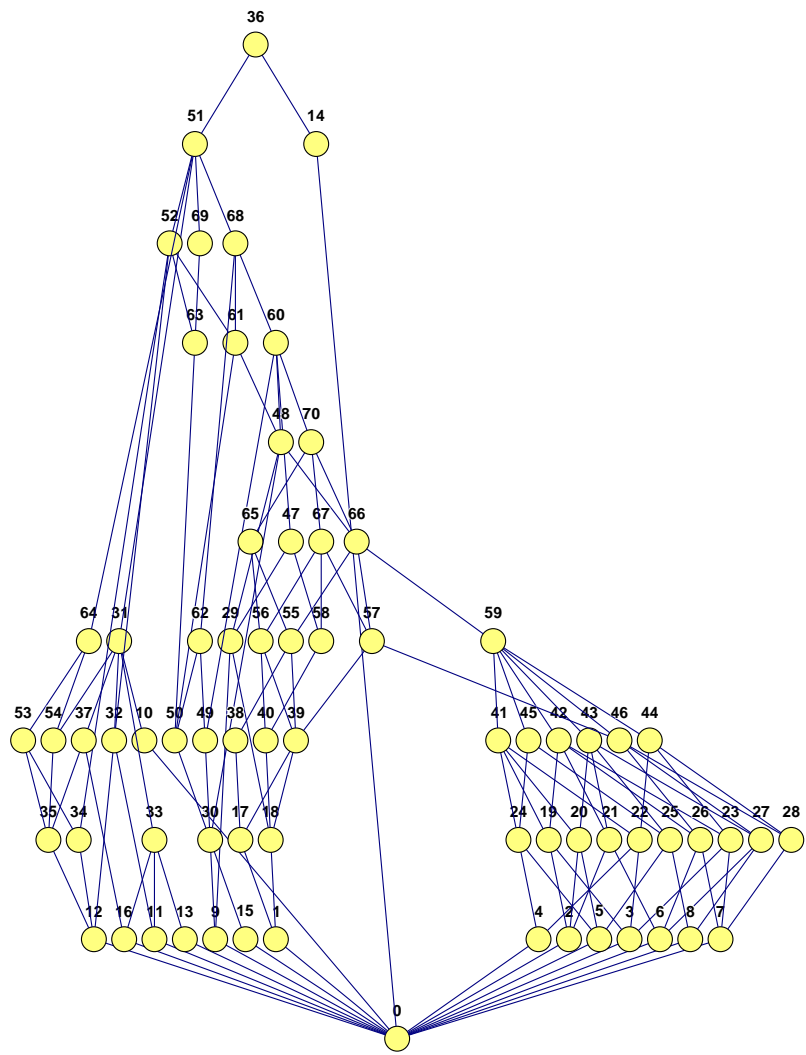


Fast Concept Analysis Algorithm:
STL/C++ Implementation

Assad Jarrahian
Gary Moore
Murugan Pandian
Marcin Zalewski

Rensselaer Polytechnic Institute
Troy, New York

April 20, 2001



Contents

1	Introduction	1
1.1	Overview	1
1.2	Intended Audience	1
1.3	Terms and Definitions	1
1.3.1	User Level	1
1.3.2	Implementation Level	2
2	Fast Concept Analysis Algorithms	3
2.1	Neighbors Algorithm	3
2.2	Lattice Algorithm	4
3	Design Issues	5
3.1	Abstraction Level	5
3.2	Implementation Level	6
4	STLFCA Example	7
4.1	Input	7
4.2	Output	8
5	graphplace	10
5.1	Input	10
5.2	Output	11
6	Source Code	12
6.1	STLFCA	12
6.1.1	main	12
6.1.2	parse	13
6.2	context.h	14
6.2.1	Constructors	15
6.2.2	Get and Set Incidence	17
6.2.3	Object Prime	18
6.2.4	Attribute Prime	19
6.2.5	Get and Set Objects	20
6.2.6	Get and Set Attributes	20
6.2.7	Object and Attribute Double Prime	21
6.2.8	Get Object and Attribute Size	21
6.2.9	Get Integer Object and Attribute Set	22
6.2.10	Private Data Members (objects, attributes, and incidence)	23
6.3	lattice.h	23
6.3.1	Constructor	23
6.3.2	InsertLookup	24
6.3.3	Insert	25
6.3.4	Next	25
6.3.5	Print graphplace Input File	26

6.3.6	Get Lattice Size	26
6.3.7	Private Data Members (count and Node)	27
6.4	neighbors.h	28
6.4.1	Concept typedef	28
6.4.2	Neighbors Algorithm	28
6.4.3	isSorted	30
6.5	latticeAlgorithm.h	30
7	Test Results	32
7.1	time vs. context size	32
7.2	time vs. lattice size	33
8	Applying STLFCAs to STL Concept Hierarchy	34
8.1	STL Concept Lattice	35
8.2	Lattice Node Concept Values	37
8.3	Objects (Types)	40
8.4	Attributes (Concepts)	44
8.5	Result	45
9	Future Work	46
A	Appendix A	47
A.1	stlfa.cc	47
A.2	context.h	47
A.3	lattice.h	49
A.4	neighbors.h	50
A.5	latticeAlgorithm.h	50
B	Appendix B	51
B.1	hirestimer.h	51
C	Appendix C	54
C.1	randomGen.h	55
C.2	randomGen.cc	55
C.3	random.cc	56
D	Bibliography	58

1 Introduction

Formal Concept Analysis, introduced in 1982 by Rudolf Wille, is a data analysis theory which identifies conceptual structures in data sets. Since its introduction, this theory has been applied to numerous fields of study; for example, computer science, ecology, musicology, linguistic databases, and civil engineering. An advantage of Formal Concept Analysis is the ability to produce graphical visualizations of concept lattices for analyzing. In some applications (i.e. program analysis) a context's resulting concept lattice—of a larger size—is difficult to analyze visually, but is still useful as part of the application's internal data structure. This document will describe and show a new Fast Concept Analysis (FCA) implementation using the Standard Template Library (STL). The new implementation was designed to produce a graphical visualization of concept lattices with the support of a graph drawing package called `graphplace`. The main objectives of this document include the following: 1. To introduce formal concept analysis, and show how Fast Concept Analysis algorithms function. 2. To compare existing implementations of Formal Concept Analysis with this new C++/STL implementation, and to show the advantages/disadvantages of this implementation. 3. Prove or disprove the correctness of SGI's STL concept hierarchy, and show its concept lattice.

1.1 Overview

1.2 Intended Audience

This document explains Formal Concept Analysis, and a STL implementation of the Fast Concept Analysis algorithms (STLFCA). It is intended for people in the computer science field with the assumption that the reader has basic knowledge of C++, STL, and discrete mathematics.

1.3 Terms and Definitions

1.3.1 User Level

The following key definitions are provided to give a user a better understanding of Formal Concept Analysis:

- **Formal Concept Analysis:** A method used mainly for data analysis, which is structured into units of formal abstractions of concepts, allowing meaningful and comprehensible interpretation.
- **Formal:** Prefix used to to emphasize that formal concepts are mathematical entities, and not associated with concepts of the mind.
- **Fast Concept Analysis:** A set of algorithms used to perform formal concept analysis in an efficient manner.

- **Formal Context (G,M,I):** A triple (G, M, I) which requires the following condition: If G and M are sets and $I \subseteq G \times M$ is a binary relation between G and M
- **Object (g):** $g \in G$
- **Objects (G):** elements of G
- **Attribute (m):** $m \in M$
- **Attributes (M):** elements of M
- **Incidence (I):** incidence of a context (G, M, I) denoted I
- **Prime ('): defined as:**
for $A \subseteq G$, $A' := \{m \in M: (g, m) \in I \forall g \in A\}$
for $B \subseteq M$, $B' := \{g \in G: (g, m) \in I \forall m \in B\}$
- **Formal Concept:** A pair (A, B) is a formal concept of (G, M, I) if and only if
$$A \subseteq G, B \subseteq M, A' = B \text{ and } A = B'$$
- **Hull:** The resulting set of applying the double prime operation to $(A \cup g)$ — $(A \cup g)''$
- **Intent (A):** intent of a concept denoted A
- **Extent (B):** extent of a concept denoted B
- **subconcept-superconcept relation:** $(A_1, B_1) \leq (A_2, B_2) \iff A_1 \subseteq A_2 (\iff B_2 \subseteq B_1)$
- **Concept Lattice:** The ordered set of all formal concepts of (G, M, I) is denoted by $\beta(G, M, I)$
- **Upper Neighbors (c^*):** A concept's associated set of all upper neighbors in a Concept Lattice
- **Lower Neighbors (x_*):** A concept's associated set of all lower neighbors in a Concept Lattice

1.3.2 Implementation Level

The following key definitions are provided to give an implementor a better understanding of the STL Fast Concept Analysis implementation:

- **Concept:** Represented by `pair<vector<int>, vector<int> >`, where the first vector contains the extent—objects' index values in the incidence matrix—and the second vector contains the intent—attributes' index values in the incidence matrix.

- **incidence:** Represented by `vector<vector<int> >`, where each row index represents a unique object (g), and each column index represents a unique attribute (m). An object (g_x) that has an attribute (m_y) sets an incidence element to one ($I_{x,y} = 1$), otherwise zero.
- **Context:** A Class which contains methods for: all set operations involved in Fast Concept Analysis and constructing, accessing, and mutating the context.
- **latticeSet:** Represented by `vector<Node>` container. `latticeSet` contains each Node structure of the lattice in a **Time Ordering**.
- **Node:** Is a node in a lattice. Node is a **struct** that contains an **index** of type `int`, a **Concept**, and a **lowerUpperNeighbors** data structure of type `pair<vector<int>, vector<int> >`. The **index** is a unique number that identifies the **Node**, which is used for referencing a concept's neighbors in the lattice and to provide the time of insertion in the `latticeSet`.
- **lowerUpperNeighbors:** Represented by `pair<vector<int>, vector<int> >`. The first element of the pair is a vector of **Node** indices, where each index is a lower neighbor of the **Node**. The second element of the pair is a vector of **Node** indices, where each index is an upper neighbor of the **Node**.

2 Fast Concept Analysis Algorithms

Fast Concept Analysis consists of two major algorithms: **Neighbors Algorithm** and **Lattice Algorithm**. A **Concept Lattice** ($\beta(G, M, I)$) is generated by applying **Lattice Algorithm** to a **Formal Context** (G, M, I). In order to create a **Concept Lattice** ($\beta(G, M, I)$) an understanding of the algorithms is needed. Since, **Neighbors Algorithm** is employed by **Lattice Algorithm** it will be explained first.

2.1 Neighbors Algorithm

The purpose of the **Neighbors Algorithm** is to compute the upper neighbors of a concept (A, B). Let min denote the minimal set of elements in $G \setminus A$, and let N denote a set of upper neighbors of the concept (A, B), initially defined as \emptyset . Then, the algorithm generates new concepts $((A \cup g)'' , (A \cup g)')$ for every $g \in G \setminus A$, and the following conditions are checked: If $(min \cap (((A \cup g)'' , (A \cup g)') \setminus A \setminus g) = \emptyset)$, then the newly generated concept is added to the upper neighbors set N ; else, the new concept does not belong to an upper neighbor, and the object g is removed from the minimal set. Initially, this algorithm assumes all elements produce neighbors.

The asymptotic complexity **Neighbors Algorithm** is $O(|G|^2 \times |M|)$: computing the hull using the $''$ -operator takes $O(|G| \times |M|)$, and is required G times when G is empty.

```

Neighbors((A, B), (G, M, I))
  Begin
    min ← G \ A
    N ← ∅
    ForEach g ∈ G \ A Do
      B1 ← (A ∪ g)'
      A1 ← (A ∪ g)''
      If ((min ∩ (A1 \ A \ {g})) = ∅) Then
        N ← N ∪ (A1, B1)
      Else
        min ← min \ g
    EndFor
  Return N
End

```

Figure 1: Neighbors Algorithm computes the upper neighbors of a concept

2.2 Lattice Algorithm

Lattice Algorithm ($Lattice(G, M, I)$) computes the **concept lattice** of a given formal context (G, M, I) . Initially, the algorithm starts with the smallest possible concept $(\emptyset'', \emptyset')$ and assigns it to a concept c , and inserts it in $\beta(G, M, I)$, which will be denoted here as a search tree L .

For every neighbor x of c in (G, M, I) : 1. If x is not an element ($lookup(x, L)$) of L , then x is inserted in L ($insert(x, L)$). 2. c is then added to the set x_* , and x is added to the set c^* . The above is done for each concept c in L , by assigning c to the next concept ($next(c, L)$) of c in L . If the next concept of c is not found then the algorithm exits, and returns the resulting **concept lattice** L .

To better understand this algorithm, a definition of a next concept ($next(x, L)$), and insert a concept ($insert(c, L)$) is required. Given a concept c , the next concept of c in L is defined as the smallest concept that is greater than c with respect to the total order \prec used inside of the tree. To ensure that all concepts inserted are also considered for their **upper neighbors**, the total tree order \prec must relate to the partial lattice order \leq by the following: $c_1 < c_2 \implies c_1 \prec c_2$. Using this order, guarantees recently inserted neighbors are greater than the actual concept, with respect to \prec , and will be considered later by $next(c, L)$.

The asymptotic worst case complexity of **Lattice Algorithm** is $O(|G| \times |M|)$, since the operations on L do not contribute to the complexity.

```

Lattice((G, M, I))
  Begin
     $c \leftarrow (\emptyset'', \emptyset')$ 
    insert( $c, L$ )
  Repeat
    ForEach  $g \in G \setminus A$  Do
       $x \leftarrow \text{lookup}(x, L)$ 
      If  $x = \text{null}$  Then
        insert( $x, L$ )
         $x_* \leftarrow x_* \cup \{c\}$ 
         $c^* \leftarrow c^* \cup \{x\}$ 
      EndFor
    next( $c, L$ )
  Until  $c = \text{null}$ 
  Return  $L$ 
End

```

Figure 2: Lattice Algorithm computes the concept lattice of (G, M, I)

3 Design Issues

This section describes issues encountered during the design of the FCA Algorithm. For further clarification, this section is divided into two subsections, the Abstraction Level and the Implementation Level. The Abstraction Level contains issues of the interface, and semantics. The Implementation Level describes the choice of internal algorithms and data structures that meet the abstraction level requirements.

3.1 Abstraction Level

The STL implementation of FCA (STLFCA) possesses an object-oriented program style. Major data structures are implemented as objects, while the main sub-algorithms are implemented as separate functions. This implementation style, compared to traditional procedural programming, allows for encapsulation of data, and increased readability. Also, using STL provides a simpler, more flexible, safer, and more efficient solution.

The objects implemented are **Context**, **Lattice**, and **Concept**. The algorithms that are implemented separately are **Neighbors**, and **LatticeAlgorithm**. This new STLFCA design mirrors the logical components of the FCA Algorithm [1]. The **Context** object provides the functionality to store a set of objects, attributes, and the incidence matrix. It also provides the ' and '' operators, which are both applied to object and attributes sets. Attributes and objects are stored as integer values, a possibility of naming them with strings is provided, but not required. The **Lattice** object stores and provides operations for the concept lattice constructed by the STLFCA algorithm. For every concept stored in the

Lattice, its lower and upper neighbors are stored. Storing the lower neighbors is not a requirement of the FCA, but including lower neighbors to the STLFCFA implementation provides additional functionality at very little cost. Presently, only upper neighbors are used when traversing the lattice; however, storing lower neighbors, as well as upper neighbors, may allow for easier traversal of a lattice in both directions (top-bottom and bottom-top). The *lookup*(x, L) and the *insert*(x, L) operations from FCA algorithm are merged together in order to increase efficiency in STLFCFA.

Concept objects do not provide any functionality, except for accessing and mutating the extent and intent sets. All the operations necessary for **Concept** are provided by other objects. Also, **Concepts** are not ordered by a **Strict Weak Ordering**, rather they are ordered by the time they are found. So, given this ordering there is no expectations about the next **Concept**. *next*(c, L) and *insert*(x, L) of the FCA algorithm are modified to conform to the ‘time ordering.’

3.2 Implementation Level

Objects and attributes are stored as sets of integers in the **Concept** object. Actually, **Concept** is only a **typedef** of a pair of two **vectors** of integers (see 1.3.2). The **Context** object stores the incidence’s sets of objects, and attributes in a two dimensional array of integer values. The array is implemented using STL’s **vectors** of **int**. The objects correspond to indices of the outer vector, and attributes correspond to indices of the inner vectors (see section 6.2.10). Originally, a two dimensional array of booleans defined the incidence matrix **incidence**, but observing the time of all functions yielded that all boolean operations took the majority of time in STLFCFA. One of the most important data structures in the implementation is the **Lattice** object. Since a lot of time is spent on looking up concepts, it is important to design an efficient implementation. STL’s **vector** container was selected for the internal storage of concepts (**LatticeSet**). **Vector** provides $O(\log(N))$ lookup, and amortized constant time insertion—using **push_back**. Using the generic algorithm **find**, only if a concept is not found, it is pushed back on the **LatticeSet**. This complies with the lattice structure, since every concept instance can only be inserted once into the lattice, and they have to be inserted in the order in which they were found.

Since the structure **LatticeSet** needs to satisfy the requirement of a time ordering of concepts, rather than a **Strict Weak Ordering**, the FCA definition of *next*(c, L), *insert*(x, L), and *lookup*(x, L) is modified for the new STLFCFA implementation. *next*(c, L) returns the concept which is immediately after the current concept c in **LatticeSet**, so every concept is considered. *insert*(x, L) inserts a concept at the end of the **LatticeSet** vector, and this is done every time a concept is found, which orders the concepts by time.

To be able to store concepts in the **LatticeSet** a **Node** structure had to be created. It allows to store a concept along with its upper and lower neighbors. While designing the **Node** class, a recursive type definition was encountered. In a

first attempt, `Node` stored two vectors of iterators to the `LatticeSet`. However, this definition required a `Node` type before its definition was finished. This resulted in unavoidable compilation problems. For that reason, a unique integer (`Node.index`) value is assigned to every `Node`. This unique assigned number is based on an `Lattice` object variable `count`, which is initially set to zero and is incremented at every `Node` insertion into the `LatticeSet`. This unique number associated with a `Node`, provides an efficient way to reference lower and upper neighboring concepts (see 1.3.2 `lowerUpperNeighbors`) for every `Node`, while keeping the `lowerUpperNeighbors` structure contained in `Node`.

4 STLFCFA Example

The following is a simple example of applying `Lattice Algorithm` to a `Formal Context`, and producing a `Concept Lattice` using STLFCFA. Given a `Formal Context` (G, M, I) , where:

$G = \{girl, boy, woman, man\}$,

$M = \{juvenile, adult, female, male\}$, and

$I = (\{1, 1, 0, 0\}, \{0, 0, 1, 1\}, \{1, 0, 1, 0\}, \{0, 1, 0, 1\})$

Table 1: Incidence matrix (I)

	girl	boy	woman	man
juvenile	1	1	0	0
adult	0	0	1	1
female	1	0	1	0
male	0	1	0	1

4.1 Input

The STLFCFA requires a specific `Formal Context` input file, with a format that is broken into the following three parts. First, every attribute name followed by a new-line is written in the file. This allows STLFCFA to know all the attributes (M) in the `Formal Context`. Then, a separator (“--”) followed by a new-line. Last, each object name is followed by a colon (“:”), white-space, all attribute names associated with the object (separated by white-space), a semi-colon (“;”), and a new-line. This allows STLFCFA to know all the objects (G) in the `Formal Context`, and how to generate the incidence matrix (I). The following is this example’s `context.in` file:

```

female
juvenile
adult
male
--
girl:  female;
girl:  juvenile;
woman: female adult;
boy:   male;
boy:   juvenile;
man:   male adult;

```

Figure 3: context.in file used in this example

4.2 Output

STLFCA produces a specific output file, with the following format. All nodes are defined by the node name in parentheses, white-space, node number, and a new-line. All edges are defined by a node number, white-space, node number, white-space, and a new-line. In this example the node name and node number are the same to easily reference the concept's extent and intent sets. The following command entered at the prompt generates the output file `lattice.g`:

```
./stlfca context.in lattice.g
```

Given larger contexts, the lattice output would be hard to analyze. `stlfca` generates two files:

- `explain.graph` (see figure 5)
- `context.list` (see figure 6)

The file `explain.graph` displays the integer representation of a node number and its corresponding concept (extent and intent sets). The integer representation is the actual `Node.index` value, which represents when the context was found (time ordering). Unfortunately, using the actual object/attribute names in the extent/intent sets in `explain.graph` can become very large and unreadable. `explain.graph` displays the concepts' extent/intent elements using the the object's/attribute's index number in the incidence matrix, respectively. `context.list` displays the objects'/attributes' index value in the incidence matrix, along with their actual string representation.

```
(0) 0 node
1 0 edge
2 0 edge
3 0 edge
4 0 edge
(1) 1 node
5 1 edge
6 1 edge
(2) 2 node
5 2 edge
7 2 edge
(3) 3 node
6 3 edge
8 3 edge
(4) 4 node
7 4 edge
8 4 edge
(5) 5 node
9 5 edge
(6) 6 node
9 6 edge
(7) 7 node
9 7 edge
(8) 8 node
9 8 edge
(9) 9 node
```

Figure 4: lattice.g file generated stlfca

```
Node 0: {}, {0, 1, 2, 3}
Node 1: {0}, {0, 1}
Node 2: {1}, {0, 2}
Node 3: {2}, {1, 3}
Node 4: {3}, {2, 3}
Node 5: {0, 1}, {0}
Node 6: {0, 2}, {1}
Node 7: {1, 3}, {2}
Node 8: {2, 3}, {3}
Node 9: {0, 1, 2, 3}, {}
```

Figure 5: explain.graph file generated by stlfca

```
Objects:
0 = girl
1 = woman
2 = boy
3 = man

Attributes:
0 = female
1 = juvenile
2 = adult
3 = male
```

Figure 6: context.list file generated by stlfca

5 graphplace

STLFCA generates a file with the extension .g with a specific file format in order to be used with `graphplace` [6]. The application `graphplace` is a general graph placement filter, with postscript features. The `man` page for `graphplace` can be viewed at:

- <http://adam.wins.uva.nl/~psf/documentation/man/cat1/graphplace.html>

This will enable a user to produce a graphical visualization of a concept lattice.

5.1 Input

The required input file for `graphplace` has the same format as the output file of STLFCA (see section 4.2).

5.2 Output

The following command entered at the prompt generates the output file `lattice.ps` (see Figure 7):

```
./graphplace -p lattice.g > lattice.ps
```

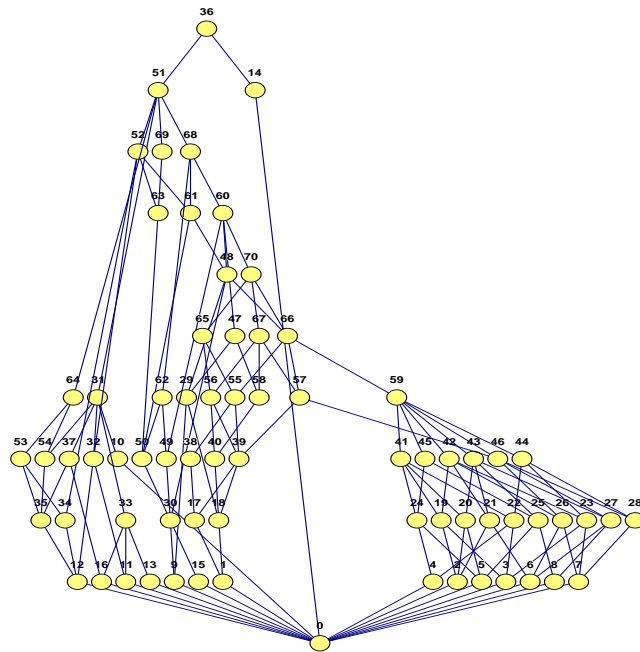


Figure 7: `lattice.ps` file generated by `graphplace` using the output in section 4.2

6 Source Code

STLFCA is written in C++ and heavily uses STL. All source code executed on a SUN sparc Ultra 10, running SunOS release 5.8, and there is no guarantee that the following source code will run on other platforms. Also, all source code compiled with g++ 2.95.2. The following sub-sections describe every file in detail. Some less important parts of the code are in Appendix A (i.e. include statement, and declarations), to just focus on the main functions and methods of STLFCA. All source code is broken into logical blocks, where each logical block is separated by a blank line

6.1 STLFCA

STLFCA (`stlfca.cc`) contains the main function that initializes and starts the lattice algorithm, and a parse function that parses the command line arguments.

"`stlfca.cc`" 12 ≡

```
    { stlfca includes and declarations 47a }
    { stlfca main 13 }
    { stlfca parse 14 }
    ◇
```

6.1.1 main

First, the necessary streams are instantiated for `stlfca`'s input files and output files. `infile` is the context input file (see section 4.1) stream, `graph1file` is the time vs. context size output file stream, `graph2file` is the time vs. lattice size output file stream, `latticefile` is the lattice output file stream (see section 4.2), `timer` is a flag that triggers timing output, and `linuxTimer` (see Appendix B) is the timing object used to calculate the time (seconds) to run `latticeAlgorithm`. `parse` then parses the command line arguments. Second, a `Context` is created with the context input file and a `Lattice` is instantiated. If the `timer` flag is set, then the `linuxTimer` starts to run (initial time of 0 seconds) to determine the amount of time it takes for `latticeAlgorithm` to generate a concept lattice. Then given a `Context` and a `Lattice`, `LatticeAlgorithm` computes the concept lattice. After the concept lattice is computed, the `timer` flag is checked. If the `timer` flag is set then: 1. The context size and the time for lattice algorithm to generate a concept lattice is written to `graph1file`. 2. The lattice size and the time for lattice algorithm to generate a concept lattice is written to `graph2file`. This collected data is used in section 7 to analyze the speed of STLFCA, and compare it to other implementations. Next, `printGraphPlace` is called to write the the lattice to file in `graphplace` format, using integer representations of nodes as node labels. Last, the `context.list` file is generated by getting the object/attribute vector from the `Context`, and for each string element of the vector the following is written to the file: index number, equal sign, string, and a new line character.

⟨stlfca main 13⟩ ≡

```
int main(int argc, char** argv)
{
    ifstream infile;
    ofstream graph1file;
    ofstream graph2file;
    ofstream latticefile;
    bool timer;
    LinuxTimer linuxTimer("ConceptAnalysis");
    parse(argc, argv, timer, infile, graph1file, graph2file, latticefile);

    Context<> c(infile);
    Lattice<Concept> l;
    if (timer)
        linuxTimer.Start();
    LatticeAlgorithm(c,l);
    if (timer)
    {
        float time = (float)linuxTimer.ElapsedTime()/linuxTimer.TicksPerSecond();
        graph1file << l.getLatticeSize() << "\t" << time << endl;
        graph2file << c.getObjSize()*c.getArgSize() << "\t" << time << endl;
    }
    l.printGraphplaceInput(latticefile, 1);

    vector<string> attributes, objects;
    c.getObjects(objects);
    c.getAttributes(attributes);
    if(objects.size() >= c.getObjSize() && attributes.size() >= c.getArgSize())
    {
        ofstream contextList("context.list");
        contextList << "Objects: " << endl;
        for(vector<string>::iterator i = objects.begin(); i < objects.end();++i)
            contextList << i - objects.begin() << " = " << *i << endl;
        contextList << "\nAttributes: " << endl;
        for(vector<string>::iterator i = attributes.begin(); i < attributes.end();++i)
            contextList << i - attributes.begin() << " = " << *i << endl;
    }

    return 0;
}
```

◇

Macro referenced in 12.

6.1.2 parse

`parse` takes `main`'s arguments, and references to `timer`, `infile`, `graph1file`, and `graph2file` as input. The condition checks if there are two arguments passed to `stlfca` (context input file and lattice output file). If there are two arguments passed, then the `timer` flag is set false—no timing analysis will be

done—and the required streams are opened for the files. Else, a condition checks if there are four arguments passed to `stlfca` (context input file, lattice output file, time vs. context file, and time vs. lattice file). If four arguments are passed, then the `timer` flag is set true—timing analysis will be done—and the required streams are opened. If neither two or four arguments are passed, or there is an error opening a stream the program will exit. `parse` returns nothing.

⟨stlfca parse 14⟩ ≡

```

void parse(int argc, char** argv, bool& timer, ifstream& infile, ofstream& graph1file,
           ofstream& graph2file, ofstream& latticefile)
{
    string message = "usage: ";
    if(argc == 3)
    {
        timer = false;
        infile.open(argv[1]);
        latticefile.open(argv[2]);
        if ((!infile)||(!latticefile))
        {
            cerr << message << endl;
            exit(1);
        }
    }
    else if(argc == 5)
    {
        timer = true;
        infile.open(argv[1]);
        latticefile.open(argv[2]);
        graph1file.open(argv[3], ios::out | ios::app);
        graph2file.open(argv[4], ios::out | ios::app);
        if ((!infile)||(!latticefile)||(!graph1file)||(!graph2file))
        {
            cerr << message << endl;
            exit(1);
        }
    }
    else
    {
        cerr << message << endl;
        exit(1);
    }
}

```

◇
Macro referenced in 12.

6.2 context.h

The `Context` class is used to contain a `Formal Context` and provide methods to perform operations on object and attribute sets. `Object` and `Attribute`

define classes used to describe objects and attributes by default strings, respectively.

```
"context.h" 15 ≡
```

```
    < context includes and declarations 47b >
    template<class Object = string, class Attribute = string>
    class Context
    < context public declaration 47c >
    < context Context Default Constructor 47d >
    < context Context Constructor2 48a >
    < context Context Constructor3 48b >
    < context Context Constructor4 48c, ... >
    < context Context Constructor5 16 >
    < context getIncidence 17 >
    < context setIncidence 18a >
    < context objPrime 18b >
    < context argPrime 19 >
    < context getObjects 20a >
    < context setObjects 20b >
    < context getAttributes 20c >
    < context setAttributes 20d >
    < context objDPrime 21a >
    < context argDPrime 21b >
    < context getObjSize 21c >
    < context getArgSize 22a >
    < context getIntObjSet 22b >
    < context getIntArgSet 22c >
    < context private fields 23a >
    < context end declarations 48e >
    ◊
```

6.2.1 Constructors

The following constructor is used by `stlfca` to initialize a context from a file stream. Instead of taking a file name, `Context` takes a reference to a file stream as input, so that the calling routine handles the file stream creation. This constructor reads in a file of a format specified in section 4.1. All other `Context` constructors of less importance are in Appendix A.

First, the file stream is checked for I/O errors, and if so then the program will display an error message and exit. Next, all attributes are read from the stream and pushed on the `attributes` set until the separator (“--”) is encountered. If the separator is not encountered, then an error message is displayed and the program exits. The outer loop `while(in >> obj)` reads in each object name, and removes the last character (“.”) from the object name. If the object name does not already exist in the objects set, it is pushed on the `objects` set and the object’s index `obj_index` is saved to reference its location in the incidence matrix. Then the incidence matrix `incidence` is resized depending on the number of attributes and objects found so far. The inner while loop

`while(in >> attr)` reads in each attribute name for the given object from the outer loop. If the last character of the attribute name read from the stream is a semi-colon (“;”) then it is removed. Next, the `attribute` set is searched for the currently read attribute name to obtain the attribute’s index number `attr_index` to reference its location in the incidence matrix. The incidence is set to one at row `obj_index` and column `attr_index`. This signifies that the object at `obj_index` has the attribute at `attr_index` in the incidence. Finally, the object size and attribute size are saved for later reference. If the end of the file is reached the method returns, otherwise an error message is displayed.

⟨context Context Constructor5 16⟩ ≡

```
Context(ifstream & in)
{
    if(in.is_open() == 0)
    {
        cerr << "Input stream is not opened" << endl;
        exit(1);
    }
    if(!in)
    {
        cerr << "Some of the error flags in input stream are set before reading" << endl;
        exit(1);
    }

    string temp;
    while((in >> temp) && (temp.compare("--") != 0))
    {
        attributes.push_back(temp);
    }
    if(!in)
    {
        cerr << "Some problem was encountered before \"--\"" << endl;
        cerr << "Check file format!!!" << endl;
        exit(1);
    }

    typename vector<Object>::iterator i;
    typename vector<Attribute>::iterator j;
    string attr, obj, temp_attr;
    int obj_index;
    int attr_index;
    while(in >> obj)
    {
        obj.erase(obj.end()-1);
        i = find(objects.begin(),objects.end(),obj);
        if(i == objects.end())
        {
            objects.push_back(obj);
```

```

        obj_index = objects.size()-1;
        incidence.resize(incidence.size()+1,vector<int>(attributes.size()));
    }
    else
    {
        obj_index = i - objects.begin();
    }
    while(in >> attr)
    {
        temp_attr = attr;
        if (*(attr.end()-1)==';' && (attr.size() > 1))
            attr.erase(attr.end()-1);
        else if (*(attr.end()-1)==';')
            break;
        j = find(attributes.begin(),attributes.end(),attr);
        if(j == attributes.end())
        {
            cerr << "Unknown attribute " << attr << endl;
            exit(1);
        }
        attr_index = j - attributes.begin();
        incidence[obj_index][attr_index] = true;
        if (temp_attr != attr)
            break;
    }
}

argsize = incidence[0].size();
objsize = incidence.size();
if(in.eof())
    return;
else
{
    cerr << "Unexpected File Error" << endl;
}
}

```

◇

Macro referenced in 15.

6.2.2 Get and Set Incidence

The following two methods get and set a value in the incidence matrix. `getIncidence` take a constant object index value (`obj`) and a constant attribute index value (`arg`) as input. The method returns the value at row `obj` and column `arg` in the `incidence`.

⟨context `getIncidence 17`⟩ ≡

```

bool getIncidence(const int obj, const int arg) const
{

```

```

    return incidence[obj][arg];
}

```

◇

Macro referenced in 15.

`setIncidence` take a constant object index value (`obj`), a constant attribute index value (`arg`), and a constant boolean value `value`—to enter in the incidence—as input. The method returns the `value` set at row `obj` and column `arg` in the `incidence`.

⟨context `setIncidence` 18a⟩ ≡

```

bool setIncidence(const int obj, const int arg, const bool value)
{
    incidence[obj][arg] = value;
    return value;
}

```

◇

Macro referenced in 15.

6.2.3 Object Prime

The following method applies prime (see 1.3.1) to an object set. `objPrime` takes a constant reference to a vector of integers `objset`, which the prime operation is going to be applied to. Also it takes a reference to a vector of integers `output`, which is the result of the prime operation. First, If the object set `objset` is empty, then `output` is set to the `Context`'s attribute set and the method returns. Otherwise, if `objset` is not empty then `output` is cleared and the following operations occur. For each attribute `k` in the incidence, if all objects in `objset` have the attribute `k`, then `k` is added to the resulting set `output`.

⟨context `objPrime` 18b⟩ ≡

```

void objPrime(const vector<int>& objset, vector<int>& output) const
{
    if(objset.empty())
    {
        output.resize(argsize);
        for(int k = 0; k < argsize; ++k)
            output[k] = k;
        return;
    }

    output.clear();
    int temp = true;
    for(int k = 0; k < argsize; ++k)
    {
        temp = true;

```

```

        for(typename vector<int>::const_iterator j = objset.begin();
            j < objset.end(); ++j)
            temp = incidence[*j][k] && temp;
        if(temp)
            output.push_back(k);
    }
}

```

◇

Macro referenced in 15.

6.2.4 Attribute Prime

The following method applies prime (see 1.3.1) to an attribute set. `argPrime` takes a constant reference to a vector of integers `argset`, which the prime operation is going to be applied to. Also it takes a reference to a vector of integers `output`, which is the result of the prime operation. First, If the attribute set `argset` is empty, then `output` is set to the `Context`'s object set and the method returns. Otherwise, if `argset` is not empty then `output` is cleared and the following operations occur. For each object `k` in the incidence, if all objects in `objset` have the attribute `k`, then `k` is added to the resulting set `output`.

⟨context argPrime 19⟩ ≡

```

void argPrime(const vector<int>& argset, vector<int>& output) const
{
    if(argset.empty())
    {
        output.resize(incidence.size());
        for(int k = 0; k < incidence.size(); ++k)
            output[k] = k;
        return;
    }

    output.clear();
    int temp = true;
    for(int k = 0; k < incidence.size(); ++k)
    {
        temp = true;
        for(typename vector<int>::const_iterator j = argset.begin();
            j < argset.end(); ++j)
            temp = incidence[k][*j] && temp;
        if(temp)
            output.push_back(k);
    }
}

```

◇

Macro referenced in 15.

6.2.5 Get and Set Objects

The following methods get and set the `object` set of strings. `getObjects` takes a reference to a vector of objects (`obj`) as input, and sets it to the `Context`'s object set. This method returns nothing.

⟨context getObjects 20a⟩ ≡

```
void getObjects(vector<Object> &obj) const
{
    obj = objects;
}
```

◇

Macro referenced in 15.

`setObjects` takes a reference to a vector of objects as input, and assigns `Context`'s object set to it. Then it assigns the `Context`'s data member `objsize` to the new size of the object set. This method returns nothing.

⟨context setObjects 20b⟩ ≡

```
void setObjects(const vector<Object> &obj)
{
    objects = obj;
    objsize = objects.size();
}
```

◇

Macro referenced in 15.

6.2.6 Get and Set Attributes

The following methods get and set the `attribute` set of strings. `getAttributes` takes a reference to a vector of attributes (`arg`) as input, and assigns it to the `Context`'s attribute set. This method returns nothing.

⟨context getAttributes 20c⟩ ≡

```
void getAttributes(vector<Attribute> &arg) const
{
    arg = attributes;
}
```

◇

Macro referenced in 15.

`setAttributes` takes a reference to a vector of attributes as input, and assigns `Context`'s attribute set to it. Then it assigns the `Context`'s data member `argsize` to the new size of the attribute set. This method returns nothing.

⟨context setAttributes 20d⟩ ≡

```

void setAttributes(const vector<Attribute> &arg)
{
    attributes = arg;
    argsize = attributes.size();
}

```

◇

Macro referenced in 15.

6.2.7 Object and Attribute Double Prime

The following two methods apply the double prime set operation to return the hull. `objDPrime` takes a constant reference to an object set (`objset`), and a reference to the hull (`output`). The hull is calculated by calling `objPrime` and then `argPrime` to the `objPrime` result. This method returns nothing.

⟨context objDPrime 21a⟩ ≡

```

void objDPrime(const vector<int>& objset, vector<int>& output) const
{
    objPrime(objset, output);
    vector<int> temp = output;
    argPrime((const vector<int>&) temp, output);
}

```

◇

Macro referenced in 15.

`argDPrime` takes a constant reference to an attribute set (`argset`), and a reference to the hull (`output`). The hull is calculated by calling `argPrime` and then `objPrime` to the `argPrime` result. This method returns nothing.

⟨context argDPrime 21b⟩ ≡

```

void argDPrime(const vector<int>& argset, vector<int>& output) const
{
    argPrime(objset, output);
    vector<int> temp = output;
    objPrime(temp, output);
}

```

◇

Macro referenced in 15.

6.2.8 Get Object and Attribute Size

The following methods get the size of the `Context`'s object/attribute set size, respectively. `getObjSize` has no input, and returns the size of the `Context`'s object set `objects`.

⟨context getObjSize 21c⟩ ≡

```

int getObjSize() const
{
    return objsize;
}

```

◇

Macro referenced in 15.

`getArgSize` has no input, and returns the size of the `Context`'s attribute set attributes.

⟨ context `getArgSize 22a` ⟩ ≡

```

int getArgSize() const
{
    return argsize;
}

```

◇

Macro referenced in 15.

6.2.9 Get Integer Object and Attribute Set

The following two methods are used to get the object/attribute vectors of integers, where the integers represent the index number of the object/attribute in the incidence matrix, respectively. This allows for a numerical representation of extent/intent sets to provide a more readable solution (if desired). Both methods return nothing.

⟨ context `getIntObjSet 22b` ⟩ ≡

```

void getIntObjSet(vector<int>& output) const
{
    output.clear();
    for(int k = 0; k < objsize; ++k)
        output.push_back(k);
}

```

◇

Macro referenced in 15.

⟨ context `getIntArgSet 22c` ⟩ ≡

```

void getIntArgSet(vector<int>& output) const
{
    output.clear();
    for(int k = 0; k < argsize; ++k)
        output.push_back(k);
}

```

◇

Macro referenced in 15.

6.2.10 Private Data Members (objects, attributes, and incidence)

`Context`'s private data members include: `incidence`, `objects`, `attributes`, `objsize`, and `argsize`. The `incidence` data member is the incidence matrix of the `Formal Context`, and it is declared as a vector of vector of integers. `incidence`'s row/column index values map directly to the `objects/attributes` index values, respectively. `objects` is defined as a vector of `Object`, where each element is a string that represent the object's name. `attributes` is defined as a vector of `Attribute`, where each element is a string that represent the attribute's name. `objsize/argsize` is declared as an integer, and is used to store the size of `objects/attributes` vector, respectively.

`<context private fields 23a> ≡`

```
private:
vector<vector<int> > incidence;
vector<Object> objects;
vector<Attribute> attributes;
int objsize;
int argsize;
```

◇

Macro referenced in 15.

6.3 lattice.h

`"lattice.h" 23b ≡`

```
<lattice includes and declarations 49a>
<lattice ostream operator 49b>
```

```
template<typename N>
class Lattice
```

```
<lattice public declaration 49c>
<lattice Default Constructor 24a>
<lattice InsertLookup 24b>
<lattice Insert 25a>
<lattice Next 25b>
<lattice printGraphplaceInput 26a>
<lattice getLatticeSize 26b>
<lattice private fields 27>
<lattice end declarations 50a>
```

◇

6.3.1 Constructor

`lattice`'s only constructor is the default constructor. The constructor just sets `count` and `last_position_cached` to zero, and `was_insert_called` flag to false.

⟨lattice Default Constructor 24a⟩ ≡

```
Lattice() : count(0), last_position_cached(0), was_insert_called(false) {}
```

◇

Macro referenced in 23b.

6.3.2 InsertLookup

The following method is a combination of both `insert(x, L)` and `lookup(x, L)` from the FCA Lattice Algorithm. `insertLookup` takes a reference to a concept `x`, and `c` as input. Concept `x` is an upper neighbor of `c` found by `Neighbors`, and `c` is the concept in which `x` was found to be its upper neighbor. First, `latticeSet` is searched for the upper neighbor of `c`, and if it is not found then it is inserted in the lattice (same fashion as `insert`). Next, the lattice is checked if it contains concept `c`, and if it does then the method returns false due to some unexpected case. This condition is checked if `insert` was called, and if the last position of a concept cached by `next` is not equal to `c`. Then, the lattice is searched for concept `c` in order to add its upper neighbor `x`. Finally, concept `x/c` adds concept `c/x` as its lower/upper neighbor, respectively. This method returns true if concept `x` was inserted in the lattice and `c` was added to its upper neighbors.

⟨lattice InsertLookup 24b⟩ ≡

```
bool insertLookup(const N& x, const N& c)
{
    typename vector<Node>::iterator x_pos = find(latticeSet.begin(),
        latticeSet.end(), Node(x, count));
    if(x_pos == latticeSet.end())
    {
        latticeSet.push_back(Node(x, count++));
        x_pos = latticeSet.end() - 1;
    }

    typename vector<Node>::iterator c_pos;
    if(was_insert_called && !(latticeSet[last_position_cached].c.first == c.first))
    {
        c_pos = find(latticeSet.begin(), latticeSet.end(), Node(c, count));
        if(c_pos == latticeSet.end())
        {
            return false;
        }
    }
    else
        c_pos = latticeSet.begin() + last_position_cached;
    x_pos->lowerUpperNeighbors.first.push_back(c_pos->index);
    c_pos->lowerUpperNeighbors.second.push_back(x_pos->index);
    return true;
}
```

◇

Macro referenced in 23b.

6.3.3 Insert

This method inserts a concept in the concept lattice. `insert` takes a reference to a concept as input. A `Node` is constructed with the input concept and the value of `count` (after it is incremented). The `Node` is pushed back in the `latticeSet`. `was_insert_called` is set to true—only if this function is called—and true is returned to signify that the concept was inserted.

⟨lattice Insert 25a⟩ ≡

```
bool insert(const N& c)
{
    latticeSet.push_back(Node(c, count++));
    was_insert_called = true;
    return true;
}
```

◇

Macro referenced in 23b.

6.3.4 Next

`Next` takes a reference to a concept (`c`) and the resulting next concept (`output`) as input. First, `latticeSet` is searched for concept `c`. If the end of `latticeSet` is reached without finding `c` or the next concept does not exist then false is returned to signify there is no next concept; otherwise, the next concept is assigned to the reference parameter `output`, the position is cached, and true is returned to signify the next concept was found.

⟨lattice Next 25b⟩ ≡

```
bool next(const N& c, N& output)
{
    typename vector<Node>::iterator pos = find(latticeSet.begin(),
        latticeSet.end(), Node(c, 0));
    if(pos == latticeSet.end() || ++pos == latticeSet.end())
        return false;
    output = pos->c;
    last_position_cached = pos - latticeSet.begin();
    return true;
}
```

◇

Macro referenced in 23b.

6.3.5 Print graphplace Input File

The following method is used to generate the required input file for `graphplace`. `printGraphplaceInput` takes a reference to an output file stream `out`, and a flag (with a default value of zero). First an output file stream `tempout` is instantiated with a file called “explain.graph” (see section 4.2). If the flag passed is zero, then each `Node`’s concept of the `latticeSet` is printed to the lattice output file `out`; else if the flag passed equals one, then each `Node`’s unique index number of the `latticeSet` is printed to the lattice output file `out`. During the creation of the lattice output file, “explain.graph” is created by writing each `Node` index and its corresponding concept to `tempout`. This method returns nothing.

⟨lattice printGraphplaceInput 26a⟩ ≡

```
void printGraphplaceInput(ofstream& out, int flag = 0)
{
    ofstream tempout("explain.graph");
    for(typename vector<Node>::iterator i = latticeSet.begin();
        i!=latticeSet.end();++i)
    {
        tempout << "Node " << i->index << ": " << i->c << endl;
        out << "(";
        if(flag == 0)
        {
            out << i->c;
        }
        else if(flag == 1)
        {
            out << i->index;
        }
        out << ")" << i->index << " node" << endl;
        for(typename vector<int>::iterator j =
            i->lowerUpperNeighbors.second.begin();
            j < i->lowerUpperNeighbors.second.end();++j)
            out << *j << " " << i->index << " edge" << endl;
    }
}
```

◇

Macro referenced in 23b.

6.3.6 Get Lattice Size

`getLatticeSize` has no input, and returns the size of the lattice (`latticeSet`).

⟨lattice getLatticeSize 26b⟩ ≡

```
unsigned int getLatticeSize()
{
    return latticeSet.size();
}
```

```
}
```

◇

Macro referenced in 23b.

6.3.7 Private Data Members (count and Node)

The following are all the private data members of the `lattice` class. The integer `count`, is used for assigning a `Nodes`'s index number in the lattice. `count` is incremented at every `Node` insertion in the `latticeSet` to ensure every `Node` is associated with a unique number and to represent the time of insertion. The `Node` structure is used to contain a concept, its unique identifier, and its lower and upper neighbors in the lattice. The `Node` constructor has a concept and an index number as input, and it binds these its own data members (`c` and `index`). `Node` also defines the `==` operator to test the equality of a `Node`'s concept with another. The `lowerUpperNeighbors` is defined as a pair of vectors, where the first/second element of the pair is the lower/upper neighboring `Node`'s index number, respectively. `latticeSet` is used to store all the `Nodes` of the concept lattice. The integer `last_position_cached` represents a the position of a `Node` in the `latticeSet` that was found by the last call to `next`. The integer `was_insert_called` is flag which is set true if `insert` is called. This flag is checked in `insertLookup` to make sure no segmentation faults occurs by not calling `insert` before calling `insertLookup`.

⟨lattice private fields 27⟩ ≡

```
private:
int count;

struct Node
{
Node() {};
Node(const N& con, int i) : c(con), index(i) {};

int index;
N c;

bool operator==(const Node& n) const
{
return c.first == n.c.first;
}

pair<vector<int>, vector<int> > lowerUpperNeighbors;
};

int last_position_cached;
bool was_insert_called;
vector<Node> latticeSet;
```

◇

Macro referenced in 23b.

6.4 neighbors.h

"neighbors.h" 28a ≡

```
⟨ neighbors includes 50b ⟩
⟨ neighbors typedef 28b ⟩
⟨ neighbors declarations 50c ⟩
⟨ neighbors NeighborAlgorithm Declarations 28c ⟩
⟨ neighbors NeighborAlgorithm ForEachLoop 29 ⟩
⟨ neighbors isSorted 30 ⟩
⟨ neighbors end declarations 50d ⟩
◇
```

6.4.1 Concept typedef

A **Concept** is type defined here has a pair of integer vectors. The first/second vector represents the extent/intent, respectively. The extent/intent integer values represent the objects'/attributes' index value in the incidence matrix, respectively.

⟨ neighbors typedef 28b ⟩ ≡

```
typedef pair<vector<int>, vector<int> > Concept;
◇
```

Macro referenced in 28a.

6.4.2 Neighbors Algorithm

The following function is very similar to the FCA **Neighbors Algorithm**; **Neighbors** finds upper neighbors output of a concept **C** in a context **GMI**. **Neighbors** takes a reference to a **Concept**, a constant **Context**, and a vector of **Concepts** (neighbors set) as input.

First, the neighbors output is cleared to make sure the neighbors set is empty. Then the extent **A1** and the intent **B1** are defined as a vector of integers. Next the set of all objects **objectSet** is declared and retrieved from the **Context**. **temp**, and **anotherTemp** are declared as integer vectors, and are used in **Neighbors** to emulate the FCA version. Then, **min** is declared as a vector of integers, and is assigned as the set difference of the object set **objectSet** and the extent **C.first**. Last, **counterset** is declared and set to **min** for the for loop of the algorithm. The set **counterset** is needed in order to preserve iterator validity for the for loop, since loop relies on the initial value of the **min** set.

⟨ neighbors NeighborAlgorithm Declarations 28c ⟩ ≡

```
int Neighbors(Concept& C, const Context<>& GMI, vector<Concept>& output)
{
    output.clear();
```

```

vector<int> B1, A1;

vector<int> objectSet;
GMI.getIntObjSet(objectSet);
if(!is_sorted(C.first))
    sort(C.first.begin(), C.first.end());

vector<int> temp, anotherTemp;

vector<int> min;
set_difference(objectSet.begin(), objectSet.end(), C.first.begin(),
    C.first.end(), insert_iterator<vector<int> >(min, min.begin()));

vector<int> counterset;
counterset = min;

```

◇

Macro referenced in 28a.

First, `temp` is used to extract the extent set from `C`. An element of the `counterset` is unioned with the `temp` set, the resulting set `B1` is primed. This is done to emulate the set operation $M1 \leftarrow (A \cup \{g\})'$ in FCA. Next, `B1` is primed to obtain the set `A1` to emulate the set operation $G1 \leftarrow (A \cup \{g\})''$ (double prime). Then `temp` is cleared, so it can be used for the following operation. After the extent and intent sets are calculated then the neighbors are added to `output`—given certain conditions. Next, the set difference of the extent `A1`, objects `C`, is found by using the generic function `set_difference`. Then the resulting set difference `temp` is searched for the current object `*k` of the loop and erased from `temp`. This emulates the set operation $(A1 \setminus A \setminus \{g\})$. The result of `anotherTemp` is calculated by intersecting `temp` with `min`. Finally, if `anotherTemp` is equal to the empty set, then the concept of the calculated extent `A1` and intent `B1` sets are unioned with the neighbors set `output`; otherwise the current object `*k` of the loop is removed from the `min` set. `neighbors` returns the size of the neighbors set `output` as an integer.

⟨neighbors NeighborAlgorithm ForEachLoop 29⟩ ≡

```

for(vector<int>::iterator k = counterset.begin(); k < counterset.end(); ++k)
{
    temp = C.first;
    temp.push_back(*k);
    sort(temp.begin(), temp.end());
    GMI.objPrime(temp, B1);

    GMI.argPrime(B1, A1);
    temp.clear();

    set_difference(A1.begin(), A1.end(), C.first.begin(), C.first.end(),
        insert_iterator<vector<int> >(temp, temp.begin()));
    vector<int>::iterator i = find(temp.begin(), temp.end(), *k);

```

```

    if(i != temp.end())
        temp.erase(i);

    anotherTemp.clear();
    set_intersection(min.begin(), min.end(), temp.begin(), temp.end(),
        insert_iterator<vector<int> >(anotherTemp, anotherTemp.begin()));

    if(anotherTemp.empty())
    {
        if(find(output.begin(), output.end(), Concept(A1, B1)) == output.end())
            output.push_back(Concept(A1, B1));
    }
    else
    {
        min.erase(find(min.begin(), min.end(), *k));
    }
}
return output.size();
}

```

◇

Macro referenced in 28a.

6.4.3 isSorted

The following function is used to check if the extent and intent sets of a concept are sorted in ascending order. `is_sorted` takes a reference to a vector of integers, and checks if it is sorted in ascending order. The function returns true if the vector is sorted in ascending order, otherwise false.

⟨neighbors isSorted 30⟩ ≡

```

bool is_sorted(const vector<int>& v)
{
    if (v.size() > 1)
    {
        for(vector<int>::const_iterator i=v.begin()+1;i<v.end();++i)
        {
            if(*i<*(i-1))
                return false;
        }
    }
    return true;
}

```

◇

Macro referenced in 28a.

6.5 latticeAlgorithm.h

`latticeAlgorithm` computes the Lattice L of a `Context`. First an empty set `temp` is declared and a concept `c`. The empty set is double primed for the

extent of the concept c , and it is primed for the intent set of the concept c . This is done to emulate the assignment $c \leftarrow (\emptyset'', \emptyset')$ in the FCA algorithm. The concept c is then inserted in the the lattice, and an empty neighbor set `neighborSet` is declared. Then the following occurs given that there is a `next` concept c in the lattice L : 1. `Neighbors` finds all the upper neighbors of c . 2. For each upper neighbor `insertLookup` is called to insert the neighbor x —if it is not in the lattice—and to add x to the upper neighbor set of c , and to add c to the lower neighbor set of x .

"latticeAlgorithm.h" 31 ≡

```

<latticeAlgorithm includes 50e>

void LatticeAlgorithm(const Context<>& GMI, Lattice<Concept>& L)
{
    vector<int> temp;
    Concept c;

    GMI.objDPrime((const vector<int>&) temp, c.first);
    GMI.objPrime((const vector<int>&) temp, c.second);
    L.insert(c);

    vector<Concept> neighborSet;
    do
    {
        Neighbors(c, GMI, neighborSet);
        for(vector<Concept>::const_iterator x = neighborSet.begin();
            x < neighborSet.end(); ++x)
        {
            L.insertLookup(*x, c);
        }
    }
    while(L.next(c, c));
}
<latticeAlgorithm end declarations 51a>
◇

```

7 Test Results

All timed results of STLFCAs `LatticeAlgorithm`, used `hirestimer.h` (see Appendix B) to calculate the elapsed time in seconds. The tests were ran on a SUN sparc Ultra 10, running SunOS release 5.8. Also, all source code compiled used g++ 2.95.2 with a level two optimization (-O2). The experiment used 2039 randomly generate context files, generated by `random` (see Appendix C). Their corresponding incidence matrices were between 1×1 and 81×81 elements in size. The probability of filling an element in an incidence matrix of a size greater than 700 is a random value between 0.0 and 0.2. The probability of filling an element in an incidence matrix of a size less than 700 is a random value between 0.0 and 1.0 (where it is ten times more likely to be below 0.35).

7.1 time vs. context size

The following figure generated by `gnuplot` displays the time in seconds vs. context size for `LatticeAlgorithm` to run. A quadratic curve fit was applied. The result of this test yields that context size has no major influence on the time to compute a concept lattice.

"graph2.dat" □

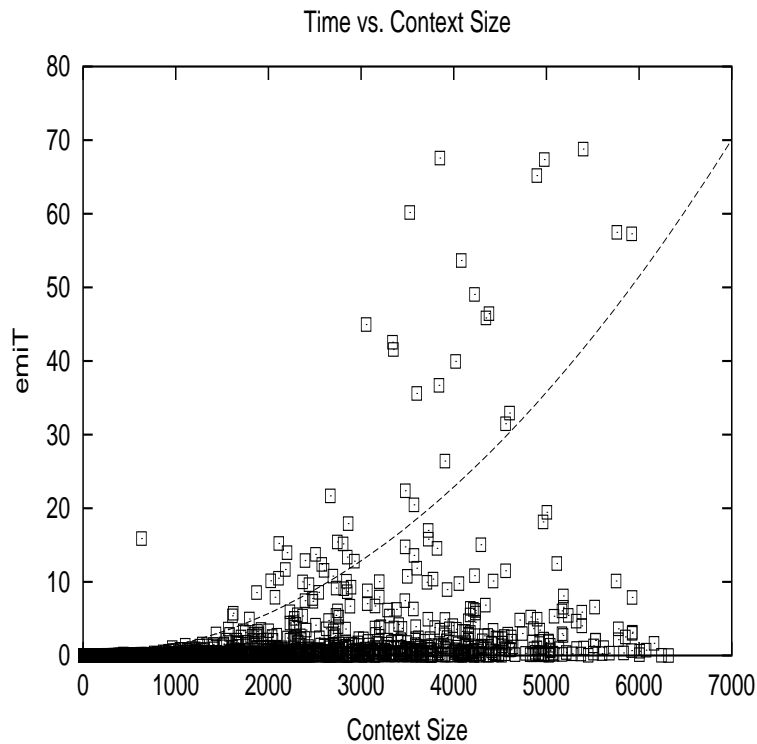


Figure 8: graph2.ps file generated by gnuplot

7.2 time vs. lattice size

The following figure generated by `gnuplot` displays the time in seconds vs. lattice size for `LatticeAlgorithm` to run. A quadratic curve fit was applied. No major conclusions could be drawn from comparing `STLFCA` to the original `FCA` [1], due to the tests performed were done on different platforms. Though it appears that the original `FCA` implementation is somewhat faster. The majority of time running `LatticeAlgorithm` is spent on `objPrime` and `argPrime`. `LatticeAlgorithm` does not grow linearly, rather it grows quadratically—the larger the lattice, the longer it takes to search the concept lattice.

"graph1.dat" □

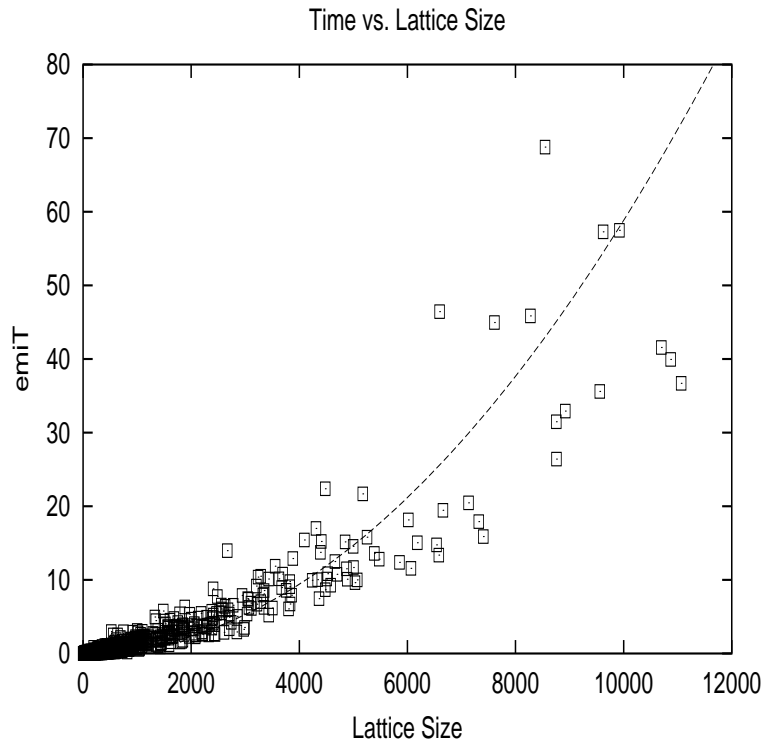


Figure 9: graph1.ps file generated by gnuplot

8 Applying STLFCA to STL Concept Hierarchy

Concept analysis has many application, and just not in computer science. The following STLFCA experiment is used to check the correctness of the STL concept and type hierarchy. All data collected is from SGI's STL site <http://www.sgi.com/Technology/STL/>. All **Concepts** were collected as attributes for the incidence matrix, and all **Types** were collected as objects of the incidence matrix. Every **Type** was checked for its "Model of" and "Refinement of" **Concepts**. Also, to help guarantee that there were no errors, every **Concept** was checked for its "Models". Figure 10 display the concept lattice generated by `stlfca`. Table 2 through Table 4 shows each **Node's** concept, where the numbers in the extent/intent represent the object/attribute index values in the incidence matrix, respectively. Table 5 through Table 8 shows the index value and its corresponding string value for every object (**Type**). Table 9 through Table 10 shows the index value and its corresponding string value for every attribute (**Concept**).

8.1 STL Concept Lattice

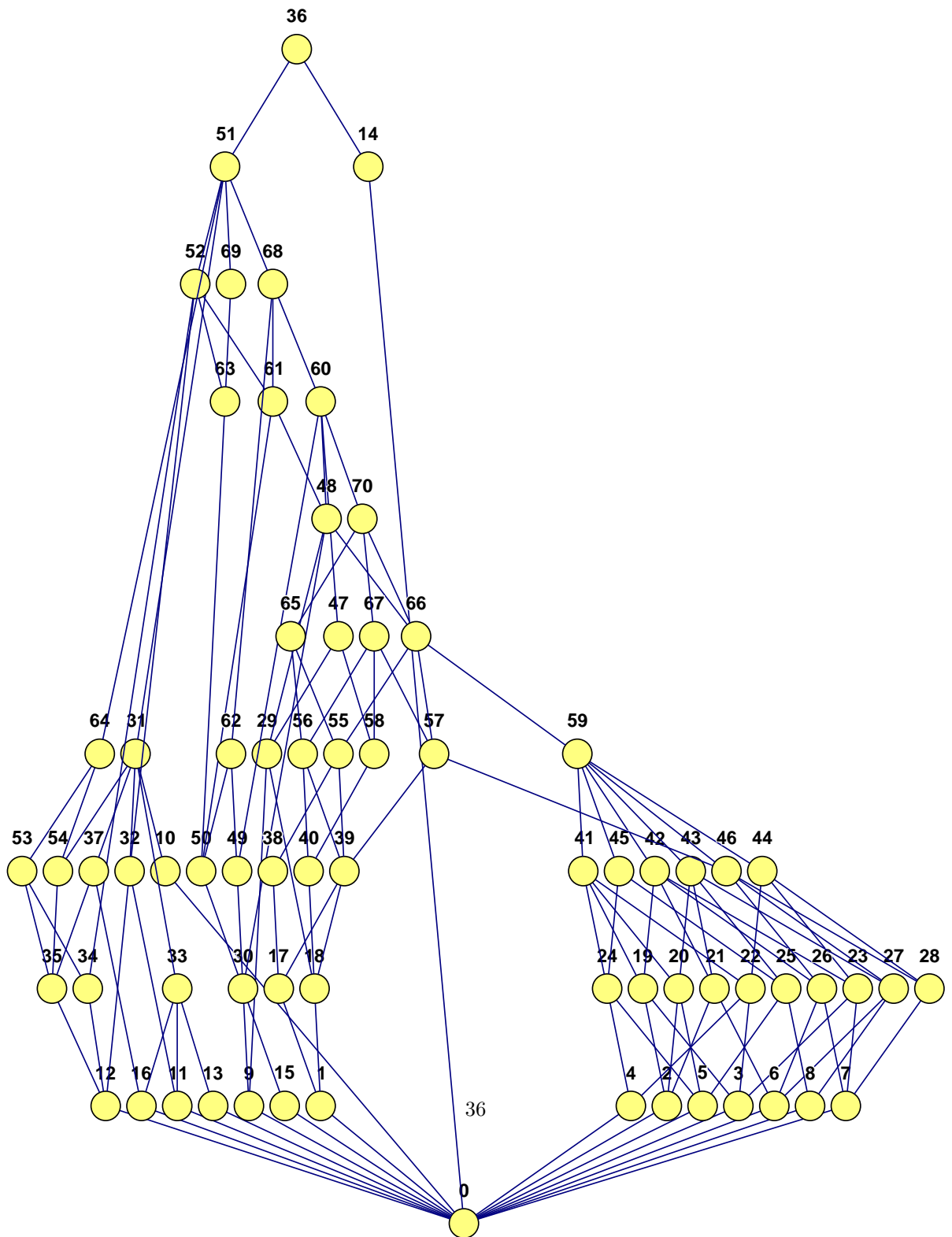


Figure 10: STL Concept Lattice: stlattice.pa file generated by graphless

8.2 Lattice Node Concept Values

Table 2: Lattice Node Concept Values

Node	Concept ($\{\text{extent set}\}, \{\text{intent set}\}$)
0	$\{\}, \{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20, 21,22,23,24,25,26,27,28,29,30, 31,32,33,34,35,36,37,38,39,40,41,42,43,44\}$
1	$\{5\}, \{2,3,4,5,12,13,14,40,42,43,44\}$
2	$\{7\}, \{0,3,4,7,11,17,18,40,42,43,44\}$
3	$\{8\}, \{0,3,4,7,8,9,11,40,42,43,44\}$
4	$\{9\}, \{0,3,4,7,8,9,15,40,42,43,44\}$
5	$\{10\}, \{0,3,4,7,15,17,18,40,42,43,44\}$
6	$\{12\}, \{0,3,4,11,13,16,17,19,40,42,43,44\}$
7	$\{13\}, \{0,3,4,8,10,11,13,16,40,42,43,44\}$
8	$\{18\}, \{0,3,4,13,15,16,17,19,40,42,43,44\}$
9	$\{39\}, \{12,20,21,22,23,25,40,42,43,44\}$
10	$\{6,50\}, \{6,39,40\}$
11	$\{64\}, \{30,39,40,42\}$
12	$\{48,49,52,53,54,55,56,65\}, \{1,27,31,36,39,40,42\}$
13	$\{73\}, \{30,37,39,40\}$
14	$\{77\}, \{41\}$
15	$\{40,95\}, \{20,21,22,23,24,25,40,42,43,44\}$
16	$\{76,99\}, \{29,30,36,39,40\}$
17	$\{5,11\}, \{2,3,4,5,13,14,40,42,43,44\}$
18	$\{2,5,21\}, \{2,3,4,12,13,14,40,42,43,44\}$
19	$\{7,8\}, \{0,3,4,7,11,40,42,43,44\}$
20	$\{7,10\}, \{0,3,4,7,17,18,40,42,43,44\}$
21	$\{7,12\}, \{0,3,4,11, 17,40,42,43,44\}$
22	$\{8,9\}, \{0,3,4,7,8, 9,40,42,43,44\}$
23	$\{8,13\}, \{0,3,4,8,11,40,42,43,44\}$
24	$\{9,10\}, \{0,3,4,7,15,40,42,43,44\}$
25	$\{10,18\}, \{0,3,4,15,17,40,42,43,44\}$
26	$\{12,13\}, \{0,3,4,11,13,16,40,42,43,44\}$
27	$\{12,18\}, \{0,3,4,13,16,17,19,40,42,43,44\}$
28	$\{13,14\}, \{0,3,4,8,10,13,16,40,42,43,44\}$

Table 3: Lattice Node Concept Values

Node	Concept ($\{\text{extent set}\}, \{\text{intent set}\}$)
29	$\{2,5,21,39\}, \{12,40,42,43,44\}$
30	$\{39,40,95\}, \{20,21,22,23,25,40,42,43,44\}$
31	$\{6,42,44,45,46,48,49,50,51,52,53,54,55,56,59,60,64,65,68,71, 72,73,74,76,79,80,81,82,87,88, 90,98,99\}, \{39,40\}$
32	$\{48,49,52,53,54,55,56,64,65\}, \{39,40,42\}$
33	$\{42,45,46,51,64,68,71,72,73,74,76,79,81,82,87,88,90,98,99\}, \{30,39,40\}$
34	$\{47,48,49,52,53,54,55,56,61,62,63,65,66\}, \{1,31,40,42\}$
35	$\{44,48,49,52,53,54,55,56,65,80\}, \{1,27,31,36,39,40\}$
36	$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22, 23,24,25,26,27,28,29,30,31,32, 33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52, 53,54,55,56,57,58,59,60,61,62, 63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82, 83,84,85,86,87,88,89,90,91,92,93, 94,95,96,97,98,99,100,101\}, \{\}$
37	$\{44,48,49,52,53,54,55,56,65,76,80,99\}, \{36,39,40\}$
38	$\{5,11,19\}, \{3,4,5,14,40,42,43,44\}$
39	$\{2,5,11,21\}, \{2,3,4,13,14,40,42,43,44\}$
40	$\{1,2,5,21\}, \{3,4,12,13,14,40,43,44\}$
41	$\{7,8,9,10\}, \{0,3,4,7,40,42,43,44\}$
42	$\{7,8,12,13\}, \{0,3,4,11,40,42,43,44\}$
43	$\{7,10,12,18\}, \{0,3,4,17,40,42,43,44\}$
44	$\{8,9,13,14\}, \{0,3,4,8,40,42,43,44\}$
45	$\{9,10,18\}, \{0,3,4,15,40,42,43,44\}$
46	$\{12,13,14,18\}, \{0,3,4,13,16,40,42,43,44\}$
47	$\{1,2,5,17,21,39\}, \{12,40,43,44\}$
48	$\{2,5,7,8,9,10,11,12,13,14,18,19,21,39,40,95\}, \{40,42,43,44\}$
49	$\{31,39,40,95\}, \{22,25,40,43,44\}$
50	$\{39,40,94,95\}, \{20,21,22,23,25,40,42,43\}$
51	$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22, 23,24,25,26,27,28,29,30,31,32, 34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53, 54,55,56,57,58,59,60,61,62,63, 64,65,66,67,68,69,70,71,72,73,74,75,76,78,79,80,81,82,83,84,85, 86,87, 88,89,90,91,92,93,94,95, 96,97,98,99,100\}, \{40\}$

Table 4: Lattice Node Concept Values

Node	Concept ($\{\text{extent set}\}, \{\text{intent set}\}$)
52	$\{0,2,3,5,7,8,9,10,11,12,13,14,15,16,18,19,20,21,22,27,30,32,38, 39,40,41,43,47,48,49,52,53,54,55, 56,61,62,63,64,65,66,75,83,84,91,92,93,94,95,96,97,100\}, \{40,42\}$
53	$\{44,47,48,49,52,53,54,55,56,57,58,61,62,63,65,66,67,69,70,80, 85,86,89\}, \{1,31,40\}$
54	$\{44,48,49,52,53,54,55,56,59,60,65,80\}, \{31,39,40\}$
55	$\{2,5,11,19,21\}, \{3,4,14,40,42,43,44\}$
56	$\{1,2,5,11,21\}, \{3,4,13,14,40,43,44\}$
57	$\{2,5,11,12,13,14,18,21\}, \{3,4,13,40,42,43,44\}$
58	$\{1,2,5,17,21\}, \{3,4,12,13,40,43,44\}$
59	$\{7,8,9,10,12,13,14,18\}, \{0,3,4,40,42,43,44\}$
60	$\{1,2,5,7,8,9,10,11,12,13,14,17,18,19,21,31,39,40,95\}, \{40,43,44\}$
61	$\{2,3,5,7,8,9,10,11,12,13,14,18,19,21,39,40,94,95\}, \{40,42,43\}$
62	$\{31,39,40,94,95\}, \{22,25,40,43\}$
63	$\{0,22,27,30,38,39,40,41,83,84,93,94,95,96,100\}, \{23,40,42\}$
64	$\{44,47,48,49,52,53,54,55,56,57,58,59,60,61,62,63,65,66,67,69,70,80,85,86,89\}, \{31,40\}$
65	$\{1,2,5,11,19, 21\}, \{3,4,14,40,43,44\}$
66	$\{2,5,7,8,9,10,11,12,13,14,18,19,21\}, \{3,4,40,42,43,44\}$
67	$\{1,2,5,11,12,13,14,17,18,21\}, \{3,4,13,40,43,44\}$
68	$\{1,2,3,5,7,8,9,10,11,12,13,14,17,18,19,21,31,39,40,94,95\}, \{40,43\}$
69	$\{0,22,27,30,34,38,39,40,41,83,84,93,94,95,96,100\}, \{23,40\}$
70	$\{1,2,5,7,8,9,10,11,12,13,14,17,18,19,21\}, \{3,4,40,43,44\}$

8.3 Objects (Types)

Table 5: Object Values

index	Object
0	back_insert_iterator<BackInsertionSequence>
1	basic_string<charT, traits, Alloc>
2	bit_vector
3	bitset <N>
4	char_producer
5	deque<T,Alloc>
6	hash <T>
7	hash_map<Key, Data, HashFcn, EqualKey, Alloc>
8	hash_multimap<Key, Data, HashFcn, EqualKey, Alloc>
9	hash_multiset<Key, HashFcn, EqualKey, Alloc>
10	hash_set<Key, HashFcn, EqualKey, Alloc>
11	list<T, Alloc>
12	map<Key,Data,Compare,Alloc>
13	multimap<Key,Data,Compare,Alloc>
14	multiset<Key,Compare,Alloc>
15	priority_queue<T,Sequence,Compare>
16	queue<T, Sequence>
17	rope<T,Alloc>
18	set<Key,Compare,Alloc>
19	slist<T,Alloc>
20	stack<T,Sequence>
21	vector<T,Alloc>
22	back insert iterator<BackInsertionSequence>
23	bidirectional iterator<T,Distance>
24	bidirectional iterator tag
25	forward iterator<T,Distance>
26	forward iterator tag
27	front insert iterator<FrontInsertionSequence>
28	input iterator<T,Distance>

Table 6: Object Values

index	Object
29	input_iterator_tag
30	insert_iterator<Container>
31	istream_iterator<T,Distance>
32	iterator_traits<Iterator>
33	ostream_iterator<T>
34	output_iterator
35	output_iterator_tag
36	random_access_iterator<T,Distance>
37	random_access_iterator_tag
38	raw_storage_iterator<ForwardIterator,T>
39	reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
40	reverse_iterator<RandomAccessIterator,T,Reference,Distance>
41	sequence_buffer<Container,buf sz>
42	binary_compose<AdaptableBinaryFunction,AdaptableUnaryFunction1, AdaptableUnaryFunction2>
43	binary_function<Arg1, Arg2, Result>
44	binary_negate<AdaptableBinaryPredicate>
45	binder1st<AdaptableBinaryFunction>
46	binder2nd<AdaptableBinaryFunction>
47	divides<T>
48	greater<T>
49	greater_equal<T>
50	hash<T>
51	identity<T>
52	less<T>
53	less_equal<T>
54	logical_and<T>
55	logical_not<T>
56	logical_or<T>

Table 7: Object Values

index	Object
57	mem_fun1_ref_t<Result, X, Arg>
58	mem_fun1_t<Result, X, Arg>
59	mem_fun_ref_t<Result, X>
60	mem_fun_t<Result, X>
61	minus<T>
62	modulus<T>
63	multiplies<T>
64	negate<T>
65	not_equal_to<T>
66	plus<T>
67	pointer_to_binary_function<Arg1, Arg2, Result>
68	pointer_to_unary_function<Arg, Result>
69	project1st<Arg1, Arg2>
70	project2nd<Arg1, Arg2>
71	select1st<Pair>
72	select2nd<Pair>
73	subtractive_rng
74	unary_compose<AdaptableUnaryFunction1, AdaptableUnaryFunction2>
75	unary_function<Arg, Result>
76	unary_negate<AdaptablePredicate>
77	char_traits
78	pair<T1, T2>
79	binary_compose<AdaptableBinaryFunction, AdaptableUnaryFunction1, AdaptableUnaryFunction2>
80	binary_negate<AdaptableBinaryPredicate>
81	binder1st<AdaptableBinaryFunction>
82	binder2nd<AdaptableBinaryFunction>
83	front_insert_iterator<FrontInsertionSequence>
84	insert_iterator<Container>

Table 8: Object Values

index	Object
85	mem_fun1_ref_t<Result, X, Arg>
86	mem_fun1_t<Result, X, Arg>
87	mem_fun_ref_t<Result, X>
88	mem_fun_t<Result, X>
89	pointer_to_binary_function<Arg1, Arg2, Result>
90	pointer_to_unary_function<Arg, Result>
91	priority_queue<T, Sequence, Compare>
92	queue<T, Sequence>
93	raw_storage_iterator<ForwardIterator, T>
94	reverse_bidirectional_iterator<BidirectionalIterator, T, Reference, Distance>
95	reverse_iterator<RandomAccessIterator, T, Reference, Distance>
96	sequence_buffer<Container, buf_sz>
97	stack<T, Sequence>
98	unary_compose<AdaptableUnaryFunction1, AdaptableUnaryFunction2>
99	unary_negate<AdaptablePredicate>
100	raw_storage_iterator<ForwardIterator, T>
101	temporary_buffer<ForwardIterator, T>

8.4 Attributes (Concepts)

Table 9: Attribute Values

index	Attribute
0	Associative_Container
1	Adaptable_Binary_Function
2	Back_Insertion_Sequence
3	Container
4	Forward_Container
5	Front_Insertion_Sequence
6	Hash_Function
7	Hashed_Associative_Container
8	Multiple_Associative_Container
9	Multiple_Hashed_Associative_Container
10	Multiple_Sorted_Associative_Container
11	Pair_Associative_Container
12	Random_Access_Container
13	Reversible_Container
14	Sequence
15	Simple_Associative_Container
16	Sorted_Associative_Container
17	Unique_Associative_Container
18	Unique_Hashed_Associative_Container
19	Unique_Sorted_Associative_Container
20	Bidirectional_Iterator
21	Forward_Iterator
22	Input_Iterator
23	Output_Iterator
24	Random_Access_Iterator
25	Trivial_Iterator
26	Adaptable_Binary_Function
27	Adaptable_Binary_Predicate
28	Adaptable_Generator

Table 10: Attribute Values

index	Attribute
29	Adaptable_Predicate
30	Adaptable_Unary_Function
31	Binary_Function
32	Binary_Predicate
33	Generator
34	Hash_Function
35	MonoidOperation
36	Predicate
37	Random_Number_Generator
38	Strict_Weak_Ordering
39	Unary_Function
40	Assignable
41	Character_Traits
42	Default_Constructible
43	Equality_Comparable
44	LessThan_Comparable

8.5 Result

The concept lattice displayed in section 8.1 is difficult to interpret, so a few key nodes in the lattice will be described. Due to the complexity of the lattice, the analysis resulted in only one discovered error. So it is left up to the reader to analyze the STL concept hierarchy in detail.

The best way to name a node is to subtract the upper nodes (intent) from the node (intent) of interest; for example, the node of interest is 59, so subtract node 66's intent from 59's intent:

$$Node59 - Node66 = \{0, 3, 4, 40, 42, 43, 44\} - \{3, 4, 40, 42, 43, 44\} = \{0\}$$

Looking this value up ($\{0\}$) in the Attribute Table (section 8.4) yields that the node is an **Associative Container**. So all objects in the extent of node 59, and all nodes to the root node 0 (where there is a path from 0 to 59), models an **Associative Container**.

Nodes on the second level from the top of the lattice do not need to be subtracted from their upper nodes (intent), since they only contain one element in their intent. Consider the two nodes 51 and 14 that appear at the top of the lattice. Node 51 are types that are all **Assignable**, and since all—but one—types model **Assignable**, that is why it is located near the top of the lattice. Node 14 is the concept **Character_Traits**, since the type `char_traits` is the only one that models the concept **Character_Traits**. The only type that is not **Assignable** is the objects in the extent of node 14, which is `char_traits`.

An interesting cluster is nodes 52, 69, 68, 64, and 31, which are all lower neighbors of node 51 (**Associative Container**) are easy to calculate. Using the above method to name a node, the following are the names of the nodes in the cluster: node 52 = **Default Constructible**, node 69 = **Output Iterator**, node 68 = **Equality Comparable**, node 31 = **Unary Function**, and node 64 = **Binary Function**.

nodes 31 and 33 are **Unary Functions** and **Adaptable Unary Functions**, respectively. This seems to be correct since 31 is the upper neighbor of 33, and **Adaptable Unary Functions** is a refinement of **Unary Functions**. Nodes 64 and 53 are **Binary Functions** and **Adaptable Binary Functions**, respectively. This seems to be correct since 64 is the upper neighbor of 53, and **Adaptable Binary Functions** is a refinement of **Binary Functions**.

All the subtractions of intent sets done so far yield a result, since each one had one upper node. An interesting node is 54 that has two upper nodes (31 and 64) **Unary Functions** and **Binary Functions**. Subtracting node 64 and 61 from 54 yields the empty set, so node 54 is a concept that is **Unary Function**, **Binary Function**, and **Assignable** only. This is due to the fact that the primitive type `bool (*)(int)` was not inserted in the list of objects. Node 54 contains an object type of `binary negate`, which is a model of **Binary Adaptable Predicate**. **Binary Adaptable Predicate**, is not contained in the intent of this node. **Binary Adaptable Predicate** is a refinement of **Predicate** and **Adaptable Binary Function**, where **Predicate** is a model of **Unary Function** and **Adaptable Binary Function** is a model of **Binary Function**. So node 54 should be labeled as **Predicate**, but since no `bool (*)(int)` was defined as an object type this error occurred.

9 Future Work

The STL implementation of Fast Concept Analysis was designed to be efficient, portable, and readable to a larger subset of the computer science community, and produce a graphical output for smaller contexts. At the present time, `graphplace` is used to produce the graphical lattices produced by the STL implementation of FCA; however, one of the drawbacks is the resulting lattices can only be viewed in 2-D. This presents a less readable solution to analyze, given larger contexts. One goal of future work is to provide an option to produce a three dimensional view of a lattice for larger contexts, and/or display the resulting lattice in another file format. Even though the STL implementation is somewhat similar in performance as earlier implementations, a primary goal is to put forth a highly efficient and scalable solution. Experimentation with other libraries (i.e. Matrix Template Library) will be applied and compared to the STL implementation.

A Appendix A

A.1 stlfca.cc

⟨stlfca includes and declarations 47a⟩ ≡

```
#include "context.h"
#include <iostream>
#include <fstream>
#include <iterator>
#include "lattice.h"
#include "latticeAlgorithm.h"
#include "neighbors.h"
#define __LINUX__
#include "hirestimer.h"
```

```
void parse(int argc, char** argv, bool& timer, ifstream& infile, ofstream& graph1file,
           ofstream& graph2file, ofstream& latticefile);
```

◇

Macro referenced in 12.

A.2 context.h

⟨context includes and declarations 47b⟩ ≡

```
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string>
#include <algorithm>
```

```
#ifndef CONTEXT_CONCEPT_ANALYSIS
#define CONTEXT_CONCEPT_ANALYSIS
```

◇

Macro referenced in 15.

⟨context public declaration 47c⟩ ≡

```
{
    public:
```

◇

Macro referenced in 15.

⟨context Context Default Constructor 47d⟩ ≡

```
Context() {}
```

◇

Macro referenced in 15.

```
<context Context Constructor2 48a> ≡
```

```
Context(const int obj, const int arg) : objsize(obj), argsize(arg),  
incidence(vector<vector<int> >(obj, vector<int>(arg))) {}
```

◇

Macro referenced in 15.

```
<context Context Constructor3 48b> ≡
```

```
Context(const vector<vector<int> >& inc) : objsize(inc.size()), incidence(inc)  
{  
if(objsize == 0)  
argsize = 0;  
else  
argsize = incidence[0].size();  
}
```

◇

Macro referenced in 15.

```
<context Context Constructor4 48c> ≡
```

```
Context(const int obj, const int arg, const vector<Object>& objlist,  
const vector<Attribute>& arglist) : objsize(obj), argsize(arg),  
incidence(vector<vector<int> >(obj, vector<int>(arg))), objects(objlist),  
attributes(arglist) {}
```

◇

Macro defined by 48cd.

Macro referenced in 15.

```
<context Context Constructor4 48d> ≡
```

```
Context(const vector<Object>& objlist, const vector<Attribute>& arglist,  
const vector<vector<int> >& inc) : objsize(inc.size()), incidence(inc),  
objects(objlist), attributes(arglist)  
{  
if(objsize == 0)  
argsize = 0;  
else  
argsize = incidence[0].size();  
}
```

◇

Macro defined by 48cd.

Macro referenced in 15.

```
<context end declarations 48e> ≡
```

```
};
```

```
#endif
```

◇

Macro referenced in 15.

A.3 lattice.h

⟨lattice includes and declarations 49a⟩ ≡

```
#include "neighbors.h"
#include "context.h"
#include <stdlib.h>
#include <vector>
#include <set>
#include <utility>
#include <functional>
#include <fstream>
#include <string>
#include <stringstream>

#ifdef LATTICE_CONCEPT_ANALYSIS
#define LATTICE_CONCEPT_ANALYSIS
```

◇

Macro referenced in 23b.

⟨lattice ostream operator 49b⟩ ≡

```
ostream& operator<< (ostream& out, const Concept& c)
{
    out << "{";
    if (c.first.size()!=0)
    {
        for(vector<int>::const_iterator j = c.first.begin();
            j < c.first.end() - 1; ++j)
            out << *j << ", ";
        out << *(c.first.end() - 1);
    }
    out << "}, {";
    if (c.second.size()!=0)
    {
        for(vector<int>::const_iterator j = c.second.begin();
            j < c.second.end() - 1; ++j)
            out << *j << ", ";
        out << *(c.second.end() - 1);
    }
    out << "}";

    return out;
}
```

◇

Macro referenced in 23b.

⟨lattice public declaration 49c⟩ ≡

```
{
    public:
    ◇
```

Macro referenced in 23b.

⟨lattice end declarations 50a⟩ ≡

```
};
```

```
#endif
```

```
◇
```

Macro referenced in 23b.

A.4 neighbors.h

⟨neighbors includes 50b⟩ ≡

```
#include <utility>
#include <algorithm>
#include <iterator>
#include "context.h"
```

```
#ifndef NEIGHBORS_CONCEPT_ANALYSIS
#define NEIGHBORS_CONCEPT_ANALYSIS
```

```
◇
```

Macro referenced in 28a.

⟨neighbors declarations 50c⟩ ≡

```
int Neighbors(Concept&, const Context<>&, vector<Concept>&) ;
bool is_sorted(const vector<int>&);
```

```
◇
```

Macro referenced in 28a.

⟨neighbors end declarations 50d⟩ ≡

```
#endif
```

```
◇
```

Macro referenced in 28a.

A.5 latticeAlgorithm.h

⟨latticeAlgorithm includes 50e⟩ ≡

```
#include "lattice.h"
#include "neighbors.h"
#include "context.h"
#include <algorithm>
```

```

#include <functional>
#include <string>

#ifdef LATTICE_ALGORITHM_CONCEPT_ANALYSIS
#define LATTICE_ALGORITHM_CONCEPT_ANALYSIS
◇
Macro referenced in 31.

<latticeAlgorithm end declarations 51a> ≡

#endif
◇
Macro referenced in 31.

```

B Appendix B

The following is the timing package used to calculate the time to run `LatticeAlgorithm`.

B.1 hirestimer.h

"hirestimer.h" 51b ≡

```

class HighResTimer {
protected:
    unsigned int startTime;
    char id[30];

public:
    HighResTimer(const char* id);
    ~HighResTimer() {}

    void Start();
    unsigned int ElapsedTime();
    virtual unsigned int GetTime() = 0;
    virtual unsigned int TicksPerSecond() = 0;
};

HighResTimer::HighResTimer(const char *id)
{
    startTime = 0;
    strncpy(this->id, id, 30);
    this->id[29] = 0;
}

void HighResTimer::Start()
{
    startTime = GetTime();
}

```

```

unsigned int HighResTimer::ElapsedTime()
{
    return GetTime() - startTime;
}

#ifdef __WINDOWS__

#include <windows.h>
#include <mmsystem.h>

class WindowsTimer : public HighResTimer {
public:
    WindowsTimer(const char* id) : HighResTimer(id) {}
    ~WindowsTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

unsigned int WindowsTimer::GetTime()
{
    LARGE_INTEGER curtime;
    QueryPerformanceCounter(&curtime);
    return curtime.LowPart;
}

unsigned int WindowsTimer::TicksPerSecond()
{
    return 1193000;
}

#endif

#ifdef __LINUX__

#include <sys/time.h>
#include <sys/times.h>
#include <sys/types.h>
#include <unistd.h>

class LinuxTimer : public HighResTimer {
public:
    LinuxTimer(const char* id) : HighResTimer(id) {}
    ~LinuxTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

```

```

unsigned int LinuxTimer::GetTime()
{
    struct timeval curtime;
    gettimeofday(&curtime, NULL);
    return (curtime.tv_sec) * 1000000 + curtime.tv_usec;
}

unsigned int LinuxTimer::TicksPerSecond()
{
    return 1000000;
}

#endif

#ifdef __BEOS__

#include <OS.h>

class BeOSTimer : public HighResTimer {
public:
    BeOSTimer(const char* id) : HighResTimer(id) {}
    ~BeOSTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

unsigned int BeOSTimer::GetTime()
{
    return system_time();
}

unsigned int HighResTimer::TicksPerSecond()
{
    return 1000000;
}

#endif

#ifdef __MACOS__

#include <QuickTimeComponents.h>

class MacTimer : public HighResTimer {
public:
    static ComponentInstance clockComponent;
    static int instance;
};

```

```

public:
    MacTimer(const char* id);
    ~MacTimer();

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

int MacTimer::instance = 0;

MacTimer::MacTimer(const char* id) : HighResTimer(id)
{
    if (instance == 0) {
        clockComponent=
            OpenDefaultComponent('clock', 'micro');
        if (clockComponent) assert(0);
    }

    ++instance;
}

MacTimer::~MacTimer()
{
    --instance;
    if (instance == 0) CloseComponent(clockComponent);
}

unsigned int MacTimer::GetTime()
{
    TimeRecord timeRec;
    ClockGetTime(clockComponent, &timeRec);
    return timeRec.value.lo;
}

unsigned int MacTimer::TicksPerSecond()
{
    return 1000000;
}

#endif
◇

```

C Appendix C

The following is the source code for a random **Formal Context** file generator. An example of a Formal Context file is shown in Figure 3. Given the desired number of objects and attributes in the context, and the probability that an object will have one of the attributes, **random** will generate a formal context file.

random (see C.3) takes one or four command line arguments:
random [<object size> <attribute size> <probability>] <context file>

C.1 randomGen.h

"randomGen.h" 55a ≡

```
#include <iostream>
#include <sys/types.h>
//#include <time.h>
#include <stdlib.h>
#include <vector>
#include <sys/time.h>
#include <sys/times.h>
#include <sys/types.h>
#include <unistd.h>

#ifndef RANDOMGEN_H
#define RANDOMGEN_H

class randomGen
{
public:
    randomGen(int obj_size, int attr_size, double probability);
    bool outputContext(ostream&);
private:
    void randomizeContext();
    vector<vector<bool> > incidence;
    double probability;
};

#endif
◇
```

C.2 randomGen.cc

"randomGen.cc" 55b ≡

```
#include "randomGen.h"

randomGen::randomGen(int obj_size, int attr_size, double p) :
    incidence(obj_size, vector<bool>(attr_size, false)),
    probability(p)
{
    struct timeval curtime;
    gettimeofday(&curtime, NULL);
    srand((curtime.tv_sec) * 1000000 + curtime.tv_usec);
    randomizeContext();
}
```

```

void randomGen::randomizeContext()
{
    for(vector<vector<bool> >::iterator i = incidence.begin(); i < incidence.end(); ++i)
        for(vector<bool>::iterator j = i->begin(); j < i->end(); ++j)
            {
                *j = ( (random() % 100000) * 0.00001 < probability );
                //cout << (*j?"TRUE":"FALSE") << endl;
            }
}

bool randomGen::outputContext(ostream& out)
{
    for(int i=0;i<incidence[0].size();++i)
        out << i << endl;
    out << "--" << endl;
    for(vector<vector<bool> >::const_iterator i = incidence.begin(); i < incidence.end(); ++i)
        {
            out << i - incidence.begin() << ":\t";
            for(vector<bool>::const_iterator j = i->begin(); j < i->end(); ++j)
                if(*j)
                    out << j - i->begin() << '\t';
            out << ";" << endl;
        }
    return !(out.fail());
}
}

```

◇

C.3 random.cc

"random.cc" 56 ≡

```

#include "randomGen.h"
#include <stdlib.h>
#include <fstream>
#include <string>
#include <sys/types.h>
#include <time.h>

void parse_command_line(int&, int&, double&, string&, const int, const char**);

int main(const int argc, const char** argv)
{
    string filename;
    int obj_size, attr_size;
    double probability;

    parse_command_line(obj_size, attr_size, probability, filename, argc, argv);

    randomGen r(obj_size, attr_size, probability);
}

```

```

ofstream out(filename.c_str());
if(!out)
{
    cerr << "Could not open file \"" << filename << "\" for writing" << endl;
    exit(1);
}
r.outputContext(out);

return 0;
}

void parse_command_line(int& obj_size, int& attr_size, double& probability,
                        string& filename, const int argc, const char** argv)
{
    string message = "usage: \nrandom [obj_size attr_size probability] filename\nobj_size:\tdes
number of objects\nattr_size:\tdesired number of attributes\nprobability:\tprobability that t
incidence of an object and attribute will be set\nfilename:\tname of output file\ndescription
generates a random context file";

    if (argc == 5)
    {
        obj_size = atoi(argv[1]);
        attr_size = atoi(argv[2]);
        probability = atof(argv[3]);
        filename = argv[4];
    }
    else if(argc == 2)
    {
        filename = argv[1];
        struct timeval curtime;
        gettimeofday(&curtime, NULL);
        srand((curtime.tv_sec) * 1000000 + curtime.tv_usec);
        obj_size = (random() % 80) + 2;
        attr_size = (random() % 81) + 1;
        for(int k = 0; k < 10; ++k)
        {
            probability = (random() % 100000) * 0.00001;
            if(probability < 0.35)
            {
                break;
            }
        }
        if(obj_size * attr_size > 700)
            while(probability > .2)
                probability = (random() % 100000) * 0.00001;
        cout << "Probability = " << probability << endl;
    }
    else

```

```
    {  
      cerr << message << endl;  
      exit(1);  
    }  
  }  
  ◊
```

D Bibliography

References

- [1] Lindig, Christian. *Fast Concept Analysis*
Cambridge, MA, 2000
- [2] B. Ganter, R. Wille. *Applied Lattice Theory:
Formal Concept Analysis*.
- [3] D.R. Musser, A. Saini. *STL Tutorial and Reference Guide:
Programming with Standard Template Library*. Addison-Wesley,
Reading, MA, 1996.
- [4] Mathew H. Austern *Generic Programming
and the STL*. Addison-Wesley, Reading, MA, 1998.
- [5] Silicon Graphics Standard Template Library Programming Guide,
online guide, <http://www.sgi.com/Technology/STL/>.
- [6] Jos van Eijndhoven, *graphplace*
Eindhoven University of Technology, The Netherlands.