertices and edges which also yields a maximum contrast to the background. When the user interacts with the displayed lattice and tries to add edges, it is necessary to inform him or her about the results of his or her actions. It is necessary to convey which vertices are selected to become the ends of the newly added edge. Also, the new edge needs to be highlighted to be easily distinguished from the originally present edges.

3.4 Perception of Three-Dimensional Graph Drawings

With many sophisticated graph drawing algorithms well suited for 2D drawings, like those proposed in [12] or [29], one question remains. Which benefits can be gained from using three-dimensional drawings instead of two-dimensional ones?

One answer can be found in [26]. This reference reports on an experiment to investigate the acceptance of web browsers. Graphs of the content of the `www.snap.com` website were presented to the test persons. They were asked to perform tasks of different types and with varying levels of difficulty. To complete these tasks, the test persons could choose between the novel representation in 3D hyperbolic space and the traditional tree-browser with collapsable view known from the file browser in various operating systems. See [23] for details on the layout in hyperbolic space. The test persons could also use the user-interface presented by all modern web browsers, namely clicking on links and entering keywords into search fields. This study shows that tasks can be completed faster with the novel 3D interface than with the traditional one. This faster completion does not have an adverse effect on the level of consistency. This is a measure for how much the test persons agreed on the results. It can also be interpreted as the number of errors the test persons made, caused by shorter completion time. In other words, 3D representations of graphs can help users to understand the structure of the graph more quickly.

Another answer to the above question can be found in [31]. This paper describes another study of the effects of three-dimensional display of graphs in contrast to the traditional two-dimensional one. In the experiment, test persons had to trace a path in a given graph, which was presented in different ways. The first presentation was in two dimensions where the three-dimensional drawing was projected onto a plane. This drawing could not be rotated or otherwise interacted with. The second way the drawing was presented to the test persons was with the help of stereo glasses. With the help of these glasses, the user perceives the displayed structure of the graph as three-dimensional. Test Persons were presented with graphs of different sizes and had to decide whether two randomly highlighted nodes in the graph were connected by a path of length two or no path. It was found that the test persons could cope with a graph increased in size by a factor of 1.6 compared to the original graph. If it was also possible to rotate the graph, the increase is almost doubled to a factor of three. If there was no stereo vision but rotation provided, the increase is still by a factor of 2.2 compared to the original graph. This leads to the following two conclusions.

- If we cannot render a graph in three-dimensional space, then, if we at least provide means to rotate it, it is easier to understand graphs or it is possible to understand larger graphs.
- If we can render a graph in three dimensions, but cannot provide means to interact with it and rotate it interactively, the gain is only moderate.

Therefore, the best results can be achieved by combining 3D rendering and interactive translation, rotation and zooming of the drawing.

4 Layout Algorithms

As it was pointed out in the previous section, force-directed layout algorithms exhibit some very nice properties. They perform quite well in terms of conforming to the aesthetic criteria stated above. Thus, the algorithms used for laying out the graph in the program described in Section 5 also belong to the family of force-directed layouts. They are adapted such that they allow the employment of level constraints. This means that vertices can be assigned to planes parallel to the x-z-plane. They cannot leave these planes during the layout process. This is extremely useful in the case of Hasse Diagrams where this kind of layering is desired. Details on the algorithms implemented in WilmaScope are described below. First, the optimization scheme known as Simulated Annealing is introduced. Then, two force-directed layouts are discussed. To aid the discussion on how algorithms perform on certain graph instances, the following notation will be used. The number of vertices in a graph G will be given as n and the number of edges in G as m.

4.1 Simulated Annealing

This approach to tackle NP hard problems was first introduced in [20]. Though this idea is adapted from statistical mechanics and originally applied to combinatorial optimization, it can be adapted for graph drawing. At first, we will introduce the central ideas of simulated annealing and then show how it can be used in the context of graph drawing.
Statistical mechanics is a body of methods for analyzing the aggregate properties of large numbers of atoms in solids and liquids. Each of these atoms has its own set of properties like its location, momentum or electrical charge. The whole state of the solid or liquid is determined by the properties of each of its constituting atoms. These properties change over time according to probabilistic laws. Thus, such a solid or liquid can be in one of a large number of states. Because of the large number of atoms involved, only the most probable state of the system at a given temperature is observed in experi-

ments. The temperature of solids or liquids can be defined in relation to the momentums of its constituting atoms. This allows many different states, as the momentum of each atom changes after collisions with other atoms. The most probable state observed in experiments is that of each atom having an average momentum with small fluctuations about this average. How large this fluctuations are can be estimated by the Maxwell-Boltzmann distribution. This assumes idealized particles to be identical point masses. Cooling the matter means reducing the momentum of the particles until they eventually freeze and stop moving. We are interested in the state where the particles rest in the end. The aim of experiments in condensed matter physics is to create solid crystals with an atomic grid structure which is as regular as possible. These states of matter are called ground states. They have an interesting property, namely a minimal energy configuration.
Let us consider the simple example of n identical particles lined up in a row, each carrying either positive or negative unit charge. Each charge is equally likely. The interaction energy between two particles p_i and p_{i+1} is denoted by J_i. The total energy of the system is given as $\sum_i J_i$ and it is in the range between zero and $|nJ|$, where $|\cdot|$ denotes the absolute value. It was proven that the probability of a system having a specified energy follows the binomial distribution. We are interested in the rare states having maximal or minimal energy as opposed to the large number of zero energy states. The ground states with maximal or minimal energy are those which are also important for graph drawing. The vertices are the particles we want to position. The interrelation energy can be thought of as a cost function taking into account the deviation of the distance between connected vertices from a desired value, the number of edge crossings and so on. A ground state of the graph is an arrangement which best satisfies all the requirements. A defect of this analogy is that all atoms are assumed to be identical, yet all the vertices in a graph are different in that they are incident to a different number of edges.
The process of simulated annealing aims to first position the vertices ran-

domly and then move them around with high momentum. In this phase the general structure of the graph should become visible. Then the momentum of the vertices is decreased and only minor improvements are allowed to get a finer and more symmetric placement.
The main difference to other layout algorithms based on the greedy heuristic is most apparent in the early stages when the system is hot. In these stages, the total energy is allowed to increase with a probability dependent on the heat of the system. This can help avoid local minima and reach a better layout in the end.
The downside of this approach is that a cooling scheme has to be chosen. On the one hand, to get the best results, a slow cooling scheme is needed. On the other hand, a slow cooling scheme requires a lot of computation. Therefore, it is difficult to find a cooling scheme which is an acceptable compromise between satisfactory results and an acceptable amount of computation.
This idea of finding ground states can be adopted for combinatorial optimization in the following way. In contrast to a greedy heuristic which tries to improve the solution in every step, simulated annealing can be used. The idea is to model the system as the physical system of a cooling liquid. In a hot liquid, the molecules have a great kinetic energy and thus move very fast. As the liquid cools down, this kinetic energy of the molecules gradually gets smaller and the speed at which they move about is getting smaller as well until the liquid eventually freezes and all molecules are confined to a place in a grid-like structure and do not move any more.
For this to work, an application-dependent cooling scheme has to be devised. Usually, such a cooling scheme is computationally expensive, but it yields competitive results.

4.2 Spring Embedder

This algorithm for laying out graphs was proposed in [5] and later optimised in [10]. The idea is to model a graph as a mechanical system of springs. Each vertex is thought of as a steel ring and each edge is modelled as a spring.

All vertices are laid out either randomly or with a certain heuristic to get an initial layout. Usually, heuristics which place nodes very close to their final position yield better results than totally random placement. Yet, care has to be taken not to constrain the initial layout too much or to place the nodes too regularly. This might have an adverse effect on the performance of the algorithm, as it might induce unstable equilibria. They might cause the algorithm to get stuck in suboptimal local minima. Depending on the way the initial layout is obtained, a strategy has to be chosen to minimize the total energy of the springs and to avoid these local minima. Minimizing the energy of the springs is done by adjusting the position of the vertices in a manner such that adjacent vertices are attracted to each other and vertices which are not connected via an edge (or sometimes via a path) are repulsed. The algorithm is time-discrete. During each step of the iteration, all attractive and repulsive forces on a vertex are computed. At the end of the iteration, the vertex is moved according to the resulting force, which is the sum of all the forces on a certain vertex. In some cases, the algorithm may get stuck in an unstable state. This state can have two forms. It can either be a local minimum or the vertices my oscillate and thus preventing the algorithm from reaching a stable state. To keep the algorithm from oscillating and never reaching a stable and optimal state, some other heuristic can be employed. One such example is Simulated Annealing described in Subsection 4.1. This allows the vertices to move further during the first few iterations and then, in the later iteration allows only small improvements on the position of the vertices.
What was not discussed so far is the computation of the forces. One can employ many different force schemes. The choice which one to use can be guided by the time it takes to compute the forces, by well researched physical models or by aesthetic criteria to which some force schemes conform better than others. Using a force model which is based on symmetric forces for attraction and repulsion results in drawings which exhibit a higher degree of symmetry than a drawing based on asymmetric forces.

4.3 Multilevel Force-Directed Placement

The program WilmaScope, which is described in detail in Subsection 5.5, implements the following layout algorithm taken from [30] which is an improvement of the approach described above. It tries to overcome some of the defects of the traditional force-directed algorithm.
One of these drawbacks of force-directed algorithms is that the general structure of the graph is not considered when laying out the graph. The multilevel approach tries to first untangle the graph and get the general structure right, before laying out the final graph like the force-directed approach.
The key idea here is to partition the graph to get a coarse layout of the general structure of it and then iteratively add vertices to get an increasingly finer grained layout until all vertices are added and the final layout is produced. There are many different ways to get coarser graphs of the original. The one chosen by Walshaw in [30] is known as maximum cardinality matching. Optimal solutions to instances of this problem can be computed by algorithms with time complexity of at least $O(n^{2.5})$. Since we are only interested in the layout of the whole graph, we do not need optimal solutions for the coarsening and we can use a heuristic as devised in [17].
This heuristic works as follows. The coarser graph is created by means of edge contraction. This means, if two vertices are connected by an edge, they are contracted into one super-vertex. This new vertex is then connected to all the vertices adjacent to the original vertices. To preserve some of the structural properties of the original graph, weights of vertices and edges are used. If the original graph was unweighted, then for the algorithm to work, each vertex and edge gets a default weight of 1. The weight of the new vertex is the sum of the weights of the original vertices. The weights of the new edges are computed as follows. If only one of the original vertices was connected to another vertex, the weight of the edge remains the same. If more than one edge points from the contracted vertex to some other vertex, these edges are contracted into one and the weight of this new edge is the sum of the weights of the original edges.

To find vertices to contract into super-vertices, a matching of vertices is computed. This is done in the following manner. The vertices are visited in a random order. If it is not yet matched, an unmatched neighbour is selected and these two vertices are matched. This requires time in $O(m)$. After all vertices are matched, the graph is coarsened with the contraction strategy described above. Then this coarsening is repeated until the size of the resulting graph falls below some threshold, which is typically two. This first graph to be laid out consists of two vertices connected with a weighted edge. This smallest graph is subject to the force-directed layout scheme. After the layout has become stable, the layout is refined by undoing the last step of the coarsening and separating the super-vertices into their constituting vertices. They are initially positioned in place of the super-vertex. After the super-vertices of this iteration are split up, another force-directed layout is produced. This splitting and layouting is performed, until the original graph has been laid out.
A detailled description of how the forces are computed can be found in [30].

5 The Lattice Drawing Program

This section describes the lattice drawing program based on the insights from the previous sections. To gain a general understanding of the functions of this software, the intended use and functionality of the program is discussed in Subsection 5.1. Then, a sample session with screenshots of the program in use can be found in Subsection 5.2. The development history of the program and the technical details are described in Subsections 5.3.2 and 5.4.
In this section the following typographical convention is used. Whenever referring to pictures or Java objects in the text, the text from the picture or the name of the object is set in `typewriter font`.
For brevity, another convention is in order. This section makes extensive use of Equation 1 as mentioned in [22] (reproduced as Equation (8) below) to demonstrate the use of the program.

$$((a \wedge b') \vee a')' \vee ((a \wedge b') \vee ((a' \wedge ((a \vee b') \wedge (a \vee b))) \vee \\ (a' \wedge ((a \vee b') \wedge (a \vee b))')))) = 1 \quad (8)$$

This equation arose in the work N. Megill and M. Pavičić on quantum logic. They were known to hold in orthomodular lattices, but they were not known to have a proof in orthologic. Since Megill and Pavičić did not find a proof by hand, they asked W. McCune for help. He tried to find a proof or a counter-examples with the help of the automated theorem prover otter together with the "model checker" MACE. A counter-example to Equation (8) was found in 15 minutes after using 84 MB of RAM. With `cgol`, such a failed proof search and the generation of a counter-example takes only a fraction of a second and less than 13kB of memory. A reason for this difference in the timing is that McCune used a different strategy than Zugaj. McCune used an exhaustive search strategy to find counter-examples of sizes 1, 2, 3 and so on. The smallest counter-example that McCune could find was of size 10. For details on the strategy of `cgol`, see Subsection 2.5. One lattice generated by `cgol` as the result of a failed proof search for this formula will be referred to as "McCune 1 lattice". It has 25 vertices, which is larger than the counter-example found by MACE. The size of this counter-example can be reduced by adding non-logical axioms and finding equivalence classes. Details on how this is done can be found in the remainder of this section.

5.1 Intended Use and Functionality

This subsection can be seen in two ways. First, it can serve as a rough specification of requirements what the program should achieve. Second, it serves as a description of the intended use cases. A graphical representation of this subsection can be found in Figure 9 to which we will refer throughout the remainder of this subsection. This figure is a UML use-case diagram conforming to version 2.1.1 of the UML specification.

In this subsection, we will stick to the following typesetting convention for grammars. Text set in **bold** face denotes terminal symbols, whereas text set in *italic* denotes non-terminals.

We will also stick to the following syntactic convention for the grammars in this subsection. An ϵ on the right hand side of a grammar production denotes an empty string. A parenthesized expression may be followed by one of three symbols to indicate the number of occurrences of this expression. A ? indicates “no or exactly one” occurrence, a + produces “one or more” occurrences and a * is a placeholder for “none or arbitrarily many” occurrences. Note that the non-terminal *Space* expands to one or more terminal whitespace characters excluding newline characters.

The user can choose a file which contains the input to the program. This input can be of two forms:

- The file contains a lattice in the notation defined in Figure 7 below. The intention is to draw the lattice given in the file.

- The file contains a formula in the notation given in Figure 8. Here, the intention is to try a proof with `cgol` first and in case the formula is unprovable, present the counter-example as a lattice.

If any other input file is chosen, the program will not process it and output an error message to the user. For simplicity, all the error handling is left out of the following discussion and consequently it is not shown in Figure 9. Let us consider the simple case, where the input conforms to Figure 7 and consists of a lattice or a lattice-like structure. In this case, we follow the lower `extend` arrow to the `Draw Lattice` use case. This means that, if we encounter this form of input, the lattice is constructed and rendered to the screen. Nothing more can be done in this case.
In the other case, where the input is an orthologic formula, we follow the upper `extend` arrow to the `Search for Proof` use case. The program `cgol` tries to prove this formula. If a proof is found, it is output to the user. If not, a counter-example is constructed. This counter-example has the form of a lattice. We follow the next `extend` arrow to `Draw Lattice`. Now, the lattice obtained as a result of the failed proof search is rendered to the screen. Once rendered to the screen, the user can then try to reduce the lattice. This reduction can be done in two ways, either automatically or manually. This is depicted in the use case called `Reduce Lattice` to the very right of the figure. This use case is optional and so is connected with an `extend` arrow. The reduction can be performed either automatically or interactively. Non-logical axioms can be added to reduce the size of the lattice. How this can be done is described in detail in Subsections 5.2.4 and 5.2.5. The result of this reduction is again rendered to the screen, as implied by the `extend` arrow in the reverse direction. This rendering is not mandatory, since the reduction might result in the formula becoming provable and then there is nothing to render. In this case, an appropriate message is displayed. The reduction step can be repeated to find smaller counter-examples or until the formula becomes provable. In any case, some undo/redo functionality is desired to improve user-friendliness. For simplicity, this is left out of the diagram.

The interaction between `cgol` and the drawing program is given in Figure 10 in the notation of a flowchart. We proceed from left to right. To the very left,

$$
\begin{array}{rcl}
\mathit{Inputfile} & \rightarrow & (\ (\mathit{Vertex})^*\ )\ (\mathit{Subformulalist})? \\
\mathit{Vertex} & \rightarrow & (\ \mathit{Index}\ (\ \mathit{Cover}\ )\ ) \\
\mathit{Index} & \rightarrow & (\ (\mathbf{c})?\ ([\mathbf{0}\text{–}\mathbf{9}])+\ )\ |\ \mathbf{top}\ |\ \mathbf{bot} \\
\mathit{Cover} & \rightarrow & \mathit{Index}\ (\mathit{Space\ Index})^*\ |\ \epsilon \\
\\
\mathit{Subformulalist} & \rightarrow & \textbf{Subformula:}\ \mathit{Index}\ \textbf{is:}\ \mathit{Formula}
\end{array}
$$

Figure 7: Grammar for a File Containing a Lattice.

we can find the orthologic formula we want to prove. We run `cgol` with this formula as an input. Then we have two scenarios. If `cgol` finds a proof, then a message is output to the user telling him or her that a proof was found. If no proof could be found, then `cgol` constructs a counter-example to this formula. This counter-example is then handed to the drawing program and rendered to the screen. We are now in the upper right corner of the diagram. From this stage on, we can try to find a smaller counter-example. This can be achieved in two ways which are both supported by the drawing program.

- The user can try to reduce the size of the lattice by introducing non-logical axioms.
- `cgol` can be told to perform a reduction based on its built-in heuristics.

In any case, `cgol` is called again to search for a proof with user-supplied non-logical axioms or with its heuristics. The result can either be that the formula becomes provable, which means that we have to go a step back and try other non-logical axioms if possible. The result can also be a smaller counter-example, which is again handed to the drawing program and we can repeat the procedure of finding non-logical axioms, running `cgol` with these non-logical axioms and getting a drawing of the result.

$$\begin{array}{rcl}
\textit{Input Sequent} & \rightarrow & (\textit{Formula} <)?\ \textit{Formula} \\
\textit{Formula} & \rightarrow & (\ \textit{Atom}\ ) \mid \sim (\ \textit{Formula}\ ) \\
 & & \mid \&\ (\ \textit{Formula}\ ,\ \textit{Formula}\ ) \\
 & & \mid \mid (\ \textit{Formula}\ ,\ \textit{Formula}\ ) \\
\textit{Atom} & \rightarrow & (\ [\mathbf{a}\text{–}\mathbf{z}]\ )+
\end{array}$$

Figure 8: Grammar for a File Containing an Orthologic Formula and an Input Sequent.

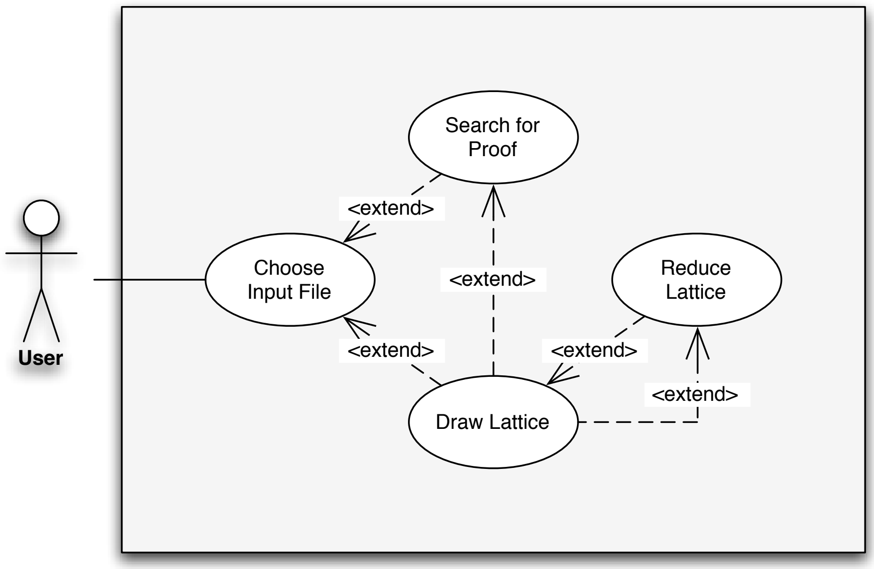


Figure 9: Intended Use of the Program.

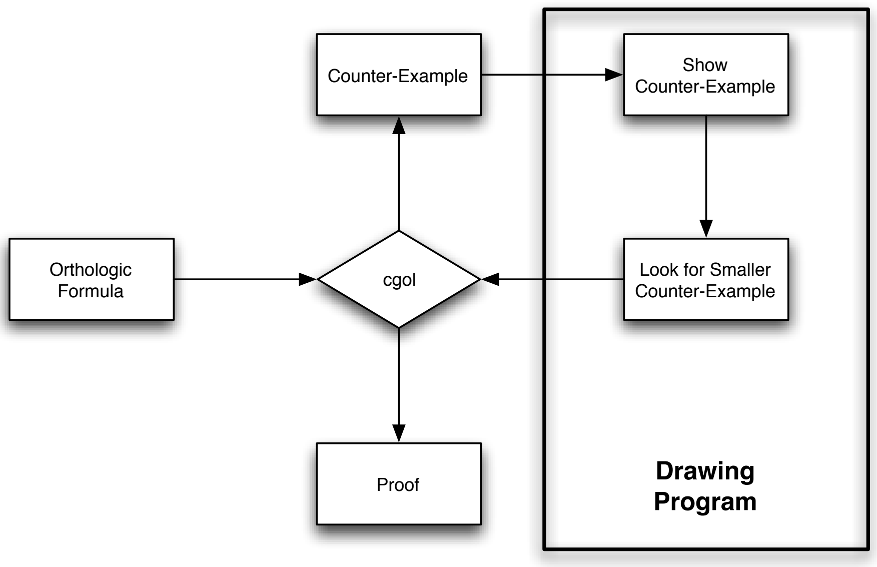


Figure 10: Flowchart of the Interaction Between `cgol` and the Drawing Program.

5.2 A Sample Session

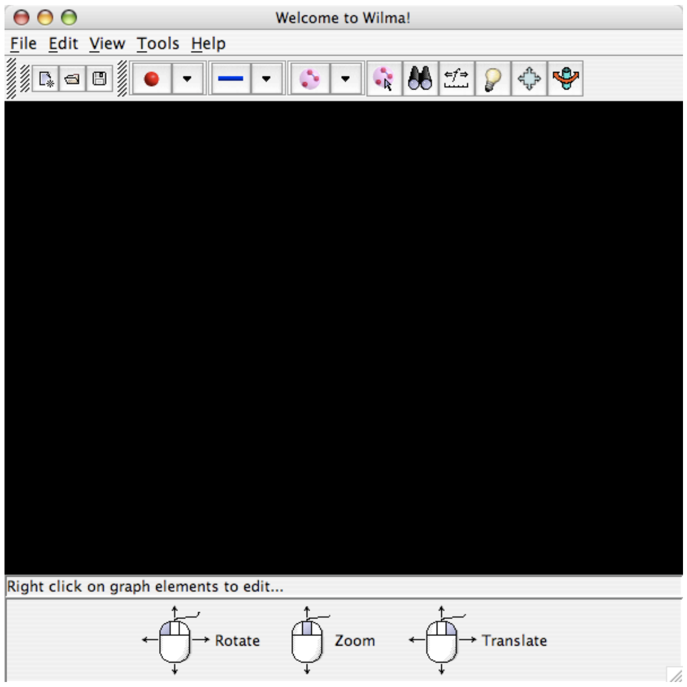
In this subsection, the use of the program with the help of screenshots is demonstrated. The use cases will be listed one by one as described in Subsection 5.1. We will also demonstrate the effect the algorithm for computing the transitive reduct has on the overall readability of the graph.

5.2.1 Drawing a Lattice

For this use case, we assume that we already have a file containing a lattice in the format described in Figure 7. In Figure 11(a) we can see the program after startup before any user-interaction is performed. The next step is then displayed in Figure 11(b). The dialog to select a file to open is shown. Note the type of files to select. Besides the filter for all files and for the `.xwgl` format that WilmaScope understands, there is a third to select lattice files and input files containing orthologic formulas. This third filter is selected by default. We will select a file called `mccune1.lat` which holds the McCune 1 lattice as calculated by `cgol` as result of a failed proof search for Equation (8). Figure 12 shows two versions of this lattice. In Figure 12(a), we can see the transitively reduced version. Figure 12(b) shows what the lattice looks like when no transitive reduction is performed and a lot of edges remain. These two figures show the tremendous effect the transitive reduction has on readability. In the upper figure, the structure of the lattice is visible. In the lower figure, the structure of the lattice is obscured by the edges in the center of the lattice. This concludes the use case of drawing a lattice. Next, the use case of proving an orthologic formula is discussed.

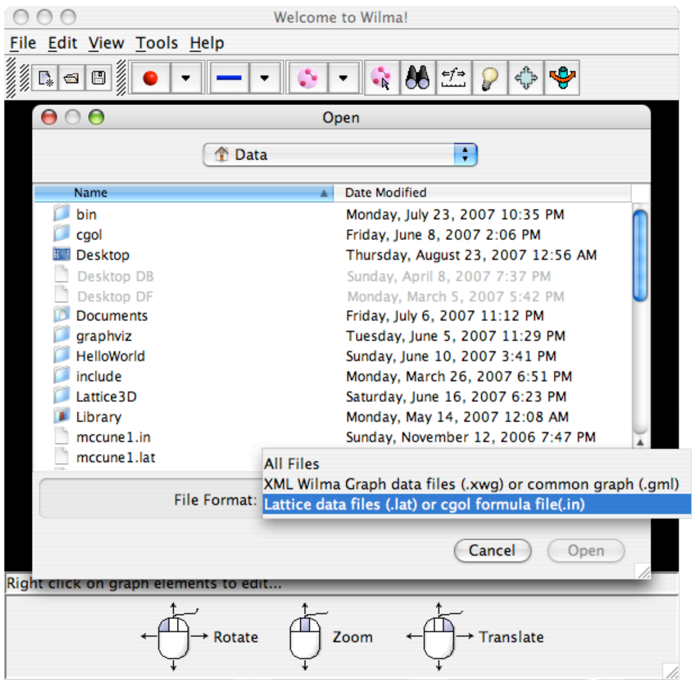
5.2.2 Proving a Formula

The starting point here is an orthologic formula given in the format described in Figure 8. The typesetting conventions of the grammar are given in Subsection 5.1. The input file containing the formula is chosen in a way similar to the one described in Subsection 5.2.1. If the proof search was successful,



Right click on graph elements to edit...

(a) Before Any User-Interaction.



Name	Date Modified
bin	Monday, July 23, 2007 10:35 PM
cgol	Friday, June 8, 2007 2:06 PM
Desktop	Thursday, August 23, 2007 12:56 AM
Desktop DB	Sunday, April 8, 2007 7:37 PM
Desktop DF	Monday, March 5, 2007 5:42 PM
Documents	Friday, July 6, 2007 11:12 PM
graphviz	Tuesday, June 5, 2007 11:29 PM
HelloWorld	Sunday, June 10, 2007 3:41 PM
include	Monday, March 26, 2007 6:51 PM
Lattice3D	Saturday, June 16, 2007 6:23 PM
Library	Monday, May 14, 2007 12:08 AM
mccune1.in	Sunday, November 12, 2006 7:47 PM
mccune1.lat	

File Format:
All Files
XML Wilma Graph data files (.xwg) or common graph (.gml)
Lattice data files (.lat) or cgol formula file(.in)

Right click on graph elements to edit...

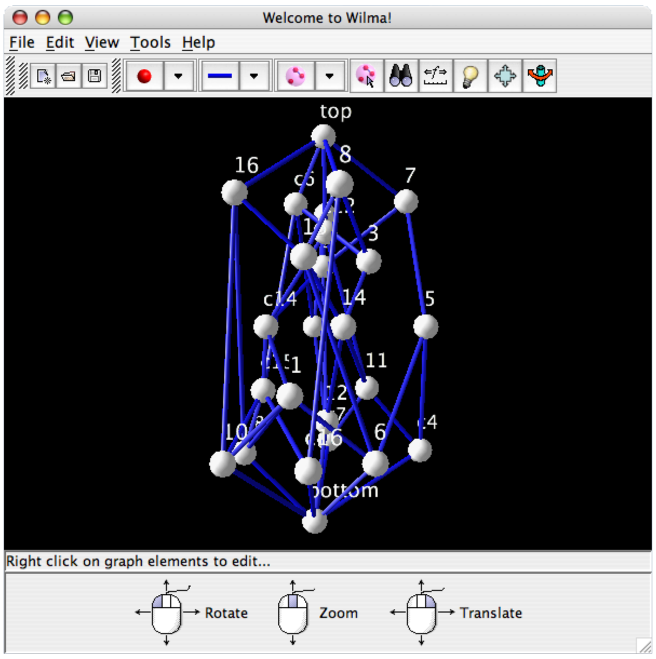
(b) Dialog to Open Files.

Figure 11: Two Screenshots of the Drawing Program.

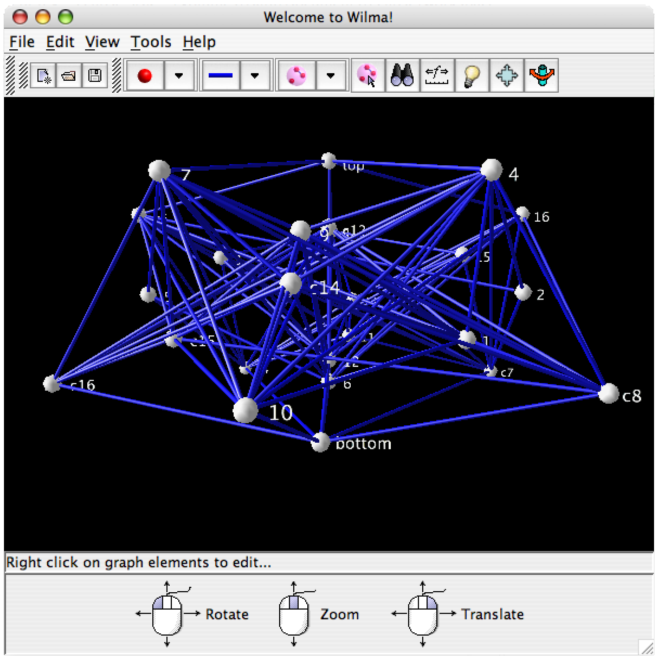
a message is displayed that a proof was found. The user can then take a look at the proof. If the proof search was not successful, `cgol` automatically creates a counter-example to the input formula. This counter-example in the form of a lattice is rendered to the screen to convince the user that the input formula is indeed not valid.
If an orthologic formula is used as input and a lattice is produced as a result of a failed proof search, some additional items in the `Edit` menu are activated. The following subsections explain how these can be used to manipulate the drawn lattice.

5.2.3 Showing Formulas

This is the first item in the `Edit` menu that becomes accessible, if a lattice is generated by `cgol`. If `cgol` produces a lattice, then a formula can be assigned to each element of this lattice. It is important to see how these formulas are interrelated and which formula is assigned to which vertex. It would seem natural to label each vertex with the corresponding formula. Unfortunately, these formulas can become fairly large and clutter up the drawing. An example of such a bad drawing is given in Figure 13. To view one particular formula as a whole, it would be necessary to zoom out of the scene, but then the writing gets too small to be read easily.
The solution here is to use something `cgol` readily provides. Each vertex is assigned a unique integer index. If a formula is the complement of another formula, it has the same index, but a "c" is prepended to the index. These indices usually are much shorter than the formulas assigned to the vertices. An example of a lattice labelled with indices can be seen in Figure 12a.
What is needed now is a way to establish a connection between formulas and vertices labelled with indices. When clicking on the `Show Formulas` item in the `Edit` menu, a frame pops up, which shows an index to the left and the corresponding formula to the right. This way it is easy to find out, which formula belongs to which vertex. A screenshot showing the frame with formulas for the McCune1 lattice is depicted in Figure 14.


Welcome to Wilma!
File Edit View Tools Help
top
16
8
7
3
14
5
11
6
bottom
Right click on graph elements to edit...
Rotate
Zoom
Translate

(a) With Transitive Reduction.


Welcome to Wilma!
File Edit View Tools Help
7
4
16
2
c14
c16
10
c8
bottom
Right click on graph elements to edit...
Rotate
Zoom
Translate

(b) Without Transitive Reduction.

Figure 12: Two Screenshots of the Drawing Program Showing the McCune 1 Lattice.

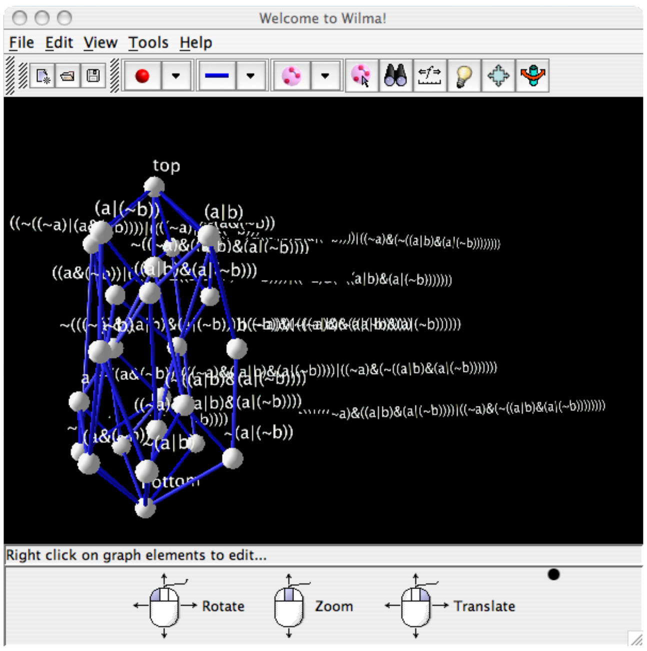
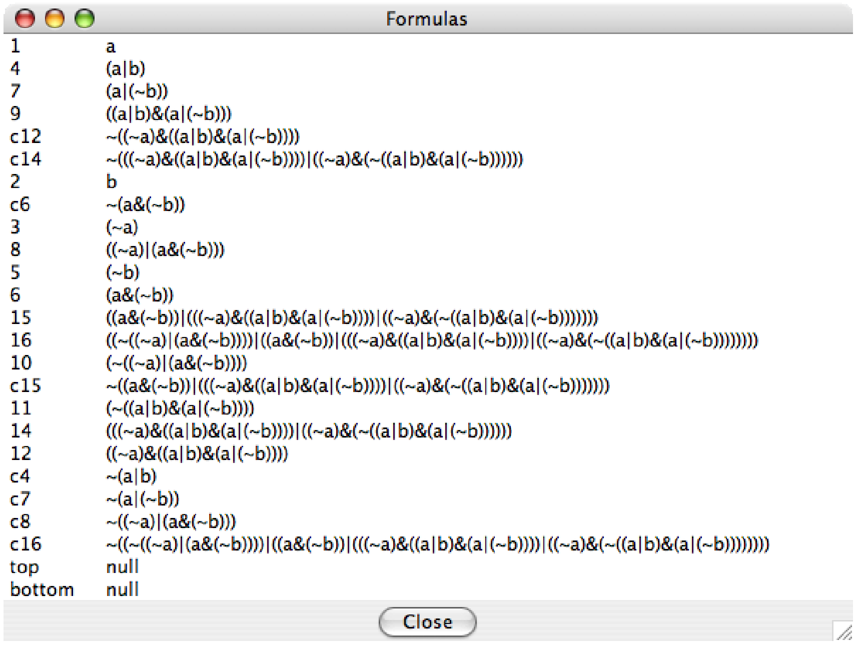

Welcome to Wilma!
File Edit View Tools Help
top
(a|(~b))
(a|b)
(a|(~b))
Right click on graph elements to edit...
Rotate
Zoom
Translate

Figure 13: McCune 1 Lattice with Large Formulas.



Formulas

```
1         a
4         (a|b)
7         (a|(~b))
9         ((a|b)&(a|(~b)))
c12       ~((~a)&((a|b)&(a|(~b))))
c14       ~(((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b))))))
2         b
c6        ~(a&(~b))
3         (~a)
8         ((~a)|(a&(~b)))
5         (~b)
6         (a&(~b))
15        ((a&(~b))|(((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b)))))))
16        ((~((~a)|(a&(~b))))|((a&(~b))|(((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b))))))))
10        (~((~a)|(a&(~b))))
c15       ~((a&(~b))|(((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b)))))))
11        (~((a|b)&(a|(~b))))
14        (((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b))))))
12        ((~a)&((a|b)&(a|(~b))))
c4        ~(a|b)
c7        ~(a|(~b))
c8        ~((~a)|(a&(~b)))
c16       ~((~((~a)|(a&(~b))))|((a&(~b))|(((~a)&((a|b)&(a|(~b))))|((~a)&(~((a|b)&(a|(~b))))))))
top       null
bottom    null
```

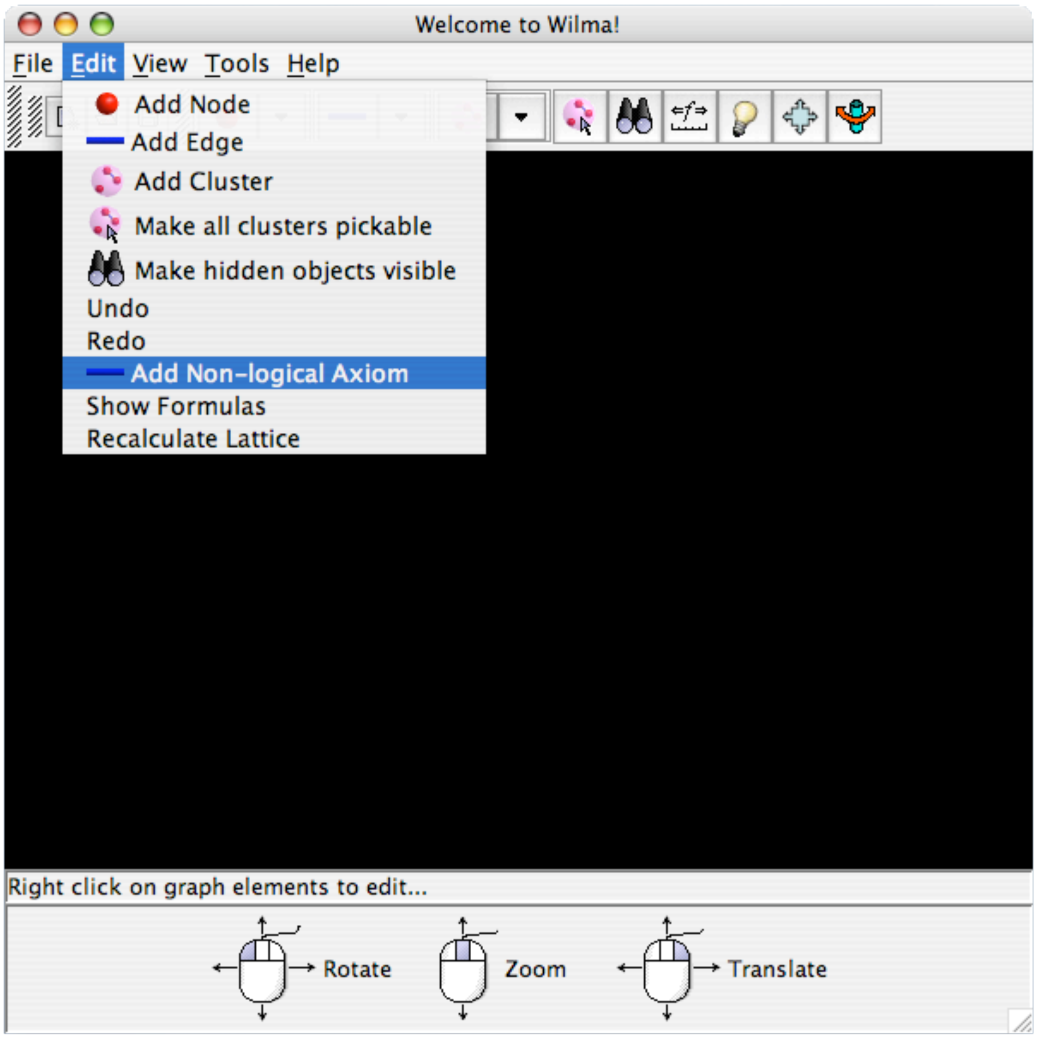
Figure 14: Frame Showing Labels and Formulas for McCune 1 Lattice.

5.2.4 Adding Non-logical Axioms

The `Reduce Lattice` use case to the right of Figure 9 is split up into two seperate actions. First, the user can add non-logical axioms, which are then used to reduce the size of the lattice. Second, the user can choose to actually compute the reduced lattice based on the non-logical axioms he has introduced earlier or with the help of a heuristic implemented in `cgol` to automatically find suitable axioms. The latter alternative is explored in detail in the following subsection. Now, we focus on the former and demonstrate the way non-logical axioms can be added manually.
This function can be accessed like all others directly manipulating the structure of a lattice via the `Edit` menu of the program. A picture of the `Edit` menu can be seen in Figure 15. The first five items there—each with an icon to the left—are WilmaScope specific functions which we will not discuss here. The lower five items with exception of `Show Formula` are those which trigger the interaction between `cgol` and the drawing program. If the user decides to add a non-logical axiom, a status bar is displayed at the bottom of the window telling the user to pick two vertices with the mouse. Once two vertices are selected, the user can click the "OK" button to the right and the non-logical axiom is added. The 3D scene is also updated with a new edge representing the non-logical axiom. This edge has a colour different to that of all other edges.

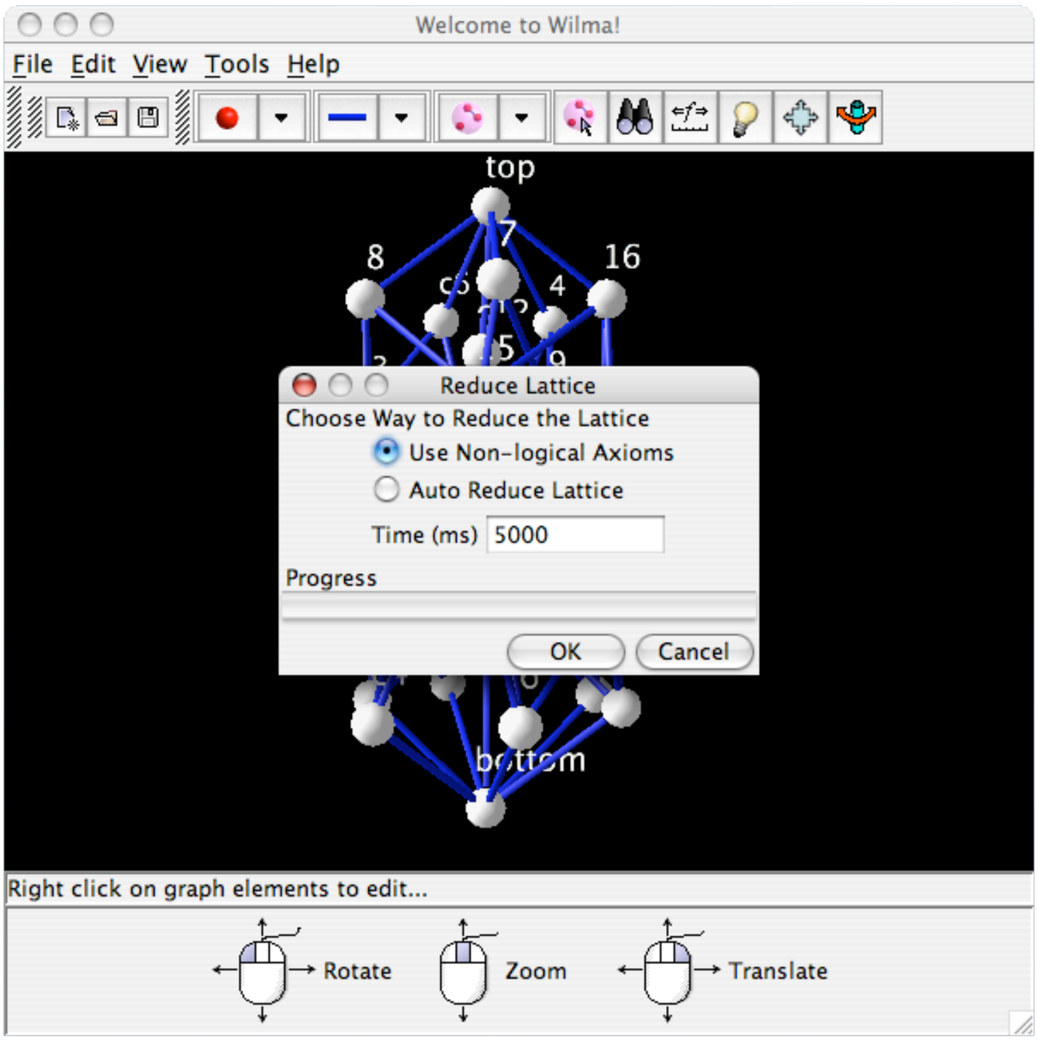
5.2.5 Calculating a Reduced Lattice

In Figure 16, we see the dialog that pops up when the user selects to reduce the lattice by clicking on the lowermost item of the `Edit` menu in Figure 15. It offers two choices, the first being selected by default. The user can either choose to reduce the lattice on the basis of non-logical axioms he has added earlier. The second choice available is to use the heuristics built into `cgol` to find a reduced lattice. Here, the user can specify a time limit as a number of milliseconds. This time allows the heuristic to look for a reduction, see [32] for details on the heuristics. In the lower part of the dialog, a progress



Right click on graph elements to edit...

Figure 15: The Edit Menu of the Drawing Program, Add Non-logical Axiom Highlighted.


Welcome to Wilma!
File Edit View Tools Help
top
8
16
4
5
9
bottom
Right click on graph elements to edit...
Rotate
Zoom
Translate

Reduce Lattice

Choose Way to Reduce the Lattice

[✓ Use Non-logical Axioms]

Auto Reduce Lattice

Time (ms) 5000

Progress

OK Cancel

Figure 16: Dialog for Computing a Reduced Lattice.

bar can be found. It informs the user of how much of the time has already passed and when to expect a result.
After the time has elapsed, the reduced lattice is rendered to the screen, if a reduction could be found. In some cases, the non-logical axioms can render a formula provable. In this case, a message is output to the user informing him or her, that the proof search was successful and the counter-example cannot be compressed by the given non-logical axioms.

5.3 Evolution of the Program

> "Do it badly, do it quickly, make it better, and then say you planned it." – Tom Peters

5.3.1 Rapid Evolutionary Prototyping

To realize the functionality described in the previous subsection, a program was developed according to the rapid prototyping approach described in [1]. This seemed the most promising choice for the following reasons. In this book on page 56, a list of benefits of this approach compared to other development processes, like the waterfall model, can be found. We will only quote the four most important benefits pertaining to this project and explain how and why they are relevant in this context.

> 2. Reducing risk by eliminating uncertainty. The initial system is often created with fewer people in less time. Cycle time to proof of concept is dramatically reduced.

Before discussing the element of uncertainty, we will start with the second part of this quote, namely that an initial system can be created within fewer man hours. Since this is a one-man project, the idea of saving time to create something tangible was very attractive. Especially as the project started off with a lot of uncertainty. It was not clear, what exactly the system has to do and how this should be done. The general idea was that we have an orthologic formula and we have a prover which can either prove it or find a counter-example in the form of a lattice. If we find such a lattice, we want to draw it nicely onto the screen. The choice which technology to use had to be made without much experience in the field of 3D visualization. All these reasons lead to the next point taken from the book.

> 4. Incorporating a learning process into the development process. Since we know that we are operating on *incomplete knowledge* whenever we start a development process, rapid evolutionary development encourages us to learn as we go, backtracking and changing things until we get them right. It encourages rather than stifles change. Frozen requirements cannot reflect the dynamics of the organization or market.

This project was started with *incomplete knowledge* as far as the functionality was concerned, but this was a minor issue. Much of the technology for 3D rendering needed to be investigated as well. Using a development process which encourages learning, not only from the books, but most importantly by trial and error, is really helpful. As this project combines many different programming languages, this approach especially helps to control complexity by developing different modules and then plugging them together one after the other. Not all the required features needed to be known at the beginning, but either revealed themselves when testing the prototype or asked for by the users. As the project proceeded, new or different approaches to some problems were discovered, which is summed up by the next point.

> 5. Encouraging discovery and serendipity in the development of desired functionality. If we learn as we go, there is a much greater chance of discovering opportunities along the way that will shape the course of the system and possibly the course of the company.

As stated above, a development process which needed less planning at the beginning and which is capable of incorporating things found in the course of the project into the final product was desired. This property was extremely helpful in this project. WilmaScope was regarded as a starting point at the

very beginning, but found too complex. Yet, after a few months developing a prototype with similar functionality and more familiarity with Java3D, its usefulness was investigated again. This time it was found flexible enough to become an integral part of the software and it has been adapted for this special project. This would not have been possible, if the waterfall process had been used.
Another defect of the waterfall process model is that testing is performed at the very end of the coding phase. If a serious flaw in the software is detected, it is very hard to *"swim upstream"* and back to the planning and coding phase. The rapid prototyping approach is better suited since coding and testing occur in short succession and all the bugs found during testing can be dealt with in the next iteration of the evolution. This is summed up in the following quote.

> 7. Reducing defects through continuous testing and evaluation of the system components during the initial prototyping and ongoing evolutionary phases. User manuals and training can be developed *using* the working prototype to ensure accuracy.

This project not only involved writing a piece of software, but also documenting it in this thesis. For this reason, it was very helpful to always have a working prototype at hand, to keep the documentation of the software accurate.

5.3.2 Chronolgy of the Implementation

Now we will give a short chronological description of how the program evolved over time. As stated above, the rapid prototyping approach was used. This means that the program was developed in stages. Each time some functionality was added it was immediately tested thoroughly. If and only if it was found to work satisfactorily, the next stage was entered.
The first prototype was just a simple graphical user-interface to become familiar with interface development with the Java language. This non-functional

user-interface was then extended by adding some simple Java3D rendering facilities and simple user-interaction. This phase largely exploited benefit 4. from above as it involved most of the learning needed for the later development. During this phase, knowledge of 3D-visualization and the data structures in Java3D was acquired. A summary on how Java3D handles the data needed for rendering is given in Subsection 5.4.1.

The next step was to integrate `cgol` to automatically generate lattices to be drawn. This was problematic for three reasons. First, `cgol` has to be called and provided with some input. Next, the output of `cgol` has to be read and used. And at last, if the output of `cgol` is a lattice, it has to be drawn. The interaction between the drawing program written in Java and Java3D and the `cgol` program written in ANSI C is described in detail in Subsection 5.4.2.

As soon as `cgol` could be called from the drawing program, it produced output which had to be processed. The detailed structure of the files used for the input to `cgol` and the output from `cgol` can be found in Subsection 5.4.3. To faciliate the processing of the files, a parser was implemented. This parser introduced JavaCC, a parser generator, as the fourth technology besides Java, Java3D and ANSI C. A detailed description of the parser which scanned the output of `cgol` to make the graph data available is given in Subsection 5.4.4.

Now was the time to implement the algorithms on lattices mentioned in Subsection 2.4. Still, no graphs were rendered onto the screen. To test the correct working of the algorithms, output files had to be checked manually. As this was quite tedious, a better way to visualize the results of these algorithms had to be devised. A few very simple layout algorithms were implemented and then the first graphs generated by `cgol` were rendered. A short summary on these algorithms is given in Subsection 5.4.5. This three dimensional rendering required setting up all the necessary Java3D datastructures, which was quite a challenging task. This completed the first part of the project to basically get something onto the screen and to have a program ready for further improvements.

As a first improvement, the functionality to interactively add non-logical axioms to the lattice and see how it reduces the size of the current lattice was added. For reasons of user-friendliness, this also included the introduction of `undo` and `redo` functionality.
The next remarkable improvement was the better integration of `cgol`. Instead of calling `cgol` as an external program, the integration as a dynamic library was added. This required the use of the Java Native Interface, a way of calling existing C code from Java programs. For this to work, some of the original code of `cgol` had to be adapted to work together with the drawing program. In essence, this was the addition of a method to pass all the necessary parameters from the drawing program to `cgol`. Difficulties here were hidden in the details of getting the right compiler options and putting things in the right places to be found first by the linker and then by the executing program.
After this, an algorithm evaluation took place. An intensive literature search resulted in several papers on different force-directed algorithms. Some of them were related to a novel approach inspired by multi-dimensional scaling like that introduced in [11]. These algorithms are implemented in the Graphviz package, see Subsection 6.2 for the details. After some work trying to interface the drawing program with the graphviz package, it turned out that the layout algorithms in Graphviz did not prove as promising as they appeared to be.
After all this, the prototype still was not satisfactory and a better alternative was looked for. All the results from the previous chapters lead to an implementation in the form of a plugin to the WilmaScope program developed by Tim Dwyer and discussed in detail in Subsection 5.5. This new approach was very promising, since much of the already quite matured prototype could be adapted to work with WilmaScope. It required only minor changes in the original source code of WilmaScope and little changes to take the prototype apart and put it back together to work with WilmaScope.

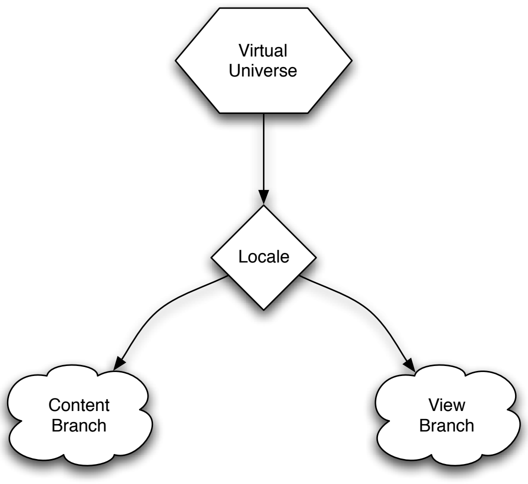


Figure 17: The General Structure of a Java3D Scenegraph.

5.4 Description of the Prototype

Since the prototype was an integral part to develop and test all the ideas, a description is given in the remainder of this subsection.

5.4.1 The Java3D Datastructure

All the data needed to render 3D objects to the screen is usually referred to as scene. This includes coordinates for the objects to be drawn, their surface materials, lighting and so on. This data can be organised in different ways. Java3D—like many other 3D programming languages—stores this information in a scenegraph. The structure of such a graph is depicted in Figure 17. This scenegraph is a collection of Java objects structured as a rooted tree. To render this scene, Java3D traverses this tree and collects all the necessary data. The root of a scenegraph is a `VirtualUniverse` object which has a `Locale` as its child.

A `Locale` can have many children but usually has only two, one subtree for all the data of the 3D objects and lights, and another one for the data used for computing the type and position of the viewer within the 3D universe.

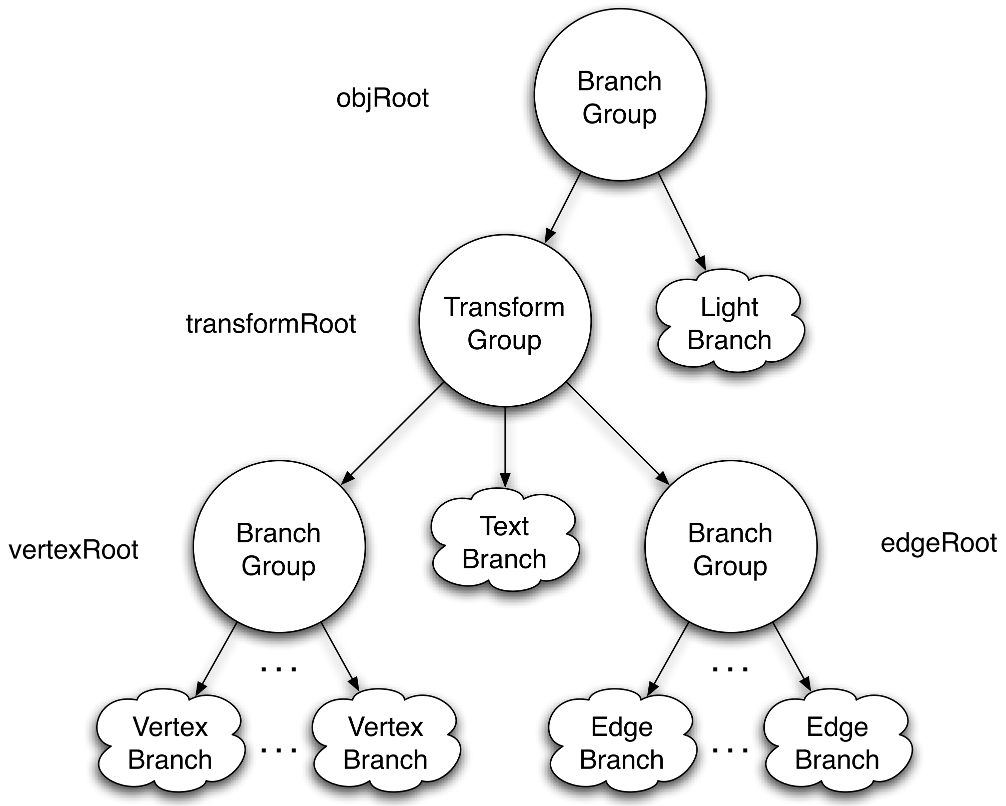


Figure 18: The General Structure of the Content Branch.

The subtree holding the data of the 3D objects is to the left, labelled as `Content Branch`, and the subtree storing the information on the view is to the right, labelled `View Branch`. They have a cloud-like shape to indicate that they do not consist of single objects, but a whole collection of objects. More details on the structure of the `Content Branch` and `View Branch` can be found in the following paragraphs. A more detailed introduction to this subject is presented in [27] and [2].

Content Branch The root of the `Content Branch` is a `BranchGroup`. This Java3D object is added as a child to the `Locale` object depicted in Figure 17 in place of the cloud-shape labelled `Content Branch`.
This `BranchGroup` is depicted at the very top of Figure 18. The name of

the variable used to refer to this `BranchGroup` is given to the left. This `BranchGroup` can store several references to other objects in the scene. One such object is its right child labelled `Light Branch`. This cloud shape is a placeholder for many objects storing information on the lights to illuminate the scene. These lights are set up at start up and not altered later on. They consist of one ambient and one directed light source. They are necessary to produce specular effects on the spheres to create a notion of depth in the 3D scene.

The left child is a `TransformGroup` which is used to rotate the whole scene. All the objects in the scene which need to be rotated are children of this object. Only the lights need not be rotated, therefore they are not children to this `TransformGroup`. The `vertexRoot` holds references to all the `Vertex Branches` in the scene. A `Vertex Branch` is a very simple structure which contains a `Sphere` object and a `TransformGroup` object to put the sphere in the right place. Similar to this structure are the `Edge Branches` to the right. They consist of a `Cylinder` object instead of a sphere to represent the edge and a `TransformGroup` to position the edge. To keep things organized, all the edges are children of the `BranchGroup` called `edgeRoot` to the right. The middle child of `transformRoot` labelled `Text Branch` is a structure similar to `vertexRoot` and `edgeRoot`. It consists of a `BranchGroup` to hold references to a collection of structures similar to the `Text Branches`. They each contain a text object representing the label of a vertex and a `TransformGroup` to position the label.

The `transformRoot` has another child which does not show up in the diagram. It is a `Behavior` object that controls the angular speed of the rotation and listens for events from the GUI to start or stop execution of this behavior.

The above separation of edges and vertices has another reason. The user wants to interact with the scene, pick vertices to draw an edge between them or switch labels on and off. The picking in Java3D is realized in a way that the scenegraph below a given object is traversed and a check is performed whether the current object was picked by the mouse click. This check is computationally expensive and thus the fewer checks have to be performed, the more responsive the user-interface stays. This is the main reason to seperate pickable vertices from non-pickable labels and edges.

View Branch The `Content Branch` holds information on the objects which populate a 3D scene. This branch of the scenegraph stores information on the view a virtual viewer has of the scene. This includes where he or she is, the direction he or she is looking in and how this view is rendered to the screen. The `View Branch` depicted as a cloud shape in Figure 17 is a very simple structure. It has a `BranchGroup` as its root which is added as a child to the `Locale` object. This `BranchGroup` has a `TransformGroup` as its child which positions the viewer and the line of sight. This `TransformGroup` has a reference to the `ViewPlatform` which stores information on the viewer and how the scene the viewer observes should be rendered to the screen.
Since the picking described above is dependent on the position of the viewer, the object controlling the picking process is added as a child of the view `TransformGroup`.

5.4.2 Communication Between `cgol` and the Drawing Program

Since the drawing program is written in Java and Java3D and `cgol` is written in ANSI C, the interaction between the two is a bit problematic. It needs more than just a method call to pass parameters and to get a return value. The first way to call `cgol` from the drawing program was to start it as an external process and wait until it has finished. The communication between the drawing program and the external process is realized via four temporary files.

- A file for the input formula, which is considered.
- A file for the non-logical axioms to reduce the size of the resulting lattice.

- A file for the lattice as the result of a failed proof search.
- A file for the saturation which contains details of the proof search.

The files involved have already been shown in Figure 5. Single arrows indicate that a file is provided by the drawing program and read by `cgol`. Double arrows indicate that the drawing program provides this file, `cgol` writes to it and after `cgol` has terminated, it is processed by the drawing program. These files all have a defined structure so they can be constructed and parsed easily. Details on the structure of these files can be found in Subsection 5.4.3.
The program `cgol` demands different input files. First and foremost, it needs a file which contains the input formula. This file has to be supplied by the user and the drawing program does not perform any computation on it. The second file which is needed is an output file to which the `cgol` program can write the lattice constituting the counter-example if the formula was not provable. This is a temporary file supplied by the drawing program. This file is later parsed to obtain the input to the drawing program. A grammar specifying the structure of this file can be found in Subsection 5.4.3. The temporary file denoted by `Saturation File` contains the structure of the proof search performed by `cgol`. It can be used to find out if a proof was found or not. If a proof was found, it contains the line "PROOF FOUND!". If this line is contained, a message has to be delivered to the user. Besides the search for proofs, `cgol` can be used with non-logical axioms to reduce the size of a lattice obtained as a result of a failed proof search. These axioms are provided in the file denoted by `Non-logical Axioms File`.
The interface to `cgol` is achieved by two classes. The `FileHandler` class handles the creation and deletion of temporary files used to communicate with `cgol` and retrieves input files and hands them to the parser or to `cgol`. The class to invoke the `cgol` program from the Java user-interface is called `CgolInterface.class`. Its method `runcgol(File infile, File outfile, File axiomFile)` is the main interface to `cgol`. It gives the files to read input from, write output to and also the file where non-logical axioms can be found—which is also provided by the `FileHandler`—to the `cgol` program.

In the first phase of the prototype, the Java UI creates a new external process to run `cgol` and waits for it to complete. Then it reads the output file in order to draw the new lattice. In the second phase as described above, `cgol` was called as a dynamic library.

5.4.3 Input and Output Files

The lattice drawing program recognizes two types of input files which are distinguished via the suffix of the files.
The input file to `cgol` which contains the input formula to obtain a lattice has the ending `.in` and the grammar specification is given in Figure 8. This structure of the input file may seem strange because of the right hand side of the first production. This less than relation is introduced due to the structure of the sequents `cgol` works on. In Definition 2.4 of [32], we find that this has the form of a *1-1-Sequent*.

> A 1-1-sequent is an ordered pair of the form $\phi \vdash \psi$, where ϕ and ψ are formulas. The formula ϕ is the only formula of the antecedent and ψ is the only formula of the succedent of the 1-1-sequent.

The left formula in the production is ϕ and the right is ψ. If the left hand side formula is omitted—as allowed by the grammar—it is assumed to be true. For reasons of practicality, $\vdash$ is replaced by $<$ in the input file. The symbol $\sim$ indicates negation, where & and | are replacements for the binary connectives $\wedge$ and $\vee$, respectively.
The last file needed for the communication between `cgol` and the drawing program is a file holding the non-logical axioms. This file usually has the ending `.nla` and has the format given in Figure 19. Two non-terminal symbols do not have productions given here. The non-terminal *Spaces* expands to one or more terminal whitespace characters, excluding newline characters. These characters are treated specially and are represented by the non-terminal *Newline*, which expands to exactly one terminal newline character.

$$\begin{array}{rcc} \textit{Axiomfile} & \rightarrow & \textit{Axiomlist} \\ \textit{Axiomlist} & \rightarrow & (\ \epsilon \mid (\textit{Axiom})+\ ) \\ \textit{Axiom} & \rightarrow & \textit{Index Space Index Newline} \\ \textit{Index} & \rightarrow & (-)?\ (\ [\mathbf{0}\text{–}\mathbf{9}]\ )+ \end{array}$$

Figure 19: Grammar for a File Containing Non-logical Axioms.

This file is needed for the lattice compression procedure implemented in `cgol`. For the following reason, the top and bottom elements **top** and **bot** of lattices do not appear in this file. Adding non-logical axioms that contain these minimal and maximal elements does not add any useful information. For all elements x of the lattice, it holds by definition that $\mathbf{bot} \leq x$ and also $x \leq \mathbf{top}$. Adding these two relations as axioms does not help the compression procedure to reduce the size of the lattice. This procedure tries to reduce the lattice by finding equivalence classes of formulas based on non-logical axioms.
The drawing program accepts another type of input file which has the ending `.lat`. This file—in contrast to those mentioned above—is not passed on to `cgol`, but handled directly by the drawing program. If an input file with the ending `.in` is selected and `cgol` cannot find a proof, then the counterexample of `cgol` is output to the drawing program in the same `.lat` format. Its grammar specification is given in Figure 7. *Formula* is specified in the grammar of the `.in` input file in Figure 8. *Space* is a non-terminal which expands to one or more terminal whitespace characters except the newline character. Although it is possible to have an input file which contains no vertices but a *Subformulalist*, it does not make much sense. In this case, no lattice is drawn and the subformulas cannot be matched to any vertices.

5.4.4 Parsing `cgol` Output

A parser is important to read the output file of `cgol`. Since the whole program is written in Java, `lex` and `yacc` are of no use. Yet, with JavaCC there is a program at hand which—for our purpose—has the same functionality. It constructs a top-down recursive descent parser from a given annotated grammar specification. This parser belongs to the class of $LL(k)$ parsers where the look-ahead k is equal to 1 for performance reasons. The specification is derived from the interface definition in [32]. The input to JavaCC is located in the file called `LatParser.jj`. After processing it with JavaCC, new files are created. The most important one of these is `LatParser.java`, which is the parser itself. Along with it, the additional files `SimpleCharStream.java`, `LatParserConstants.java`, `LatParserTokenManager.java`, `Token.java`, `ParseException.java`, and `TokenMgrError.java` are created automatically. For more detailed information on how the parser generator works, please see the URL `http://javacc.dev.java.net/`.

5.4.5 Simple Layout Algorithms

To get all the vertices layouted as quickly as possible, a very simple layout algorithm was implemented. Since all vertices were assigned a level by Algorithm 1, this level was used as the z-coordinate. The other coordinates were calculated according to the following formulas where n is equal to the number of vertices at the same level and $i = 0, ..., n-1$. All the vertices at the same level are arranged in a circle with equal angle ϕ between two neighbouring vertices.

$$\begin{aligned} x &= n \times \sin(i\phi) \\ y &= n \times \cos(i\phi) \\ \phi &= \frac{2\pi}{n} \end{aligned}$$

5.5 WilmaScope

In this subsection, we will give a short overview of the features of WilmaScope and justify the use of this program as a basis for the lattice drawing program instead of the prototype developed in earlier stages of this project. A more

detailed description of WilmaScope can be found in [19] or [4].
When creating WilmaScope, Tim Dwyer aimed to achieve the following goals.

- Create "a general purpose 3D visualization system".
- "Provide easy-to-use components which can be employed by future software across different application domains".
- Develop software "that is flexible, interactive and easily extensible".
- "Provide these benefits as widely as possible" by releasing the software under a free software licence.

All the above points make it a good framework to develop an application to display lattices. With very little changes in the original WilmaScope code, it was possible to extend the application to work together with `cgol`. A great deal of this ability to be extensible is due to the way WilmaScope is designed. It follows the model-view-controller architecture which aims to separate the underlying data model from parts of the program which manipulate the model and other parts which are needed to render the graph to the screen. A schematic diagram of this approach can be found in Figure 20. To accomodate for some WilmaScope specific needs, the `Controller` and `Model` parts are split up into smaller parts. Still the general structure and the mode of interaction as indicated by the arrows connecting the components is visible.

This architecture makes it possible to change only a small part of the code in order to make it work together with parts of the prototype which are needed to interact with `cgol`. All the code that needed adaption belongs to the `Controller` part located in the upper part of Figure 20. The code constituting the `View` and `Model` parts of the WilmaScope program could be left unchanged.
The changes made in the WilmaScope source code pertain to the `GUI` and `FileHandler` parts of the `Controller`. The `GUI` was changed such that the additional menu items to manipulate lattices was added. The `FileHandler`

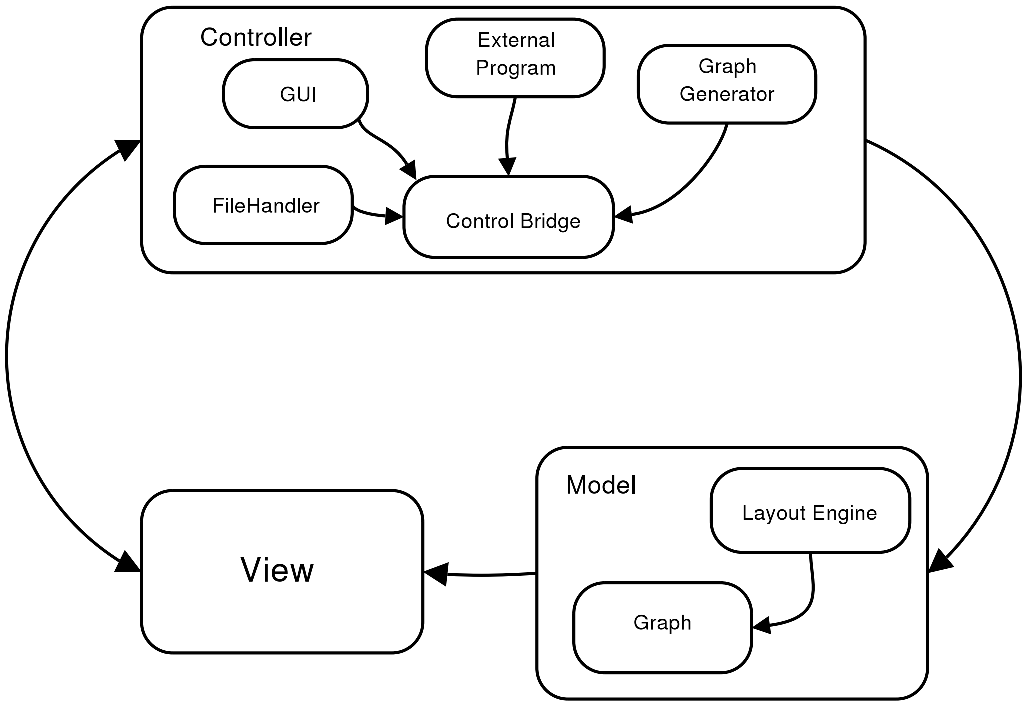


Figure 20: The WilmaScope Model-View-Controller Architecture.

part was adapted such that input files with the endings `.in` and `.lat` could be passed to and processed by `cgol`.

6 Comparison With Existing Programs

In this section, two other programs to display graphs are compared to the program described in the previous sections. This list is by no means exhaustive as there are more than these two programs available for drawing graphs. Yet, these two seemed to be the two most important programs for drawing lattices. This is the reason for evaluating them in the course of this thesis.

6.1 LatDraw

First and foremost, the program of R. Freese has to be mentioned which produces quite pleasing drawings, but it has some important drawbacks, as we will point out below. A Java Applet of this program for drawing lattices is also available[1]. In Figure 21, we see two different versions of the McCune 1 lattice produced by the `cgol` program as a counter-example to Equation (8). These pictures are obtained by different angles of rotation about the vertical axis. Figure 21(a) is a very bad example of the layout. In the center of the picture, there are many edge crossings and some vertices are hidden behind others. This makes it very hard to discern which edges belong to which vertex. Figure 21(b) is a better picture of the same lattice. Here, only one vertex is hidden by another and there are fewer edge crossings than in the above picture. Still, these pictures are not very pleasing. The applet does not support zooming in or out of the scene, consequently, the size of the lattice that can be viewed is rather limited. All what is supported is to automatically scale the lattice to fill the given area. This is problematic as indicated in Figure 21(a), since larger lattices clutter the picture and make it hard to see the structure of the lattice.
From the usability point of view, Freese's applet does not support direct interaction with cgol or interactively adding non-logical axioms.Using this program as a starting point to build in interaction with `cgol` was considered, but the poor user-interface and the fact that applets cannot use ANSI

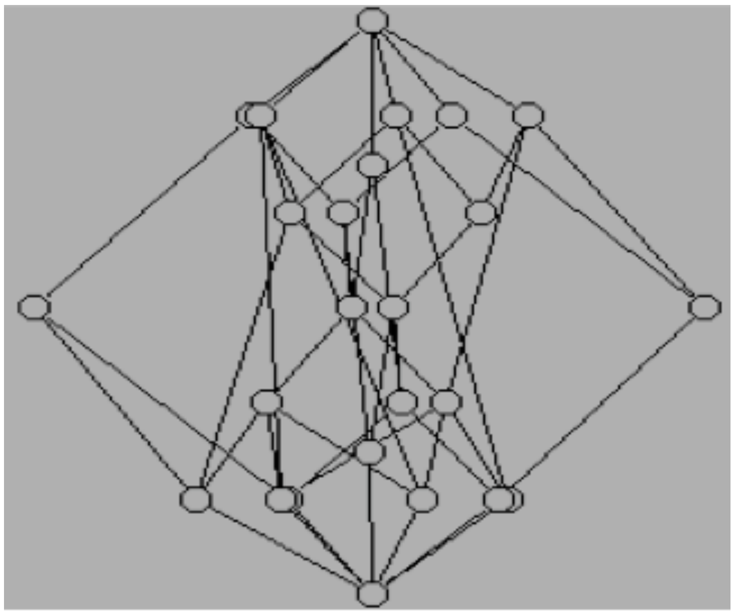
[1]Available at: `http://www.math.hawaii.edu/~ralph/LatDraw`

C libraries suggested not to implement an interface between `cgol` and the applet and look for other alternatives. As was pointed out, WilmaScope is the better choice.
The last drawback we want to mention is that the grey background unnecessarily reduces the contrast between nodes, edges and the background. The drawing is not finely grained either and thus the edges do not look very pleasing. Furthermore, it does not make use of colours.

6.2 Graphviz

The Graphviz library[2] by E. Gansner including the `dot` and `neato` layout engines seemed to be quite promising but was found to perform rather poorly on lattices in this context. The `dot` layout could not be used since it only produces two-dimensional layouts. If one looks for 2D-layout, then it is a good choice. The other layout engine `neato` produced three-dimensional drawings of lattices. Unfortunately, it is not possible to constrain vertices to certain levels. These drawings then lack the usual layering expected in Hasse Diagrams. An adaption of the algorithm did not seem practical. As a matter of user-friendliness, it was not a feasible solution to build an application relying on a seperate installation of the Graphviz library when WilmaScope provides everything that is necessary to render graphs. A screenshot of the same lattice as in Figure 21, this time rendered with `dot` from the Graphviz package can be seen in Figure 22. Note that the `dot` drawing, as opposed to the Freese ones, is aligned top to bottom instead of bottom to top. If we compare this picture to the ones produced by Freese's applet, then it does not show any symmetries or other structural properties. It only has the right layering. The graph in Figure 22 is not very large or very dense. A problem that is not readily visible here, but occurs in larger and denser graphs, is that `dot` may produce confusingly routed edges and many edge crossings.

[2]Available at: `http://www.graphviz.org`


(a) A Bad Drawing.

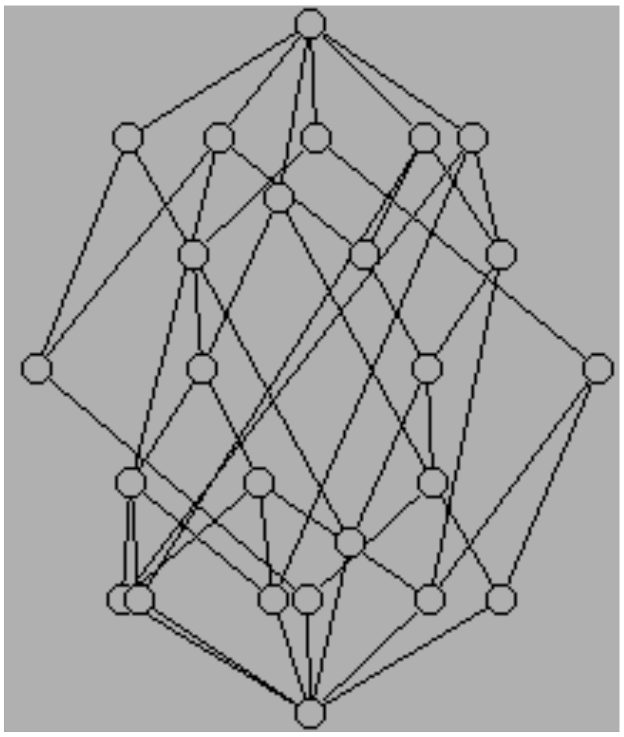

(b) A Better Drawing.

Figure 21: McCune 1 Lattice Drawn by the Freese Applet.

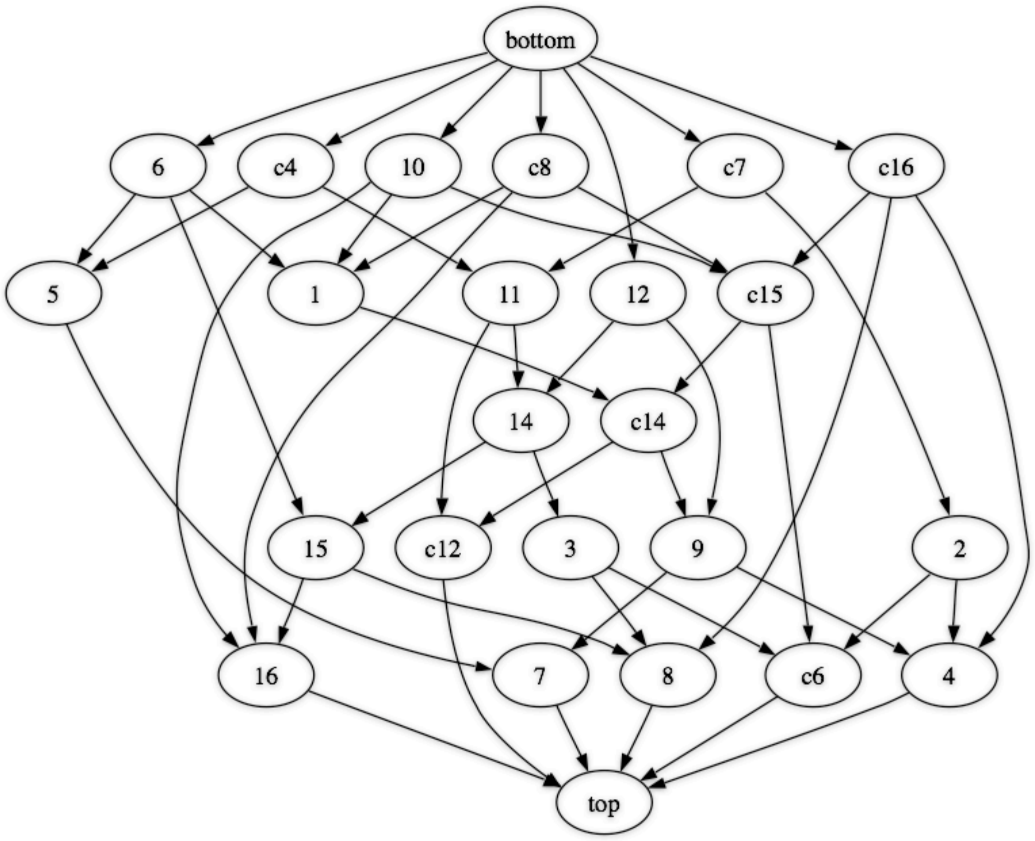


Figure 22: McCune 1 Lattice Drawn by `dot`.

7 Conclusion

In this thesis we have investigated a way to render lattices which are the result of a failed proof search in orthologic to the screen three-dimensionally. For this reason, lattice theoretic and algorithmic aspects were considered. Psychological and physiological properties of the intended users were also considered. According to insights gained from intensive study of the literature, a drawing system was implemented—first as a prototype and then as an extension to an existing program. This system was then compared to the two most important programs in the field of lattice drawing. The results obtained from this comparison are quite promising.

This system is a first attempt to create a program which works together with `cgol` and renders the results in three dimensions. It performs quite well, yet there is ample space for improvements. New layout algorithms could be added to better convey the structure of lattices. The algorithm used in LatDraw performs quite well and since it does 3D-layout, it could be integrated. Beside the layout part, also the rendering component of the program can be improved. Some parts of WilmaScope might be removed, since they are not really needed for manipulating lattices.

This thesis involved very little testing of the user-interface. Only a few persons had access to the program, actively used it and provided feedback for improvements. Future work might involve more extensive testing and improvement of the user-interface. It would also be beneficial to test this software in a virtual or augmented reality environment where an even greater degree of three-dimensionality can be achieved.

Another improvement is tighter integration of `cgol` and the drawing program. This can be achieved by replacing the way these two pieces of software communicate with each other. It would be interesting to see a Java implementation of `cgol` to compare it to the ANSI C version. This would be interesting in terms of performance. Then the whole software could be distributed as a Java Applet, which would make it more accessible and easier to use.

The goal of creating a piece of software which makes the use of `cgol` a great deal easier and convenient was achieved. How great the benefits really are has yet to be determined.

It was also pointed out in Subsection 2.6 that this drawing program may be useful to display concept lattices. This needs some further investigation and some minor adjustments of the software.

List of Figures

List of Algorithms

Bibliography

[1] L. J. Arthur. *Rapid Evolutionary Development: Requirements, Prototyping & Software Creation.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[2] J. Barrilleaux. *3D User Interfaces with Java 3D.* Manning Publications Co., Greenwich, CT, USA, 2001.

[3] D. E. Berlyne. *Aesthetics and Psychobiology.* Century Psychology Series. Meredith Corporation, New York, 1971.

[4] T. Dwyer. Extending the WilmaScope 3D graph visualisation system - software demonstration, 2005.

[5] P. Eades. A heuristic for graph drawing. In *Congressus Numerantium*, volume 42, pages 149–160, Manitoba, Winnipeg, 1984. Utilitas Mathematica Publishing.

[6] U. Egly. Counter (ortho-)lattice construction. unpublished, 2006.

[7] U. Egly and H. Tompits. On different proof-search strategies for orthologic. *Studia Logica*, 73(1):131–152(22), February 2003.

[8] M. W. Eysenck and M. T. Keane. *Cognitive Psychology - A Student's Handbook.* Psychology Press, 4^{th} edition, 2003.

[9] R. Freese, J. Ježek, and J. Nation. Free lattices. In *Mathematical Surveys and Monographs*, volume 42. American Mathematical Society, Providence, 1993.

[10] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software—Practice and Experience*, 21(11):1129–1164, 1991.

[11] E. R. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In J. Pach, editor, *Graph Drawing, New York, 2004*, pages 239–250. Springer, 2004.

[12] E. R. Gansner, E. Koutsofio, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[13] B. Ganter and R. Wille. *Formale Begriffsanalyse: Mathematische Grundlagen.* Springer, 1996.

[14] B. Ganter, R. Wille, and K. E. Wolff, editors. *Beiträge zur Begriffsanalyse.* B.I.–Wissenschaftsverlag, Mannheim, 1987.

[15] A. Goralčíková and V. Koubek. A reduct-and-closure algorithm for graphs. In J. Bečvář, editor, *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 301–307. Springer, 1979.

[16] R. N. Haber and M. Hershenson. *The Psychology of Visual Perception.* Holt, Rinehart and Winston, Inc., 1973.

[17] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, New York, NY, USA, 1995. ACM.

[18] P. W. Jordan. *Designing Pleasurable Products.* Taylor & Francis Ltd., 2002.

[19] M. Jünger and P. Mutzel, editors. *Graph Drawing Software.* Mathematics and Visualization. Springer, 2004.

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[21] K. Koffka. *Principles of Gestalt Psychology.* Harcourt Brace, 1935.

[22] W. McCune. Automatic proofs and counterexamples for some ortholattice identities. *Information Processing Letters*, 65(6):285–291, 1998.

[23] T. Munzner. H3: laying out large directed graphs in 3d hyperbolic space. In *IEEE Symposium on Information Visualization 1997*, pages 2–10. IEEE Computer Society, October 1997.

[24] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997.

[25] S. Ratcliffe, editor. *The Oxford Dictionary of Phrase, Saying and Quotation*, page 251. Oxford University Press, 2^{nd} edition, 2002.

[26] K. Risden, M. P. Czerwinski, T. Munzner, and D. Cook. An initial examination of ease of use for 2D and 3D information visualizations of web content. *International Journal of Human Computer Studies*, 53(5):695–714, Nov 2000.

[27] D. Selman. *Java 3D Programming.* Manning Publications Co., Greenwich, CT, USA, 2002.

[28] M. Stern. *Semimodular Lattices - Theory and Applications.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1999.

[29] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Systems, Man, Cybernetics*, 11(2):109–125, 1981.

[30] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Proceedings of the 8th International Symposion on*

Graph Drawing, GD, volume 1984 of *Lecture Notes in Computer Science*, pages 171–182. Springer-Verlag, September 2000.

[31] C. Ware and G. Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2):121–140, 1996.

[32] A. Zugaj. Counterlattice generation for non-provable formulas. Master's thesis, Vienna University of Technology, 2006.