# Lattice Theory with Applications

Vijay K. Garg
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712-1084

*To*
*my teachers and*
*my students*

# Contents

# List of Figures

# Preface

Partial order and lattice theory now play an important role in many disciplines of computer science and engineering. For example, they have applications in distributed computing (vector clocks, global predicate detection), concurrency theory (pomsets, occurrence nets), programming language semantics (fixed-point semantics), and data mining (concept analysis). They are also useful in other disciplines of mathematics such as combinatorics, number theory and group theory. In this book, I introduce important results in partial order theory along with their applications in computer science. The bias of the book is on computational aspects of lattice theory (algorithms) and on applications (esp. distributed systems).

Since many books exist on lattice theory, I must justify writing another book on the subject. This book is written from the perspective of a computer scientist rather than a mathematician. In many instances, a mathematician may be satisfied with a non-constructive existential proof — a proof that may not be of much use to a computer scientist who is not only interested in construction but also the computational complexity of the construction. I have attempted to give "algorithmic" proof of theorems whenever possible.

This book is also written for the sake of students rather than a practicing mathematician. In many instances, a student needs to learn the heuristics that guide the proof, besides the proof itself. It is not sufficient to learn an important theorem. One must also learn ways to extend and analyze a theorem.

I have made an effort to include some exercises with each chapter. A mathematical subject can only be learned by solving problems relatd to that subject.

I would like to thank the students in the class of Fall 2003 at the University of Texas at Austin, who took my course on lattice theory. The notes scribed by these students formed a very convenient reference point for this book. and Brian Waldecker (weak and strong predicates). Anurag Agarwal, Arindam Chakraborty, Selma Ikiz, Neeraj Mittal, Sujatha Kashyap, I owe special thanks to Omer Shakil, Alper Sen, and Roopsha Samanta who reviewed parts of the book. Roopsha Samanta, in particular, pointed out many inconsistencies in the first draft and also helped me with figures.

I thank the Department of Electrical and Computer Engineering at The University of Texas at Austin, where I was given the opportunity to develop and teach a course on lattice theory.

I was supported in part by many grants from the National Science Foundation over the last 14 years. Many of the results reported in this book would not have been discovered by me and my research group without that support. I also thank John Wiley & Sons, Inc. for supporting the project.

Finally, I thank my parents, wife and children. Without their love and support, this book would not have been even conceived.

The list of known errors and the supplementary material for the book will be maintained on my homepage:

http://www.ece.utexas.edu/~garg

Vijay K. Garg

Austin, Texas.

# Chapter 1

# Introduction

## 1.1 Introduction

Partial order and lattice theory now play an important role in many disciplines of computer science and engineering. For example, they have applications in distributed computing (vector clocks, global predicate detection), concurrency theory (pomsets, occurrence nets), programming language semantics (fixed-point semantics), and data mining (concept analysis). They are also useful in other disciplines of mathematics such as combinatorics, number theory and group theory.

This book differs from earlier books written on the subject in two aspects. First, the present book takes a computational perspective — the emphasis is on algorithms and their complexity. While mathematicians generally identify necessary and sufficient conditions to characterize a property, this book focuses on efficient algorithms to test the property. As a result of this bias, much of the book concerns itself only with finite sets. Second, existing books do not dwell on applications of lattice theory. This book treats applications at par with the theory. In particular, applications of lattice theory to distributed computing are treated in extensive detail. I have also shown many applications to combinatorics because the theory of partial orders forms a core topic in combinatorics.

This chapter covers the basic definitions of partial orders.

## 1.2 Relations

A partial order is simply a relation with certain properties. A **relation** $R$ over any set $X$ is a subset of $X \times X$. For example, let

$$X = \{a, b, c\}.$$

Then, one possible relation is

$$R = \{(a, c), (a, a), (b, c), (c, a)\}.$$

It is sometimes useful to visualize a relation as a graph on the vertex set $X$ such that there is a directed edge from $x$ to $y$ iff $(x, y) \in R$. The graph corresponding to the relation $R$ in the previous example is shown in Figure 1.1.

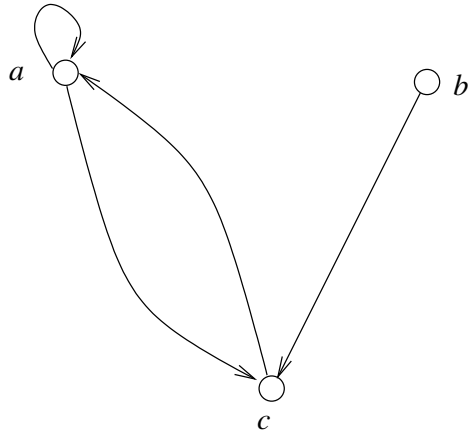



Figure 1.1: The graph of a relation

A relation is **reflexive** if for each $x \in X$, $(x, x) \in R$. In terms of a graph, this means that there is a self-loop on each node. If $X$ is the set of natural numbers, $\mathcal{N}$, then "$x$ divides $y$" is a reflexive relation. R is **irreflexive** if for each $x \in X$, $(x, x) \notin R$. In terms of a graph, this means that there are no self-loops. An example on the set of natural numbers, $\mathcal{N}$, is the relation "$x$ less than $y$." Note that a relation may be neither reflexive nor irreflexive.

A relation R is **symmetric** if for all $x, y \in X$, $(x, y) \in R$ implies $(y, x) \in R$. An example of a symmetric relation on $\mathcal{N}$ is

$$R = \{(x, y) \mid x \ mod \ 5 = y \ mod \ 5\}. \tag{1.1}$$

A symmetric relation can be represented using an undirected graph. R is **antisymmetric** if for all $x, y \in X$, $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$. For example, the relation *less than or equal to* defined on $\mathcal{N}$ is anti-symmetric. A relation $R$ is **asymmetric** if for all $x, y \in X$, $(x, y) \in R$ implies $(y, x) \notin R$. The relation *less than* on $\mathcal{N}$ is asymmetric. Note that an asymmetric relation is always irreflexive.

A relation R is **transitive** if for all $x, y, z \in X$, $(x, y) \in R$ and $(y, z) \in R$ implies $(x, z) \in R$. The relations *less than* and *equal to* on $\mathcal{N}$ are transitive.

A relation R is an **equivalence** relation if it is reflexive, symmetric, and transitive. When $R$ is an equivalence relation, we use $x \equiv_R y$ (or simply $x \equiv y$ when $R$ is clear from the context) to denote that $(x, y) \in R$. Furthermore, for each $x \in X$, we use $[x]_R$, called the **equivalence class of** $x$, to denote the set of all $y \in X$ such that $y \equiv_R x$. It can be seen that the set of all such equivalence classes forms a **partition** of $X$. The relation on $\mathcal{N}$ defined in (1.1) is an example of an equivalence relation. It partitions the set of natural numbers into five equivalence classes.

Given any relation $R$ on a set $X$, we define its irreflexive transitive closure, denoted by $R^+$, as follows. For all $x, y \in X : (x, y) \in R^+$ iff there exists a sequence $x_0, x_1, ..., x_j, j \geq 1$ with $x_0 = x$ and $x_j = y$ such that

$$\forall i : 0 \leq i < j : (x_i, x_{i+1}) \in R.$$

Thus $(x, y) \in R^+$ iff there is a nonempty path from $x$ to $y$ in the graph of the relation $R$. We define the reflexive transitive closure, denoted by $R^*$, as

$$R^* = R^+ \cup \{(x, x) \mid x \in X\}$$

Thus $(x, y) \in R^*$ iff $y$ is reachable from $x$ by taking a path with zero or more edges in the graph of the relation $R$.

## 1.3 Partial Orders

A relation R is a **reflexive partial order** (or, a **non-strict partial order**) if it is reflexive, antisymmetric, and transitive. The *divides* relation on the set of natural numbers is a reflexive partial order. A relation $R$ is an **irreflexive partial order**, or a **strict partial order** if it is irreflexive and transitive. The *less than* relation on the set of natural numbers is an irreflexive partial order. When $R$ is a reflexive partial order, we use $x \leq_R y$ (or simply $x \leq y$ when $R$ is clear from the context) to denote that $(x, y) \in R$. A reflexive partially ordered set, **poset** for short, is denoted by $(X, \leq)$. When $R$ is an irreflexive partial order, we use $x <_R y$ (or simply $x < y$ when $R$ is clear from the context) to denote that $(x, y) \in R$. The set $X$ together with the partial order is denoted by $(X, <)$. We use $P = (X, <)$ to denote the poset.

The two versions of partial orders — reflexive and irreflexive — are essentially the same. Given an irreflexive partial order, we can define $x \leq y$ as $x < y$ or $x = y$ which gives us a reflexive partial order. Similarly, given a reflexive partial order $(X, \leq)$, we can define an irreflexive partial order $(X, <)$ by defining $x < y$ as $x \leq y$ and $x \neq y$. In this book, we use a *poset* to mean a set $X$ with either an irreflexive partial order or a reflexive partial order.

A relation is a **total order** if $R$ is a partial order and for all distinct $x, y \in X$, either $(x, y) \in R$ or $(y, x) \in R$. The natural order on the set of integers is a total order, but the "divides" relation is only a partial order.

### 1.3.1 Hasse Diagrams

Finite posets are often depicted graphically using **Hasse diagrams**. To define Hasse diagrams, we first define a relation **covers** as follows. For any two elements $x, y \in X$, $y$ covers $x$ if $x < y$ and $\forall z \in X : x \leq z < y$ implies $z = x$. In other words, there should not be any element $z$ with $x < z < y$. We use $x \prec y$ to denote that $y$ covers $x$ (or $x$ is covered by $y$). We also say that $y$ is an **upper cover** of $x$ and $x$ is a **lower cover** of $y$. A Hasse diagram of a poset is a graph with the property that there is an edge from $x$ to $y$ iff $x \prec y$. Furthermore, when drawing the graph on a Euclidean plane, $x$ is drawn lower than $y$ when $y$ covers $x$. This allows us to suppress the directional arrows in the edges. For example, consider the following poset $(X, \leq)$,

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s)\}. \tag{1.2}$$

Its Hasse diagram is shown in Figure 1.2. Note that we will sometimes use directed edges in Hasse diagrams if the context demands it. In general, in this book, we switch between the directed graph and undirected graph representations of Hasse diagrams.

### 1.3.2 Chains and Antichains

Given a poset $(X, \leq_X)$ a **subposet** is simply a poset $(Y, \leq_Y)$ where $Y \subseteq X$, and

$$\forall x, y \in Y : x \leq_Y y \stackrel{\text{def}}{=} x \leq_X y$$

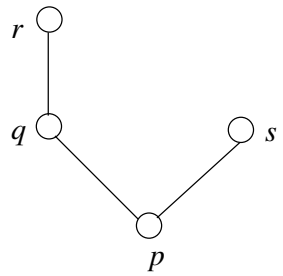



Figure 1.2: Hasse diagram

Let $x, y \in X$ with $x \neq y$. If either $x < y$ or $y < x$, we say $x$ and $y$ are **comparable**. On the other hand, if neither $x < y$ nor $x > y$, then we say $x$ and $y$ are incomparable, and write $x||y$. A poset $(Y, \leq)$ (or a subposet $(Y, \leq)$ of $(X, \leq)$) is called a **chain** if every distinct pair of elements from $Y$ is comparable. Similarly, we call a poset an **antichain** if every distinct pair of elements from $Y$ is incomparable. For example, for the poset represented in Figure 1.2, $\{q, r\}$ is a chain, and $\{q, s\}$ is an antichain.

A chain $C$ of a poset $(X, \leq)$ is a **maximum chain** if no other chain contains more elements than $C$. We use a similar definition for **maximum antichain**. The **height** of the poset is the number of elements in the maximum chain, and the **width** of the poset is the number of elements in a maximum antichain.For example, the poset in Figure 1.2 has height equal to 3 (the maximum chain is $\{p, q, r\}$) and width equal to 2 (a maximum antichain is $\{q, s\}$).

### 1.3.3 Join and Meet Operations

We now define two operators on subsets of the set X—**meet or infimum** (or **inf**) and **join** or **supremum** (or **sup**). Let $Y \subseteq X$, where $(X, \leq)$ is a poset. For any $m \in X$, we say that $m = inf\ Y$ iff

1. $\forall y \in Y : m \leq y$, and
2. $\forall m' \in X : (\forall y \in Y : m' \leq y) \Rightarrow m' \leq m$.

The condition (1) says that $m$ is a lower bound of the set $Y$. The condition (2) says that if $m'$ is another lower bound of $Y$, then it is less than $m$. For this reason, $m$ is also called the **greatest lower bound** ($glb$) of the set $Y$. It is easy to check that the infimum of $Y$ is unique whenever it exists. Observe that $m$ is not required to be an element of $Y$.

The definition of $sup$ is similar. For any $s \in X$, we say that $s = sup\ Y$ iff

1. $\forall y \in Y : y \leq s$
2. $\forall s' \in X : (\forall y \in Y : y \leq s') \Rightarrow s \leq s'$

Again, $s$ is also called the **least upper bound** ($lub$) of the set $Y$. We denote the **glb** of $\{a, b\}$ by $a \sqcap b$, and **lub** of $\{a, b\}$ by $a \sqcup b$. In the set of natural numbers ordered by the *divides* relation, the $glb$ corresponds to finding the greatest common divisor (gcd) and the $lub$ corresponds to finding the least common multiple of two natural numbers. The greatest lower bound or the least upper bound may not always exist. In Figure 1.3, the set $\{e, f\}$ does not have any upper bound. In the

third poset in Figure 1.4, the set $\{b, c\}$ does not have any least upper bound (although both $d$ and $e$ are upper bounds).

The following lemma relates $\leq$ to the meet and join operators.

**Lemma 1.1** *[Connecting Lemma]*

1. $x \leq y \equiv (x \sqcup y) = y$, *and*
2. $x \leq y \equiv (x \sqcap y) = x$.

**Proof:**

$x \leq y$ implies that $y$ is an upper bound on $\{x, y\}$. $y$ is also the least upper bound because any upper bound of $\{x, y\}$ is greater than both $x$ and $y$. Therefore, $(x \sqcup y) = y$. Conversely, $(x \sqcap y) = y$ means $y$ is an upper bound on $\{x, y\}$. Therefore, $x \leq y$.

The proof for the second part is the dual of this proof.

■

Lattices, which are special kinds of posets, can be defined using the join and meet operators as shown below.

**Definition 1.2 (Lattice)** *A poset $(X, \leq)$ is a* **lattice** *iff $\forall x, y \in X : x \sqcup y$ and $x \sqcap y$ exist.*

**Definition 1.3 (Distributive Lattice)** *A lattice $L$ is distributive if*

$$\forall a, b, c \in L : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

$$X = \{a, b, c, d, e, f\}$$

$$R = \left\{ \begin{array}{l} (a, b), (a, c), (b, d), \\ (c, f), (c, e), (d, e) \\ (a, d), (a, e), (a, f), \\ (b, e) \end{array} \right\}$$

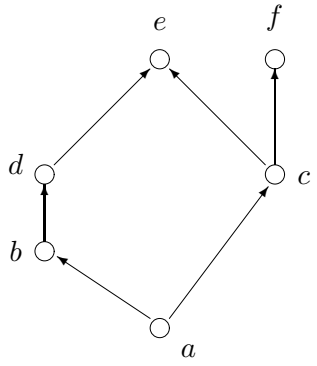



Figure 1.3: The set $\{e, f\}$ does not have any upper bound.

### 1.3.4 Locally Finite Posets

Generalizing the notation for intervals on the real-line, we define an **interval** $[x, y]$ in a poset $(X, \leq)$ as

$$\{z | x \leq z \leq y\}$$

The meaning of $(x, y)$ and $[x, y)$ and $(x, y]$ is similar. A poset is **locally finite** if all intervals are finite. Most posets in this book will be locally finite if not finite.

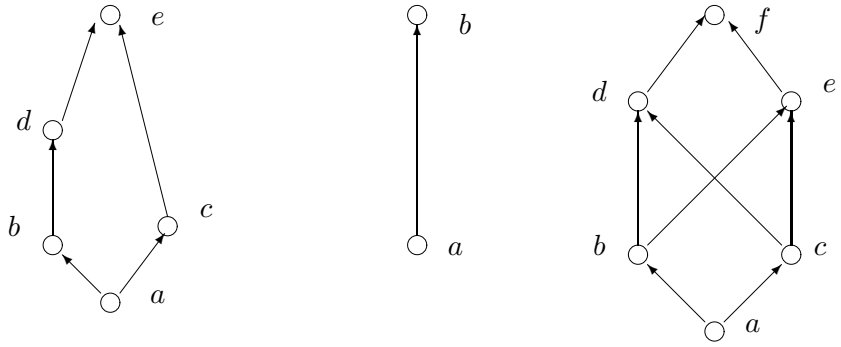



Figure 1.4: Various posets.

### 1.3.5 Well-Founded Posets

A poset is **well-founded** iff it has no infinite decreasing chain. The set of natural numbers under the usual relation is well-founded but the set of integers is not well-founded. A poset corresponding to a distributed computation may be infinite but is usually well-founded.

### 1.3.6 Poset Extensions

Poset $Q = (X, \leq_Q)$ extends the poset $P = (X, \leq_P)$ if

$$\forall x, y \in X : x \leq_P y \Rightarrow x \leq_Q y$$

If $Q$ is a total order, then we call $Q$ a linear extension of $P$. For example, for the poset $(X, \leq)$ defined in Figure (1.2), a possible **linear extension** $Q$ is

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq_Q \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s), (q, s), (r, s)\}.$$

## 1.4 Ideals and Filters

Let $(X, \leq)$ be any poset. We call a subset $Y \subseteq X$ a **down-set** if

$$\forall y, z \in X : z \in Y \wedge y \leq z \Rightarrow y \in Y.$$

Down-sets are also called **order ideals**. It is easy to see that for any $x$, $D[x]$ defined below is an order ideal. Such order ideals are called **principal order ideals**,

$$D[x] = \{y \in X | y \leq x\}.$$

For example, in Figure 1.2, $D[r] = \{p, q, r\}$. Similarly, we call $Y \subseteq X$ an **up-set** if

$$y \in Y \wedge y \leq z \Rightarrow z \in Y.$$

Up-sets are also called **order filters**. We also use the notation below to denote **principal order filters**

$$U[x] = \{y \in X | x \leq y\}.$$

In Figure 1.2, $U[p] = \{p, q, r, s\}$.

The following lemmas provides a convenient relationship between principal order filters and other operators defined earlier.

**Lemma 1.4** *1.* $x \leq y \equiv U[y] \subseteq U[x]$

*2.* $x = sup(Y) \equiv U[x] = \cap_{y \in Y} U[y]$

**Proof:** Left as an exercise.

■

In some applications, the following notation is also useful:

$$U(x) = \{y \in X | x < y\}$$

$$D(x) = \{y \in X | y < x\}.$$

We will call $U(x)$ the **upper-holdings** of $x$, and $D(x)$, the **lower-holdings** of $x$. We extend the definitions of $D[x], D(x), U[x], U(x)$ to sets of elements, $A$. For example,

$$U[A] = \{y \in X | \exists x \in A : x \leq y\}.$$

## 1.5 Special Elements in Posets

In this section we define some special elements of posets such as bottom and top elements, irreducibles and dissectors.

### 1.5.1 Bottom and Top Elements

An element $x$ is a **bottom** element or a **minimum** element of a poset $P$ if $x \in P$, and

$$\forall y \in P : x \leq y$$

For example, 0 is the bottom element in the poset of whole numbers, and $\emptyset$ is the bottom element in the poset of all subsets of a given set $W$. Similarly, an element $x$ is a **top** element, or a **maximum** element of a poset $P$ if $x \in P$, and

$$\forall y \in P : y \leq x$$

A bottom element of the poset is denoted by $\perp$ and the top element by $\top$. It is easy to verify that if bottom and top elements exist, they are unique.

### 1.5.2 Minimal and Maximal Elements

An element $x$ is a **minimal** element of a poset $P$ if

$$\forall y \in P : y \not< x.$$

The minimum element is also a minimal element. However, a poset may have more than one minimal element. Similarly, an element $x$ is a **maximal** element of a poset $P$ if

$$\forall y \in P : y \not> x.$$

### 1.5.3 Irreducible Elements

**Definition 1.5** *An element $x$ is join-irreducible in $P$ if it cannot be expressed as join of other elements of $P$. Formally, $x$ is* **join-irreducible** *if*

$$\forall Y \subseteq P : x = sup(Y) \Rightarrow x \in Y$$

Note that if a poset has $\perp$, then it is not join-irreducible because when $Y = \{\}$, $sup(Y)$ is $\perp$. The set of join-irreducible elements of a poset $P$ will be denoted by $\mathcal{J}(P)$. The first poset in Figure 1.4 has $b$,$c$ and $d$ as join-irreducible. The element $e$ is not join-irreducible in the first poset becase $e = b \sqcup c$. The second poset has $b$ and the third poset has $b$, $c$, $d$ and $e$ as join-irreducible. It is easy to verify that $x \notin \mathcal{J}(P)$ is equivalent to $x = sup(D(x))$.

By duality, we can define meet-irreducible elements of a poset, denoted by $\mathcal{M}(P)$.

Loosely speaking, the set of (join)-irreducible elements forms a basis of the poset because all elements can be expressed using these elements. More formally,

**Theorem 1.6** *Let $P$ be a finite poset. Then, for any $x \in P$*

$$x = sup(D[x] \cap \mathcal{J}(P))$$

**Proof:** We use $I[x]$ to denote the set $D[x] \cap \mathcal{J}(P)$. We need to show that $x = sup(I[x])$ for any $x$. The proof is by induction on the cardinality of $D[x]$.

*(Base case)* $D[x]$ is singleton.
This implies that $x$ is a minimal element of the poset. If $x$ is the unique minimum, $x \notin \mathcal{J}(P)$ and $I[x]$ is empty; therefore, $x = sup(I[x]$. Otherwise, $x \in \mathcal{J}(P)$ and we get that $x = sup(I[x])$.

*(Induction case)* $D[x]$ is not singleton.
First assume that $x$ is join-irreducible. Then, $x \in I[x]$ and all other elements in $I[x]$ are smaller than $x$. Therefore, $x$ equals $sup(I[x])$. If $x$ is not join-irreducible, then $x = sup(D(x))$. By induction, each $y \in D(x)$ satisfies $y = sup(I[y])$. Therefore, we have

$$\forall y \in D(x) : U[y] = \cap_{z \in I[y]} U[z] \tag{1.3}$$

We now have,
$x = sup(D(x))$
$\equiv$ { property of $sup$ }
$U[x] = \cap_{y \in D(x)} U[y]$
$\equiv$ { Equation 1.3 }
$U[x] = \cap_{y \in D(x)} \cap_{z \in I[y]} U[z]$
$\equiv$ { definition of $I[y]$ }
$U[x] = \cap_{y \in D(x)} \cap_{z \in (D[y] \cap \mathcal{J}(P))} U[z]$
$\equiv$ { definition of $I[x]$ and simplification }
$U[x] = \cap_{z \in I[x]} U[z]$
$\equiv$ { property of $sup$ }
$x = sup(I[x])$

■

An equivalent characterization of join-irreducible element is as follows.

**Theorem 1.7** *For a finite poset $P$, $x \in \mathcal{J}(P)$ iff $\exists y \in P : x \in minimal(P - D[y])$*

**Proof:** First assume that $x \in \mathcal{J}(P)$.

Let $LC(x)$ be the set of elements covered by $x$. If $LC(x)$ is singleton, then choose that element as $y$. It is clear that $x \in minimal(P - D[y])$.

Now consider the case when $LC(x)$ is not singleton (it is empty or has more than one element). Let $Q$ be the set of upper bounds for $LC(x)$. $Q$ is not empty because $x \in Q$. Further, $x$ is not the minimum element in $Q$ because $x$ is join-irreducible. Pick any element $y \in Q$ that is incomparable to $x$. Since $D[y]$ includes $LC(x)$ and not $x$, we get that $x$ is minimal in $P - D[y]$.

The converse is left as an exercise.

■

### 1.5.4 Dissector Elements

We now define a subset of irreducibles called dissectors.

**Definition 1.8** *For a poset $P$, $x \in P$ is an upper dissector if there exists $y \in P$ such that*

$$P - U[x] = D[y]$$

In the above definition $y$ is called a lower dissector. We will use dissectors to mean upper dissectors.

A dissector $x$ decomposes the poset into two parts $U[x]$ and $D[y]$ for some $y$. In the first poset in Figure 1.4, $b$ is an upper dissector because $P - U[b] = P - \{b, d, e\} = \{a, c\} = D[c]$. However, $d$ is not an upper dissector because $P - U[d] = \{a, b, c\}$ which is not a principal ideal.

The following result is an easy implication of Theorem 1.7.

**Theorem 1.9** *$x$ is a dissector implies that $x$ is join-irreducible.*

**Proof:** If $x$ is an upper dissector, then there exists $y$ such that $x$ is *minimum* in $P - D[y]$. This implies that $x$ is minimal in $P - D[y]$.

■

## 1.6 Applications

### 1.6.1 Distributed Computations

We will be concerned with a single computation of a distributed program. Each process $P_i$ in that computation generates a sequence of *events*. It is clear how to order events within a single process. If event $e$ occurred before $f$ in the process, then $e$ is ordered before $f$. How do we order events across processes? If $e$ is the send event of a message and $f$ is the receive event of the same message, then we can order $e$ before $f$. Combining these two ideas, we obtain the following definition.

**Definition 1.10 (Happened-Before Relation)** *The happened-before relation ($\rightarrow$) is the smallest relation that satisfies*

1. *If $e$ occurred before $f$ in the same process, then $e \rightarrow f$,*
2. *If $e$ is the send event of a message and $f$ is the receive event of the same message, then $e \rightarrow f$, and*
3. *If there exists an event $g$ such that $(e \rightarrow g)$ and $(g \rightarrow f)$, then $(e \rightarrow f)$.*

In Figure 1.5, $e_2 \rightarrow e_4$, $e_3 \rightarrow f_3$, and $e_1 \rightarrow g_4$.

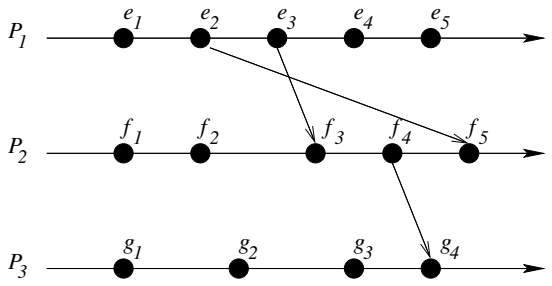



Figure 1.5: A computation in the happened-before model

A *computation* in the happened-before model is defined as a tuple $(E, \rightarrow)$ where $E$ is the set of all events and $\rightarrow$ is a partial order on events in $E$ such that all events within a single process are totally ordered. Figure 1.5 illustrates a computation.

### 1.6.2 Integer Partitions

We now discuss an example from combinatorics.

**Definition 1.11** *$\lambda = (\lambda_1, ..., \lambda_k)$ is an unordered partition of the integer $n$ if*

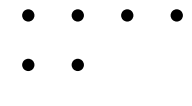
1. $\forall i \in [1, k-1] : \lambda_i \geq \lambda_{i+1}$
2. $\sum_{i=1}^{k} \lambda_i = n$
3. $\forall i : \lambda_i \geq 1$

**Definition 1.12** *A Ferrer's diagram for an integer partition $\lambda = (\lambda_1, ..., \lambda_k)$ of integer $n$ is a matrix of dots where the $i^{th}$ row contains $\lambda_i$ dots (example Figure 11.4). Thus each row represents the size of a partition and the number of rows represents the number of parts in the partition.*

We define an order on the partitions as follows: Given two partitions $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_m), \delta = (\delta_1, \delta_2, \ldots, \delta_n)$, we say that $\lambda \geq \delta$ iff $m \geq n$ and $\forall i : 1 \leq i \leq n, \lambda_i \geq \delta_i$. This can also be viewed in terms of containment in the Ferrer's diagram, i.e. $\lambda \geq \delta$ if the Ferrer's diagram for $\delta$ is contained in the Ferrer's diagram of $\lambda$. For example, consider the partitions $(4, 3, 2)$ and $(2, 2, 2)$. The Ferrer's diagram for $(2, 2, 2)$ is contained in the Ferrer's diagram for $(4, 3, 2)$ as shown in Figure 11.6. Hence, $(4, 3, 2) > (2, 2, 2)$.

**Definition 1.13** *Given a partition $\lambda$,* Young's lattice *$Y_\lambda$ is the poset of all partitions that are less than or equal to $\lambda$.*

The Young's lattice for $(3, 3, 3)$ is shown in figure 14.9. Note that partitions less than a partition are not necessarily the partitions of the same integer.



$$7 = 4 + 2 + 1$$

Figure 1.6: Ferrer's diagram for the integer partition $(4, 2, 1)$ for 7.

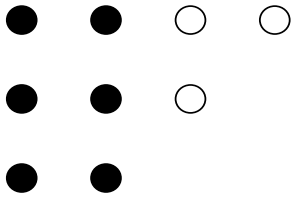


Figure 1.7: Ferrer's diagram for $(4, 3, 2)$ shown to contain $(2, 2, 2)$

## 1.7 Problems

1. Give an example of a nonempty binary relation which is symmetric and transitive but not reflexive.

2. The *transitive closure* of a relation $R$ on a finite set can also be defined as the smallest transitive relation on $S$ that contains $R$. Show that the transitive closure is uniquely defined. We use "smaller" in the sense that $R_1$ is smaller than $R_2$ if $|R_1| < |R_2|$.

3. Show that if $P$ and $Q$ are posets defined on set $X$, then so is $P \cap Q$.

4. Draw the Hasse diagram of all natural numbers less than 10 ordered by the relation *divides*.

5. Show that if $C_1$ and $C_2$ are down-sets for any poset $(E, <)$, then so is $C_1 \cap C_2$.

6. Show the following properties of $\sqcup$ and $\sqcap$.

| | | |
|---|---|---|
| $(L1)$ | $a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$ | (associativity) |
| $(L1)^\delta$ | $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$ | |
| $(L2)$ | $a \sqcup b = b \sqcup a$ | (commutativity) |
| $(L2)^\delta$ | $a \sqcap b = b \sqcap a$ | |
| $(L3)$ | $a \sqcup (a \sqcap b) = a$ | (absorption) |
| $(L3)^\delta$ | $a \sqcap (a \sqcup b) = a$ | |

7. Prove Lemma 1.4.

8. Complete the proof of Theorem 1.7.

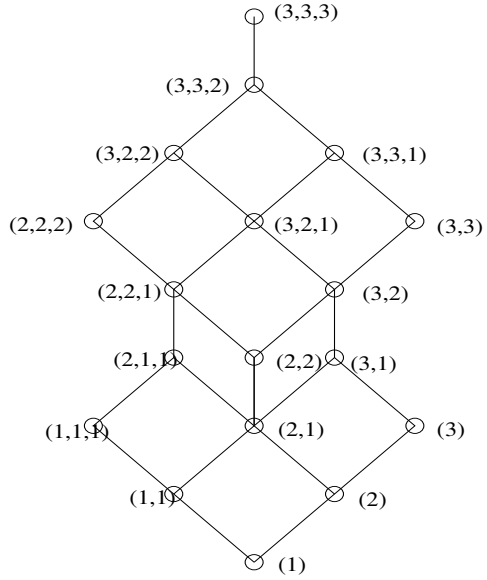



Figure 1.8: Young's lattice for $(3, 3, 3)$

## 1.8 Bibliographic Remarks

The first book on lattice theory was written by Garrett Birkhoff [Bir40, Bir48, Bir67]. The reader will find a discussion of origins of lattice theory and an extensive bibliography in the book by Gratzer [Grä71, Grä03]. The book by Davey and Priestley[DP90] provides an easily accessible account of the field.

Reading[**?**] gives a detailed discussion of dissectors and their properties.

# Chapter 2

# Representing Posets

## 2.1 Introduction

We now look at some possible representations for efficient queries and operations on posets. For this purpose, we first examine some of the common operations that are generally performed on posets. Given a poset $P$, the following operations on elements $x, y \in P$ are frequently performed :

1. Check if $x \leq y$. We denote this operation by $leq(x, y)$.
2. Check if $x$ covers $y$
3. Compute $x \sqcap y$ and $x \sqcup y$

In all representations, we first need to number elements of the poset. One profitable way of doing this is discussed in Section 2.2. In the following sections, we consider some representations for posets and compare their performance for the set of operations mentioned above.

## 2.2 Labeling Elements of the Poset

Assume that a finite poset with $n$ elements is given to us as a directed graph (Hasse diagram) where nodes represent the elements in the poset and the edges represent the cover relation on the elements. We number the nodes, i.e., provide a label to each node from $\{1..n\}$ such that if $x < y$ then $label(x) < label(y)$. This way we can answer some queries of the form "Is $x < y$ ?" directly without actually looking at the lists of edges of the elements $x$ and $y$. For example, if $label(x)$ equals 5 and $label(y)$ equals 3 then we know that $x$ is not less than $y$. Note that $label(x) < label(y)$ does not necessarily imply that $x < y$. It only means that $x \not> y$. One straightforward way of generating such a labeling would be :

1. Choose a node $x$ with in-degree 0
2. Delete $x$ and all outgoing edges from $x$
3. Repeat step 1 until there are no nodes left in the graph

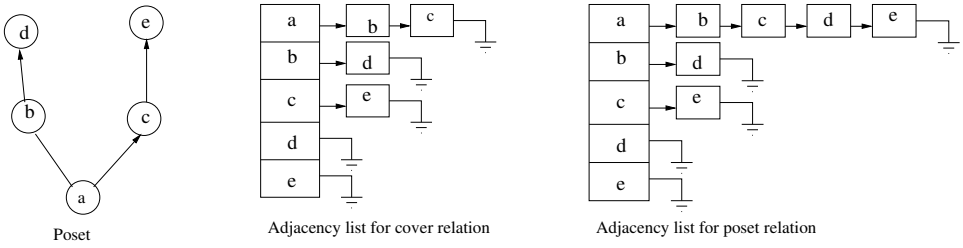



Figure 2.1: Adjacency list representation of a poset

The above procedure is called **topological sort** and it is easy to show that it produces a linear extension of the poset. The procedure also shows that every finite poset has an extension. The following result due to Szpilrajn shows that this is true for all posets, even the infinite ones.

**Theorem 2.1** *([Szp37]) Every partial order $P$ has a linear extension.*

**Proof:** Let $\mathcal{W}$ be the set of all posets that extend $P$. Define a poset $Q$ to be less than or equal to $R$ is the relation corresponding to $Q$ is included in the relation $R$. Consider a maximal element $Z$ of the set $\mathcal{W}$[1]. If $Z$ is totally ordered, then we are done as $Z$ is the linear extension. Otherwise assume that $x$ and $y$ are incomparable in $Z$. By adding the tuple $(x, y)$ to $Z$ and computing its transitive closure we get another poset in $\mathcal{W}$ that is bigger than $Z$ contradicting maximality of $Z$.

■

## 2.3 Adjacency List Representation

Since a poset is a graph, we can use an adjacency list representation as is typically done for a graph. In an adjacency list representation, we have a linked list for each element of the poset. We can maintain two relationships through a adjacency list :

1. Cover Relation ($e_{\prec}$) : The linked list for element $x \in P = \{y : y \in P, x \prec y\}$
2. Poset Relation ($e_{\leq}$) : The linked list for element $x \in P = \{y : y \in P, x \leq y\}$

Figure 2.1 shows an example of a poset with its adjacency list representation for both the relations. Choosing between the two relations for creating an adjacency list involves a tradeoff. Representing $e_{\prec}$ requires less space, while most operations are faster with the $e_{\leq}$ representation. For example, representing a chain of $n$ elements using $e_{\prec}$ requires $O(n)$ space as compared to $O(n^2)$ space using $e_{\leq}$. On the other hand, checking $x \leq y$ for some elements $x, y \in P$ requires time proportional to the size of the linked list of $x$ using $e_{\leq}$ as compared to $O(n + |e_{\prec}|)$ time using $e_{\prec}$.

For very large graphs, one can also keep all adjacent elements of a node in a (balanced) tree instead of a linked list. This can cut down the worst case complexity for answering queries of the form $x \prec y$ or $x \leq y$. Balanced trees are standard data structures and will not be discussed any further. For simplicity we will continue to use linked lists. Next, we discuss algorithms when the poset is represented using adjacency lists representing (1) the cover relation, and (2) the poset relation.

[1]This step requires Zorn's lemma which is equivalent to the axiom of choice

### 2.3.1 Cover Relation Representation with Adjacency List

Since the adjacency list corresponds to the cover relation, the query "Does $y$ cover $x$?" can be answered by simply traversing the linked list for node $x$. The query "Is $x \leq y$?" requires a Breadth-First Search (BFS) or Depth-First Search (DFS) starting from node $x$ resulting in time $O(n + |e_{\prec}|)$ as mentioned before.

We now give an $O(n + |e_{\prec}|)$ algorithm to compute the join of two elements $x$ and $y$. The algorithm returns *null* if the join does not exist. We first assume that we are using the cover relation. To compute $join(x, y)$, we proceed as follows:

- Step 0: Color all nodes white.
- Step 1: Color all nodes reachable from $x$ as grey. This can be done by a BFS or a DFS starting from node $x$.
- Step 2: Do a BFS/DFS from node $y$. Color all grey nodes reached as black.
- Step 3: We now determine for each black node $z$, the number of black nodes that point to $z$. Call this number $inBlack[z]$ for any node $z$. This step can be performed in $O(n + |e_{\prec}|)$ by going through the adjacency lists of all black nodes and maintaining cumulative counts for $inBlack$ array.
- Step 4: We count the number of black nodes $z$ with $inBlack[z]$ equal to 0. If there is exactly one node, we return that node as the answer. Otherwise, we return *null*.

It is easy to see that a node is black iff it can be reached from both $x$ and $y$. We now show that $z$ equals $join(x, y)$ iff it is the only black node with no black nodes pointing to it. Consider the subgraph of nodes that are colored black. If this graph is empty, then there is no node that is reachable from both $x$ and $y$ and therefore $join(x, y)$ is null as returned by our method. Otherwise, the subgraph of black nodes is nonempty. Due to acyclicity and finiteness of the graph there is at least one node with $inBlack$ equal to 0. If there are two or more nodes with $inBlack$ equal to 0, then these are incomparable upper bounds and therefore $join(x, y)$ does not exist. Finally, if $z$ is the only black node with $inBlack$ equal to 0, then all other black nodes are reachable from $z$ because all black nodes are reachable from some black node with $inBlack$ equal to 0. Therefore, $z$ is the least upper bound of $x$ and $y$.

### 2.3.2 Poset Relation Representation with Adjacency List

Now assume that the adjacency list of $x$ includes all elements that are greater than or equal to $x$, i.e., it encodes the poset relation ($e_{\leq}$). Generally, the following optimization is made for representation of adjacency lists. We keep the adjacency list in a sorted order. This enables us to search for an element in the list faster. For example, if we are looking for node 4 in the list of node 1 sorted in ascending order and we encounter the node 6, then we know that node 4 cannot be present in the list. We can then stop our search at that point instead of looking through the whole list. This optimization result in a better average case complexity.

Let us now explore complexity of various queries in this representation. To check that $y$ covers $x$, we need to check that $x \leq y$ and for any $z$ different from $x$ and $y$ such that $x \leq z$, $z \leq y$ is not true. This can be easily done by going through the adjacency lists of all nodes that are in the adjacency list of $x$ and have labels less than $y$ in $O(n + |e_{\leq}|)$ time.

Checking $x \leq y$ is trivial, so we now focus on computing $join(x, y)$.

- Step 0: We take the intersection of the adjacency lists of nodes $x$ and $y$. Since lists are sorted, this step can be done in $O(u_x + u_y)$ time where $u_x$ and $u_y$ are the size of the adjacency lists of $x$ and $y$, respectively. Call this list $U$.
- Step 1: If $U$ is empty, we return $null$. Otherwise, let $w$ be the first element in the list.
- Step 2: if $U$ is contained in the adjacency list of $w$, we return $w$; else, we return $null$.

A further optimization that can lead to reduction in space and time complexity for dense posets is based on keeping intervals of nodes rather than single nodes in adjacency lists. For example, if the adjacency list of node 2 is $\{4, 5, 6, 8, 10, 11, 12\}$, then we keep the adjacency list as three intervals $\{4-6, 8-8, 10-12\}$. We leave the algorithms for computing joins and meets in this representation as an exercise.

## 2.4 Skeletal Representation

Of all operations performed on posets, the most fundamental is $leq(x, y)$. We have observed that by storing just the cover relation, we save on space but have to spend more time to check $leq(x, y)$. On the other hand, keeping the entire poset relation can be quite wasteful as the adjacency lists can become large as the number of nodes in the poset increase. We now show a representation that provides a balance between the cover relation and the poset relation. This representation, called skeletal representation, assumes that the poset is decomposed into $k$ chains for some $k$. Algorithms for decomposition of posets into minimum number of chains is discussed in Chapter 3. For skeletal representation, we do not require the decomposition to be minimal.

Let there be $k$ chains in some decomposition of the poset. All elements in any chain are totally ordered and we can number them from 1 to the total number of elements in the chain. Thus, a node is labeled as a tuple (chain number, rank in the chain). Now to store the poset relation, for each element $y \in P$ with label $(j, r_y)$ we set $y.V$ as follows:

$y.V[i]$ = the rank of the largest element in chain $i$ which is less than or equal to $y$.

If no element in chain $i$ is less than or equal to $y$, then we set $y.V[i]$ to 0.

Now, given any element $x = (i, r_x)$, it is clear that $x \leq y$ iff $r_x \leq y.V[i]$. Thus, $leq(x, y)$ can be determined in constant time.

Just as we kept $y.V[i]$ as the rank of the largest element on chain $i$ which is less than or equal to $y$, we can keep the dual vector $F$, defined as follows:
$y.F[i]$ = the rank of the smallest element in chain $i$ which is greater than or equal to $y$ ($\infty$ if no such element).

Given a skeletal representation of a poset, one can easily generate adjacency lists for the poset relation. Assume that we are given $F$ vectors for each node. We generate the adjacency lists of the nodes in reverse topological order guaranteeing that if $y$ is less than $z$, then the adjacency list of $z$ will be generated before that of $y$. To generate the adjacency list of $y$, we simply need to take union of all nodes and their adjacency lists given by the vector $y.F$.

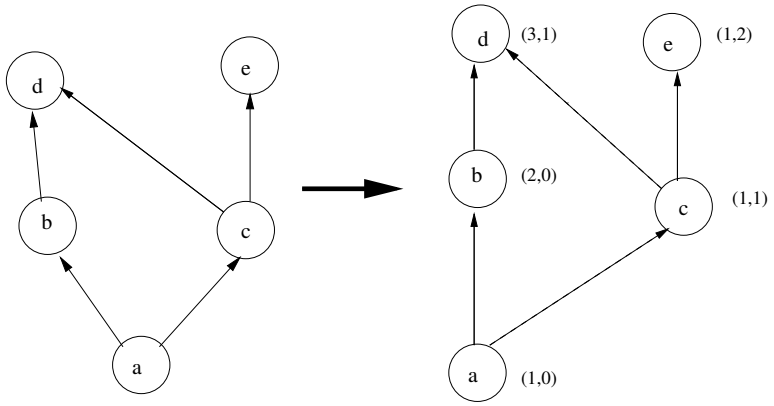



Figure 2.2: Vector clock labeling of a poset

An example of a poset $V$ labels is shown in Figure 2.2. Skeletal representation is called the *Fidge and Mattern Vector clock* in distributed systems. This representation is specially suited for distributed systems as it can be computed online. In many distributed computing applications such as distributed monitoring, debugging, and recovery, one needs to timestamp events so that the following property holds

$$e \rightarrow f \equiv e.v < f.v$$

The algorithm for doing this for process $P_j$ is given in Figure 2.4. The algorithm is described by the initial conditions and the actions taken for each event type. A process increments its own component of the vector clock after each event. Furthermore, it includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by taking a componentwise maximum with the vector clock included in the message.

Thus the skeletal representation of the poset is in fact just $k$ queues of vectors, each queue representing vector clock timestamps of all events in a process.

## 2.5 Matrix Representation

Like a directed graph, a poset can also be represented using a matrix. If there is a poset $P$ with $n$ nodes, then it can be represented by an $n \times n$ matrix, $A$. As before, we can either represent the cover relation or the poset relation through the matrix. Since in both cases we would be using $O(n^2)$ space, we generally use a matrix representation for the poset relation. Thus, for $x_i, x_j \in P$, $x_i \leq x_j$ iff $A[i,j] = 1$. The main advantage of using the matrix representation is that it can answer the query "Is $x \leq y$?" in $O(1)$ time.

## 2.6 Dimension Based Representation

The motivation for the dimension based representation comes from the fact that in the vector clock algorithm, we require $O(k)$ components in the vector. The minimum value of $k$ is bounded by the smallest number of chains required to decompose the poset. As we will see in Chapter **??**, Dilworth's theorem shows that the least number of chains required equals the width of the poset.

```
P_j::
 var
     v: array[1...N] of integer
          initially (∀i : i ≠ j : v[i] = 0) ∧ (v[j] = 1);

 send event :
     v[j] := v[j] + 1;

 receive event with vector u
     for i := 1 to N do
          v[i] := max(v[i], u[i]);
     v[j] := v[j] + 1;

 internal event
     v[j] := v[j] + 1
```

Figure 2.3: Vector clock algorithm

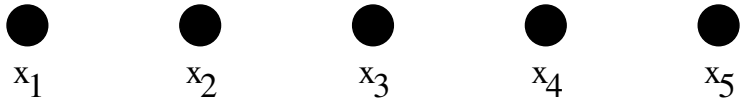



Figure 2.4: An antichain of size 5

So the natural question to ask is if there exists an alternate way of timestamping elements of a poset with vectors that requires less components than the width of the poset and can still be used to answer ahe query of the form $leq(x, y)$.

Surprisingly, in some cases, we can do with fewer components. For example, consider the poset shown in Figure 2.4. Since the poset is just an antichain, the minimum number of chains required to cover the poset is 5. So for using the vector clock representation, we require 5 components in the vector. But consider the following two linear extensions of the poset : $x_1, x_2, x_3, x_4, x_5$ and $x_5, x_4, x_3, x_2, x_1$. Using the ranks of the elements in these two linear extensions, we get the following vectors : $(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)$. These vectors have one-to-one correspondence with the poset order, i.e., they form an anti-chain. So in this case we can get away with just 2 components in the vector. This idea of using linear extensions of the poset to assign a vector to each element can be generalized to any poset. The minimum number of linear extensions required is called the *dimension* of the poset. Determining the dimension of a given poset is NP-hard whereas determining the width of a poset is computationally easy. For this reason, the dimension based representation is generally not used in distributed systems. The concept of dimension and related results are described in Chapter 15. The important point to note is that the dimension may be smaller than the width of the poset.

## 2.7 Irreducibles and Dissectors

In this section we briefly consider algorithms for determining irreducibles and dissectors of a poset. First assume that we have the adjacency list representation of the lower cover relation. To determine whether $x$ is join-irreducible we first compute $LC(x)$, the set of lower covers of $x$. Then, we use the following procedure to check if $x$ is join-irreducible .

- Step 1: If $LC(x)$ is singleton, return true;
- Step 2: If $LC(x)$ is empty, then if $x$ is the minimum element of the poset, return false, else return true;
- Step 3: If $join(LC(x))$ exists, return false, else return true;

The complexity of this procedure is dominated by the third step which takes $O(n + e_{\prec})$ in the worst case. The proof of correctness of the procedure is left as an exercise. To check if $x$ is an upper dissector, we use the following procedure.

- Step 1: Consider the set $W := P - U[x]$.
- Step 2: If $W$ has a maximum, return true, else return false.

This can again be done in $O(n + e_{\prec})$ time.

## 2.8 Infinite Posets

We now consider representation of (countably) infinite posets in a computer. Since computers have finite memory, it is clear that we will have to use finite representation for these posets. We define a p-diagram (poset diagram) as $(V, F, R, B)$ where

- $V$ is a set of vertices,
- $F$ (forward edges) is a subset of $V \times V$,
- $R$ (recurrent vertices) is a subset of $V$,
- $B$ (backward edges) is a subset of $R \times R$,

with the following constraints :

- (P0) $F$ is acyclic, i.e., $(V, F)$ is a directed acyclic graph
- (P1) If $u$ is a recurrent vertex and $(u, v) \in F$ or $(u, v) \in B$, then $v$ is also recurrent.

Thus, a p-diagram is a finite directed acyclic graph $(V, F)$ together with a set of *back-edges* given by $B$ and a subset $R$ denoting the vertices that appear an infinite number of times. Each p-diagram represents an infinite poset $(X, \leq)$ defined as follows. $X$ consists of two types of elements. First, for all vertices $u \in V$ we have elements $u^1$ in $X$. Further, for all vertices $u \in R$, we add an infinite number of elements $\{ u^i | i \geq 2\}$. It is clear that when $R$ is empty we get a finite poset and when $R$ is nonempty we get an infinite poset. We now define $\leq$ relation of the poset based on the given

p-diagram. The relation $\leq$ is the smallest reflexive transitive relation that satisfies:
(1) if $(u, v) \in F$ and $u \in R$, then $\forall i : u^i \leq v^i$, (2) if $(u, v) \in F$ and $u \notin R$, then $u^1 \leq v^1$, and (3) if $(v, u) \in B$, then $v^i \leq u^{i+1}$.

We now have the following theorem.

**Theorem 2.2** *$(X, \leq)$ as defined above is a reflexive poset.*

**Proof:** We show that $\leq$ is reflexive, transitive and antisymmetric. It is reflexive and transitive by construction. We now show that $\leq$ as defined above does not create any nontrivial cycle (cycle of size greater than one). Any cycle in elements with the same index would imply cycle in $(V, F)$. Thus, any cycle would have elements with at least two indices. However, $\leq$ never makes a higher indexed element smaller than a lower indexed element.

■


Figure 2.5: Trivial examples of d-diagrams

We also define a visual representation of p-diagrams. The recurrent vertices in the p-diagram are represented by hollow circles and the non-recurrent vertices through filled circles. The forward edges are represented by solid arrows and the shift-edges by dashed arrows. Let us now consider some examples.

1. The set of natural numbers with the usual order can be modeled using the p-diagram shown in Figure 2.8(i), where
   $V = \{u\}$; $F = \{\}$; $B = \{(u, u)\}$; $R = \{u\}$.

   Note that this poset has infinite height and width equal to 1.

2. The set of natural numbers with no order can be modeled as shown in Figure 2.8(ii) with
   $V = \{u\}$; $F = \{\}$; $B = \{\}$; $R = \{u\}$.

3. Figure **??** shows an example of a p-diagram along with the corresponding infinite poset. This p-diagram captures infinite behavior of a distributed computation. Along with each process, the local variables $x, y, z$ on processes $P_1, P_2$ and $P_3$ are also listed. For each event the value of the local variable is listed.

We now give some examples of posets that cannot be represented using p-diagrams.

1. The set of all integers (including negative integers) under the usual order relation cannot be represented using a p-diagram. The set of integers does not have any minimal element. Any poset defined using a p-diagram is well-founded.

2. Consider the set of all natural numbers with the order that all even numbers are less than all the odd numbers. This poset cannot be represented using p-diagram. In this poset, the upper covers of an element may be infinite. For example, the number 2 is covered by infinite number of elements. In a p-diagram, an element can only have a bounded number of lower (and upper covers).

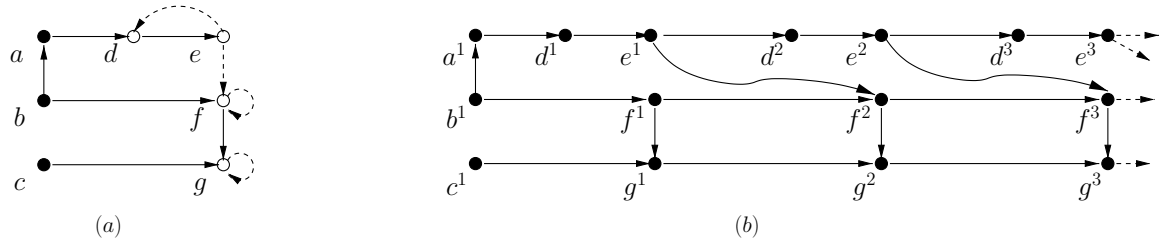



Figure 2.6: (a) A p-diagram and (b) its corresponding infinite poset

3. Consider the poset of two-dimensional vectors with natural coordinates. Define $(x_1, y_1) \leq (x_2, y_2)$ iff $(x_1 \leq x_2) \wedge (y_1 \leq y_2)$. This poset can be visualized as a grid. It can be shown that there is no p-diagram for this poset. (Why?)

The following properties of a p-diagram posets are easy to show.

**Lemma 2.3** *A poset $P$ defined by a p-diagram has the following properties:*

1. *$P$ is well-founded.*

2. *There exists a constant $k$ such that the size of the set of upper covers and lower covers of every element is bounded by $k$.*

From a distributed computing perspective, our intention is to provide a model for an infinite run of a distributed system which is eventually periodic. Runs of a distributed computation generated by a finite number of processes have finite width. The following lemma characterizes those p-diagrams for which the corresponding posets have finite width.

**Lemma 2.4** *A poset $P$ defined by a p-diagram has finite width iff for every recurrent vertex there exists a cycle in the graph $(R, F \cup B)$.*

**Proof:** If every recurrent vertex $v$ is in a cycle then we get that the set

$$\{v^i | i \geq 1\}$$

is totally ordered. This implies that the poset can be decomposed into a finite number of chains and therefore has a finite width. Conversely, if there exists any recurrent vertex $v$ that is not in a cycle, then $v^i$ is incomparable to $v^j$ for all $i, j$. Then, the set

$$\{v^i | i \geq 1\}$$

forms an antichain of infinite size. Thus, $P$ has infinite width.

■

Algorithms to check whether $x \leq y$, or $x$ covered by $y$ are left as an exercise.

## 2.9 Problems

1. Design algorithms for performing the common poset operations for the matrix representation and analyze their time complexity.
2. Give efficient algorithms for calculating the join and meet of the elements of a poset using the implicit representation.
3. Give parallel and distributed algorithms for topological sort.
4. Give parallel and distributed algorithms for computing the poset relation from the cover relation.
5. Give parallel and distributed algorithms for computing $join(x, y)$ in various representations.
6. (Open) How many different (non-isomorphic) posets can be defined on $n$ elements?
7. (Open) Given two posets $P$ and $Q$, what is the complexity of checking whether they are isomorphic?
8. Give an efficient algorithm that takes a poset as input and returns all its irreducibles and dissectors.
9. (Open) Give a polynomial-time algorithm to check if an arbitrary finite directed acyclic graph is a valid Hasse diagram.
10. Give efficient algorithms to determine $join(x, y)$, $leq(x, y)$ and $isCoveredBy(x, y)$ for elements $x$ and $y$ in a p-diagram.
11. Give an algorithm to compute the width of a p-diagram.

## 2.10 Bibliographic Remarks

The reader is referred to the book by Cormen, Leiserson, Rivest and Stein[**?**] for details on topological sort and adjacency representation of graphs. The dimension based representation of posets is due to Dushnik and Miller[**?**]. The discussion of p-diagrams is taken from the paper by Agarwal, Garg and Ogale[**?**].

# Chapter 3

# Dilworth's Theorem

## 3.1 Introduction

Of all the results in lattice theory, perhaps the most famous is Dilworth's theorem for decomposition of a poset. Dilworth's theorem belongs to the special class of results, called *min-max* results, which relate a maximal value in a structure to a minimal value. Dilworth's theorem states that the minimum number of chains a poset can be partitioned into equals the maximum size of an antichain. In this chapter, we cover this result and associated algorithms.

## 3.2 Dilworth's Theorem

We first present a proof of Dilworth's theorem due to Galvin (1994).

**Theorem 3.1** *[Dil50] [Dilworth's Theorem]Any finite poset $P = (X, \leq)$ of width $w$ can be decomposed into $w$ chains. Furthermore, $w$ chains are necessary for any such decomposition.*

**Proof:** We begin by proving that $w$ chains are necessary and that it is not possible to decompose the poset into less than $w$ chains. Consider the longest antichain (of length $w$). Each element of this antichain must belong to a seperate chain in the chain decomposition of the poset and hence we need at least $w$ chains.

We now show that $w$ chains are indeed sufficient to completely partition the poset. The proof is by induction on $n$, the size of the poset. The base case $n = 1$ is simple because a poset containing one element can be partitioned into one chain containing that element.

Consider a poset $P$ of size $n$. Let $x$ denote a maximal element of $P$. Consider the subposet $P' = (X - \{x\}, \leq)$. Since this is a poset of size less than $n$, by the induction hypothesis, we can decompose it into $t$ disjoint chains where $t$ is the width of $P'$. Let $C_1, ..., C_t$ denote the chains obtained after decomposition. We are required to prove that either $P$ has an antichain of size $t+1$ or that $P$ is a union of $t$ chains.

Let $a_i$ be the maximal element of $C_i$ which is part of some antichain of size $t$. The set $A = \{a_1, ..., a_t\}$ forms an antichain. To see this, consider an antichain containing $a_j$. There exists $b_i \in C_i$ which belongs to this antichain because any antichain of size $t$ must have exactly one element from

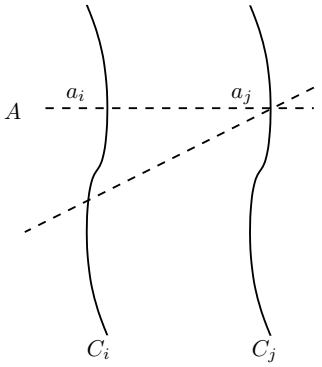



Figure 3.1: Proof of A being an antichain.

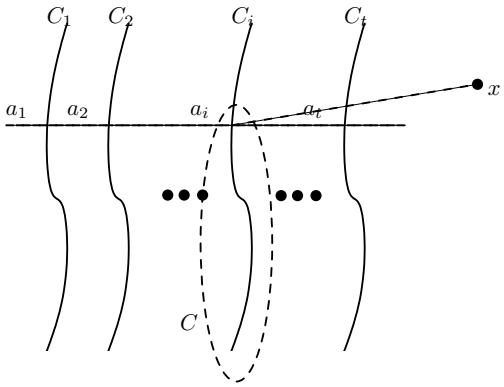



Figure 3.2: Decomposition of $P$ into $t$ chains.

each chain $C_i$. (see Figure 3.1). Since $a_i$ is the maximum element in the chain $C_i$ which is part of some antichain of size $t$, $b_i \leq a_i$. Therefore, $b_i || a_j$ implies that $a_i \not\leq a_j$.

We now compare $x$ with all the elements of $A$. If $A \cup \{x\}$ forms an antichain, then the width of $P$ is $t+1$ and the chain decomposition of $P$ is $C_1, ..., C_t, \{x\}$. Otherwise, there exists $a_k \in C_k$ such that $a_k < x$ for some $k$. The set $C = \{y \in C_k | y \leq a_k\})$ forms a chain in $P$. Since $a_k$ is the maximal element in $C_k$ is a part of some $t$-sized antichain in $P$, the width of $Q = (X - C, \leq)$ is less than $t$. Then by the induction hypothesis we can partition $Q$ into $t-1$ chains. Hence along with $C$ this forms a decomposition of $P$ into $t$ chains (Figure 3.2).

Any partition of a poset into chains is also called a **chain cover** of the poset.

■

## 3.3 Appreciation of Dilworth's Theorem

This was one of our first non-trivial proofs in the book. A delightful result such as Dilworth's theorem must be savored and appreciated. We now look at a few questions that naturally arise out of the above proof which will help the reader understand the proof better.

### 3.3.1 Special Cases

For any non-trivial proof, the reader should try out special cases. For example, in Dilworth's theorem, one may consider the special case of posets of height 1 (i.e., a bipartite graph). In this case, one obtains the Konig-Egervary theorem discussed in Chapter **??**.

### 3.3.2 Key Ingredients

Look for lemmas which can be abstracted out of the proof and can be reused in other proofs. In the proof of Dilworth's Theorem, the key ingredient was to prove that the set $A$ is an antichain. This gives us the following result.

**Theorem 3.2** *Let $A_n$ be the set of all antichains of width $w$ in a finite poset $P$ of width $w$. Then, $A_n$ is a lattice under the relation*

$$Y \ll Z \iff \forall y \in Y, \exists z \in Z : y \leq z$$

**Proof:** Let $\{C_1, C_2, ..., C_n\}$ be a decomposition of $P$ into $n$ chains. It is clear that each antichain $Y \in A_n$ consists of exactly one element from each chain. Let $Y[i]$ denote the element in $Y$ from chain $C_i$. Given $Y, Z \in A_n$, we define a set formed by

$$W[i] := min(Y[i], Z[i])$$

We first show that $W \in A_n$. It is sufficient to show that $W[i]||W[j]$ for any $i$ and $j$. By symmetry it is sufficient to show that $W[i] \not< W[j]$. Without loss of generality, let $W[i] = Y[i]$. If $W[j] = Y[j]$, then $W[i] \not< W[j]$ follows from $Y \in A_n$. If $W[j] = Z[j]$, then $W[i] \not< W[j]$ follows from $Z[j] \leq Y[j]$ and $Y[i] \not< Y[j]$.

We now show that $W$ is the greatest lower bound in $(A_n, \ll)$. To that end, we first show that $Y \ll Z \equiv \forall i : Y[i] \leq Z[i]$. The backward direction follows from the definition of $\ll$. For the forward direction, we do a case analysis on $Y[i]$ and $Z[i]$. $Y[i]$ and $Z[i]$ cannot be concurrent because they are from the same chain. If $Z[i] < Y[i]$, then from $Y \ll Z$, we know that there exists $k$ such that $Y[i] \leq Z[k]$. $Z[k] \neq Z[i]$ because $Z[i] < Y[i]$. From $Z[i] < Y[i]$ and $Y[i] \leq Z[k]$, we get that $Z[i] < Z[k]$. But this contradicts the fact that $Z$ is an antichain. Therefore, $Y[i] \leq Z[i]$ holds.

From $Y \ll Z \equiv \forall i : Y[i] \leq Z[i]$, it follows that $W$ is the greatest lower bound of $Y$ and $Z$ in $(A, \ll)$.

A dual argument holds for the supremem of $Y$ and $Z$.

■

### 3.3.3 Other Proofs

To appreciate a non-trivial theorem, one must look for alternative proofs. For example, we used induction on the size of the poset in our proof. Is there a proof that uses induction on the size of the cover relation? Harzheim [**?**] has given such a proof. Here we give another proof which is based on removing a chain instead of an element as in Galvin's proof. This is closer to Dilworth's original proof.

**Proof:** Let $P$ be a poset of width $k$. For induction, we assume that any poset with smaller cardinality of width $w$ or less can be partitioned into $w$ chains. If $P$ is an empty relation then (i.e.

has no comparable elements), then $P$ has exactly $k$ elements and can be trivially decomposed into $k$ chains. Otherwise, consider a chain $C = \{x, y\}$ where $x$ is a minimal element and $y$ a maximal element of $P$. If the width of $P - C$ is less than $k$, we are done because we can use that partition combined with $C$ to get the desired partition. If the width of $P - C$ is $k$, then there exists an antichain $A$ of size $k$. We consider the subposets $U[A]$ and $D[A]$. Since $x \notin U[A]$ and $y \notin D[A]$, we know that both $U[A]$ and $D[A]$ are strictly smaller than $P$. By induction, each of these sets can be partitioned into $k$ chains and since $U[A] \cap D[A] = A$, we can combine these chains to form the partition of the original set.

■

### 3.3.4 Duals of the Theorem

In lattice theory, one must also explore duals of any non-trivial theorem. For Dilworth's theorem, we get the following result.

**Theorem 3.3** *[?] Any poset $P = (X, \leq)$ of height $h$ can be decomposed into $h$ antichains. Furthermore, $h$ antichains are necessary for any such decomposition.*

**Proof:** All minimal elements of a poset form an antichain. When we delete this antichain, the height of the poset decreases by exactly one. By continuing this procedure we get the desired decomposition. The necessity part follows from the fact that no two elements of a chain can be in the same antichain.

■

### 3.3.5 Generalization of Dilworth's Theorem

We can generalize Dilworth's theorem in the following manner:

1. Go from finite posets to infinite ones.
2. Generalize the theorem to directed graphs.
3. Decompose the poset into other structures, like $k-$families. A $k-$family in a poset $P$ is a subposet containing no chains of size $k + 1$. So an antichain forms a $1-$family. We can try generalizing the dual of Dilworth's theorem to this structure.

Different proofs and their generalization can be found in the literature. We refer the interested reader to [Wes04].

### 3.3.6 Algorithmic Perspective

In practical applications, it may not be sufficient to prove existence of a chain partition. One may be required to compute it. Thus, from an algorithmic perspective one can look at the following questions:

1. How can we convert the proof of the theorem into an algorithm for chain decomposition of a poset?

2. Given any algorithm for chain decomposition, we analyze its time and space complexity. The natural question to ask is if we can improve the algorithm complexity or prove lower bounds on the time and space complexity.

3. Given any sequential algorithm, one can study variations of the algorithm that cater to different needs. A *parallel* algorithm tries to speed up the computation by using multiple processors. A *distributed* algorithm assumes that the information about the poset is distributed across multiple machines. An *online* algorithm assumes that the elements of the poset are presented one at a time and we maintain a chain decomposition at all times. The new element is inserted into one of the existing chains or into a new chain. The online algorithm cannot change the existing chain decomposition and as a result may require more chains than an offline algorithm. An *incremental* algorithm also assumes that elements of the poset are presented one at a time. The algorithm must compute the chain decomposition as we are presented with the new element. However, the algorithm is free to rearrange existing elements into chains when a new element arrives. The key issue here is to avoid recomputing chain decomposition for the entire poset whenever a new element is presented.

We will explore such algorithmic questions in Section 4.3. For now, we look at some applications of Dilworth's theorem.

## 3.4 Applications

In this section, we show the relationship between the problem of chain partition of a poset and matching in a bipartite graph. This relationship allows us to use the bipartite matching algorithms for chain-partition and vice versa.

### 3.4.1 Hall's Marriage Theorem

We first describe an application of Dilworth's theorem to combinatorics. We are given a set of $m$ girls $G = \{y_1, ..., y_m\}$ and $n$ boys $B = \{x_1, ..., x_n\}$ with $m < n$. Each girl $y_i$ has a set of boys $S(y_i)$ whom she is willing to marry. The aim is to find a condition on these sets so that it is possible to find a pairing of the girls and boys which enables each girl to get married. Each boy chosen from the set $S(y_i)$ must be distinct. This problem is also known as computing the set of distinct representatives.

A solution to this problem is given by the following theorem. Let $I \subseteq G$. Define $S(I) = \bigcup_{y_i \in I} S(y_i)$.

**Theorem 3.4** *[Hall's Theorem] The necessary and sufficient condition for the above set of distinct representatives problem is* $\forall I \subseteq G, |S(I)| \geq |I|$.

**Proof:** The necessary part of the condition is trivial because if $I$ is smaller than $S(I)$ then there will not be enough boys for pairing off with the girls. To prove that the above condition is sufficient, we assume that the given condition holds and view the $S$ relation as a poset of height 1 (Figure 3.3). This forms a bipartite poset. If we can split it into $n$ chains then we are done. Thus we need to prove that there is no antichain in the poset of size greater than $|B| = n$. Let $A$ be an antichain of the poset and $I = A \cap G$ (all the girls included in the antichain $A$). Since $A$ is an antichain, it does not contain an element belonging to $S(I)$. Hence $|A| \leq |I| + |B| - |S(I)| \leq |B|$ as $|I| \leq |S(I)|$

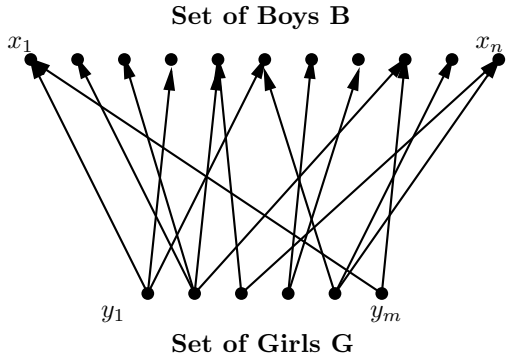



Figure 3.3: Hall's Marriage Theorem.

from Hall's condition. Thus the maximum size of an antichain is $|B| = n$. By Dilworth's theorem, the poset can be partitioned into $n$ chains. Since the antichain $B$ is maximal, each chain in such a partition must contain exactly one point of $B$. Thus we can take the chains in the partition of the poset which contain points of $G$. These give the desired pairings of boys and girls such that all the girls can get married. This proves the sufficiency of the given condition.

■

We now show applications of Dilworth's Theorem to bipartite matching. A **matching** in a graph $G = (V, E)$ is a subset of edges $E' \subseteq E$ such that no two edges in $E'$ have any vertex in common. A **vertex cover** in a graph is a subset of vertices $V' \subseteq V$ such that for every edge $(x, y) \in E$, either $x \in V'$ or $y \in V'$.

**Theorem 3.5** *[Kon31, Ege31] In a* bipartite *graph, the maximum size of a matching equals the minimum size of a vertex cover.*

**Proof:** Let $G = (L, R, E)$ be the bipartite graph with a total of $n = |L| + |R|$ vertices. View this bipartite graph as a poset with vertices in $L$ as the minimal vertices and $R$ as the maximal vertices. Any chain decomposition of this poset can have chains of size either 1 or 2. We note that

1. The chain decomposition with the smallest number of chains gives us the maximum matching.
2. Any antichain in the poset corresponds to an independent set in the graph. Thus the complement of the antichain corresponds to a vertex cover of the graph.

From Dilworth's theorem, the size of the biggest antichain equals the size of the smallest chain decomposition. Therefore, the size of the smallest vertex cover equals the size of the largest matching.

■

### 3.4.2 Strict Split of a Poset

The above construction shows that by using chain decomposition, one can solve the problem of bipartite matching. Now we consider the converse. Assume that we have an algorithm for bipartite matching. How do we use it for chain decomposition? This relationship between chain decomposition and bipartite matching is based on the idea of a strict split of a poset.

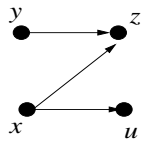



Figure 3.4: A poset of width 2 forcing an algorithm to use 3 chains for decomposition

**Definition 3.6 (Strict Split of a Poset)** *Given a poset $P$, the strict split of the poset $P$, denoted by $S(P)$ is a bipartite graph $(L, R, E)$ where $L = \{x^- | x \in P\}$, $R = \{x^+ | x \in P\}$, and $E = \{(x^-, y^+) | x < y \text{ in } P\}$.*

The following theorem is due to Fulkerson.

**Theorem 3.7** *Any matching in $S(P)$ can be reduced to a chain cover of $P$.*

**Proof:** We start with a trivial chain partition of $P$ with each element in $P$ being a chain. Thus initially we have $n$ chains for $n$ elements. We now consider each edge $(x^-, y^+)$ in the matching. Each such edge implies that $x < y$ in $P$ and we combine the chains corresponding to $x$ and $y$. Since there can be at most one edge incident on $x^-$, $x$ is the top element of its chain and similarly $y$ is the lowest element of the chain. Therefore, these chains can be combined into one chain. If there are $e$ edges in the matching, after this procedure we have a chain cover with $n - e$ chains.

■

Thus one can use any algorithm for bipartite matching to also determine a chain cover. There are many techniques available for bipartite macthing. One can use *augementing path* algorithms that start with the trivial matching and increase the size of the matching one edge at a time. In a graph of $n$ vertices and $m$ edges these algorithms take $O(mn)$ time. Karp and Ullman have given an $O(m\sqrt{n})$ algorithm for bipartite matching.

It is important to note that bipartite matching can also be solved by a more general technique of computing max-flows which itself is a special case of linear programming.

## 3.5 Online Decomposition of Posets

Dilworth's theorem states that a finite poset of width $k$ requires at least $k$ chains for decomposition [**?**]. However, constructive proofs of this result require the entire poset to be available for the partition. The best known *online* algorithm for partitioning the poset is due to Kierstead [**?**] which partitions a poset of width $k$ into $(5^k - 1)/4$ chains. The lower bound on this problem, due to Szemérdi (1982), states that there is no online algorithm that partitions all posets of width $k$ into fewer than $\binom{k+1}{2}$ chains.

We will focus on a special version of the problem where the elements are presented in a total order consistent with the poset order. Felsner [**?**] has shown that even for the simpler problem, the lower bound of $\binom{k+1}{2}$ holds. As an insight into the general result, we show how any algorithm can be forced to use 3 chains for a poset of width 2. Consider the poset given in Figure 3.4. Initially two incomparable elements $x$ and $y$ are presented to the chain decomposition algorithm. It is forced to assign $x$ and $y$ to different chains. Now an element $z$ greater than both $x$ and $y$ is presented. If the algorithm assigns $z$ to a new chain, then it has already used 3 chains for a poset of width

```
var
    B_1, ..., B_k: sets of queues
        ∀i : 1 ≤ i ≤ k, |B_i| = i
        ∀i : q ∈ B_i, q is empty

When presented with an element z:
    for i = 1 to k
        if ∃q ∈ B_i : q is empty or q.head < z
            insert z at the head of q
            if i > 1
                swap the set of queues B_{i-1} and B_i \ {q}
```

Figure 3.5: Chain Partitioning algorithm

2. Otherwise, without loss of generality, assume that the algorithm assigns $z$ to $x$'s chain. Then the algorithm is presented an element $u$ which is greater than $x$ and incomparable to $y$ and $z$. The algorithm is forced to assign $u$ to a new chain and hence the algorithm uses 3 chains for a poset of width 2.

Furthermore, Felsner showed the lower bound to be strict and presented an algorithm which requires at most $\binom{k+1}{2}$ chains to partition a poset. We present a simple algorithm, due to Agarwal and Garg, which partitions a poset into at most $\binom{k+1}{2}$ chains and requires at most $O(k^2)$ work per element.

The algorithm for online partitioning of the poset into at most $\binom{k+1}{2}$ chains is given in Figure 3.5. The algorithm maintains $\binom{k+1}{2}$ chains as queues partitioned into $k$ sets $B_1, B_2..., B_k$ such that $B_i$ has $i$ queues. Let $z$ be the new element to be inserted. We find the smallest $i$ such that $z$ is comparable with heads of one of the queues in $B_i$ or one of the queues in $B_i$ is empty. Let this queue in $B_i$ be $q$. Then $z$ is inserted at the head of $q$. If $i$ is not 1, queues in $B_{i-1}$ and $B_i \setminus \{q\}$ are swapped. Every element of the poset is processed in this fashion and in the end the non-empty set of queues gives us the decomposition of the poset.

The following theorem gives the proof of correctness of the algorithm.

**Theorem 3.8** *The algorithm in Figure 3.5 partitions a poset of width $k$ into $\binom{k+1}{2}$ chains.*

**Proof:** We claim that the algorithm maintains the followings invariant:
**(I)** For all i: Heads of all nonempty queues in $B_i$ are incomparable with each other.

Initially, all queues are empty and so the invariant holds. Suppose that the invariant holds for the first $m$ elements. Let $z$ be the next element presented to the algorithm. The algorithm first finds a suitable $i$ such that $z$ can be inserted in one of the queues in $B_i$.

Suppose the algorithm was able to find such an $i$. If $i = 1$, then $z$ is inserted into $B_1$ and the invariant is trivially true. Assume $i \geq 2$. Then $z$ is inserted into a queue $q$ in $B_i$ which is either empty or has a head comparable with $z$. The remaining queues in $B_i$ are swapped with queues in $B_{i-1}$. After swapping, $B_i$ has $i-1$ queues from $B_{i-1}$ and the queue $q$, and $B_{i-1}$ has $i-1$ queues from $B_i \setminus \{q\}$. The heads of queues in $B_{i-1}$ are incomparable as the invariant $I$ was true for $B_i$ before $z$ was inserted. The heads of queues in $B_i$ which originally belonged to $B_{i-1}$ are incomparable to each other due to the invariant $I$. The head of $q$, $z$, is also incomparable to the

heads of these queues as $i$ was the smallest value such that the head of one of the queues in $B_i$ was comparable to $z$. Hence, the insertion of the new element still maintains the invariant.

If the algorithm is not able to insert $z$ into any of the queues, then all queue heads and in particular, queue heads in $B_k$ are incomparable to $z$. Then $z$ along with the queue heads in $B_k$ forms an antichain of size $k+1$. This leads to a contradiction as the width of the poset is $k$. Hence, the algorithm is always able to insert an element into one of the queues and the poset is partitioned into fewer than $\binom{k+1}{2}$ chains.

■

Note that our algorithm does not need the knowledge of $k$ in advance. It starts with the assumption of $k = 1$, i.e., with $B_1$. When a new element cannot be inserted into $B_1$, we have found an antichain of size 2 and $B_2$ can be created. Thus the online algorithm uses at most $\binom{k+1}{2}$ chains in decomposing posets without knowing $k$ in advance.

This algorithm can be easily extended for online chain decomposition when the new elements are either maximal elements or minimal elements of the poset seen so far (see Problem **??**).

## 3.6 Exercises

1. Prove that in any poset of size $n \geq sr+1$, there exists a chain of length $s+1$ or an antichain of length $r+1$. Use the first part to prove the following result known as Erdos-Szekeres theorem: In any sequence $A$ of $n$ different real numbers where $n \geq sr+1$, either $A$ has a monotonically increasing subsequence of $s+1$ terms or a monotonically decreasing subsequence of $r+1$ terms.

2. Give an algorithm for chain decomposition of a poset based on Galvin's proof of Dilworth's Theorem.

3. In the online chain decomposition algorithm, when a new element $y$ arrives, we know that $y$ is a maximal element of the poset seen so far. Now consider a weaker property where we only know that $y$ is either a maximal element or a minimal element. Give an online chain decomposition algorithm for this case. (Hint: It is sufficient to maintain $\binom{2k+1}{2-1}$ chains in the online decomposition.)

4. Give an algorithm to find the maximum element of $A_n$ defined in Theorem 3.2.

## 3.7 Bibliographic Remarks

Dilworth's theorem is taken from [Dil50]. It is a prime example of min-max structure theorems that relate a minimum value (the number of chains to cover a poset) to a maximum value (the size of the biggest antichain). Dilworth's theorem is a special case of min-cut max-flow theorem which itself is a special case of the duality theorem for linear programming.

# Chapter 4

# Merging Algorithms

## 4.1 Introduction

In this chapter we describe two algorithms for determining the width of a given poset.

## 4.2 Algorithm Based on Reducing Sequences for Chain Partition

We are given a chain partition of a poset into $t$ disjoint chains $\{C_1, C_2, ..., C_t\}$. Our goal is to rearrange these chains into $t-1$ chains if possible. We call a sequence of elements

$$a_0, b_0, ..., a_s, b_s$$

a *reducing sequence* if

1. $a_0$ is the least element of some chain
2. $b_i$ is the immediate predecessor of $a_{i+1}$ in some chain
3. all $b_i$'s are distinct
4. for all $i$: $a_i > b_i$ in the partial order
5. $b_s$ is the greatest element of its chain.

We now show

**Theorem 4.1** *The reduction in the number of chains is possible iff a* reducing sequence *exists.*

**Proof:** If a reducing sequence exists, then one can (easily) reduce the total number of chains. We modify $C_s$ by including all elements of $C_{s-1}$ which are greater than or equal to $a_s$. We then modify $C_{s-1}$ by replacing the tail beginning with $a_S$ with all elements of $C_{s-2}$ which are greater than or equal to $a_{s-1}$. Continuing this procedure, the first chain is eliminated.

We now show the converse: if no reducing sequence exists then the number of chains cannot be reduced.

We define an alternating sequences to be

$$a_0, b_0, ..., a_s$$

with the first four properties of a *reducing sequence*.

Let $E$ be set of all $a$'s that are in some alternating sequence. Note that $E$ contains at least one point from each chain because the least element of each chain $C_i$ is part of some alternating sequence. Let $A$ be the set of maximal elements of $E$. Clearly, $A$ is an antichain. We now claim that $A$ consists of at least one element from from each chain or a reducing sequence exists. In the first case, we will have an antichain of size $t$.

Assume that $A$ does not contain any element from the chain $C$. Let $x$ be the maximal element of $C$ which is in $E$. Since $x \notin A$, there exists $y \in A$ such that $x < y$. Since $y \in E$, there exists an alternating sequence with $a_s = y$. Let $b_s = x$. If $x$ is not the greatest element of $C$, then we can let $a_{s+1}$ to be successor of $x$. But his implies that $x$ was not the maximal element of $C$ which is in $E$. Therefore, $x$ is the greatest element of $C$. Thus, we have a reducing sequence.

■

$1x_1$ $2x_2$ $3x_3$

## 4.3 Algorithm to Merge Chains in Vector Clock Representation

We are given a poset $R$ partitioned into $N$ chains, $R_i$, and we need to determine if there exists an antichain of size at least $K$. It follows from Dilworth's theorem that $R$ can be partitioned into $K-1$ chains if and only if there does not exist an antichain of size at least $K$. So our problem is now reduced to taking the $N$ chains $R_i$, $1 \leq i \leq N$, and trying to merge them into $K-1$ chains. The approach we take is to choose $K$ chains and try to merge them into $K-1$ chains. After this step we have $N-1$ chains left. We do this step ("choose $K$ chains, merge into $K-1$ chains") $N-K+1$ times. If we fail on any iteration, then $R$ could not be partitioned into $K-1$ chains. Thus there exists an antichain $c$ such that $|c| > K$. If we succeed then we have reduced $R$ to $K-1$ chains and there does not exist an antichain $c$ such that $|c| > K$.

The algorithm uses queues to represent chains. Each queue is stored in increasing order so the head of a queue is the smallest element in the queue.

The algorithm *FindAntiChain* (Figure 4.1) calls the *Merge* function $N-K+1$ times. The *Merge* function takes $K$ queues as input and returns $K-1$ queues if successful. If not, then an antichain has been found and is given by the heads of the returned queues.

**while** the number of queues is greater than $K-1$
  choose $K$ queues which have been merged least number of times
  merge $K$ queues into $K-1$ queues

Figure 4.1: Function that determines if an antichain of size $K$ exists

There are two important decisions in this algorithm. The first is how to choose the $K$ chains for the merge operation. The answer follows from classical merge techniques used for sorting. We choose the chains which have been merged the least number of times. (Alternatively, one can use the $K$ smallest queues. We have used the metric "merged least number of times" for convenience

in the complexity analysis.) This strategy reduces the number of comparisons required by the algorithm.

The second and more important decision is how to implement *Merge*. The merge is performed by repeatedly removing an element from one of the $K$ input chains and inserting it in one of the $K-1$ output chains. An element will be moved from the input to the output if it is smaller than an element on the head of some other input chain. The problem is deciding on which output chain to place the element.

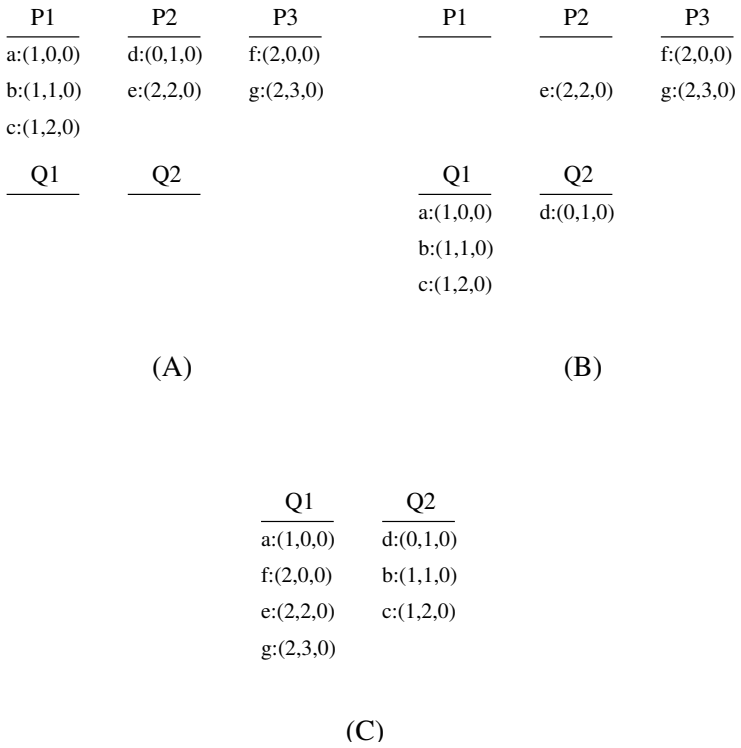



Figure 4.2: An example of a failed naive strategy. Diagram $A$ shows the initial configuration. Diagram $B$ shows the point at which the strategy fails: there is no where to insert $(2,0,0)$. Diagram $C$ shows that this example can be merged into two chains.

Note that this is trivial for $K = 2$, when two input queues are merged into a single output queue as is done in the merge sort. For $K > 2$, the number of output queues is greater than one, and the algorithm needs to decide which output queue the element needs to be inserted in. The simple strategy of inserting the element in any output queue does not work as shown in Figure 4.2 where there are three input queues $(P_1, P_2, P_3)$ which need to be merged into two output queues $(Q_1, Q_2)$. Suppose we use a simple strategy which results in the operations listed below. Initially $Q_1$ and $Q_2$ are empty. Each operation moves an element from the head of $P_1$, $P_2$ or $P_3$ to one of the two output queues. Note that we can only move the head of $P_i$ if it is smaller than the head of some other queue $P_j$. The operations we perform are:

1. $(1,0,0) < (2,0,0)$. So move $(1,0,0)$ to some output queue, say $Q_1$.
2. $(0,1,0) < (1,1,0)$. So move $(0,1,0)$ to some output queue, say $Q_2$.

3. $(1,1,0) < (2,2,0)$. So move $(1,1,0)$ to some output queue, say $Q_1$.

4. $(1,2,0) < (2,2,0)$. So move $(1,2,0)$ to some output queue, say $Q_1$.

5. $(2,0,0) < (2,2,0)$. So move $(2,0,0)$ to some output queue, but which one?

Notice that we have worked ourselves into a corner because when we decide to move $(2,0,0)$ there is no output queue in which we can insert it. The output queues must be sorted since they represent chains. This is done by inserting the elements in increasing order, but $(2,0,0)$ is not larger than the tails of any of the output queues. Thus we have nowhere to insert it.

This situation does not imply that the input queues cannot be merged. In fact in this case they can be merged into two queues as shown in Figure 4.2(C). It merely implies that we did not intelligently insert the elements in to the output queues. The function $FindQ$ chooses the output queue without running into this problem. A discussion of $FindQ$ is deferred until after we describe the details of the $Merge$ function.

The $Merge$ function compares the heads of each input queue with the heads of all other queues. Whenever it finds a queue whose head is less than the head of another queue, it marks the smaller of the two to be deleted from its input queue and inserted in one of the output queues. It repeats this process until no elements can be deleted from an input queue. This occurs when the heads of all the queues are incomparable, that is, they form an antichain. Note that it may be the case that some input queues are empty. If none are empty, then we have found an antichain of size $K$. The heads of the input queues form the antichain. If one or more are empty, the merge operation (which is not complete yet) will be successful. All that is left to do is to take the non-empty input queues and append them to the appropriate output queues. This is done by the *FinishMerge* function whose implementation is not described because it is straight forward.

The *Merge* algorithm is shown in Figure 4.3. Note that it only compares the heads of queues which have not been compared earlier. It keeps track of this in the variable *ac*, which is a set of indices indicating those input queues whose heads are known to form an antichain. Initially *ac* is empty. The $Merge$ algorithm terminates when either *ac* has $K$ elements or one of the input queues is empty.

The first *for* loop in *Merge* compares the heads of all queues which are not already known to form an antichain. That is, we compare each queue not in *ac* to every other queue. This avoids comparing two queues which are already in *ac*. Suppose $e_i = head(P_i)$ and $e_j = head(P_j)$ and inside the first *for* loop it is determined that $e_i < e_j$. This implies that $e_i$ is less than all elements in $P_j$. Thus, $e_i$ cannot be in an antichain with any element in $P_j$ and therefore cannot be in any antichain of size $K$ which is a subset of the union of the input queues. Thus, we can safely move $e_i$ to an output queue, which eliminates it from further consideration. The set *move* records which elements will be moved from an input queue to an output queue. The array *bigger* records the larger of the two elements which were compared. In this example, $bigger[i]$ equals $j$, implying that the head of $P_j$ is bigger than the head of $P_i$. This information is used by $FindQ$ to choose the output queue where the head of $P_i$ will be inserted.

The second *for* loop just moves all elements in *move* to an output queue. Consider the state of the program just before the second *for* loop begins. If the head, $e$, of an input queue is not marked to be moved, then $e$ is not less than the head of any other input queue or else it would have been marked to be moved. This implies that any two elements which are not moved are concurrent, which in turn implies that the set of heads which are not moved form an antichain. This antichain is recorded in *ac* for the next iteration of the *while* loop.

```
function Merge(P_1, ..., P_K: queues): Q_1, ..., Q_{K-1}: queues;
const all = {1,...K};
var  ac,move: subsets of all;
     bigger: array[1..K] of 1..K;
     G: initially any acyclic graph on K vertices;
begin
     ac := ∅;
     while (|ac|≠ K  ∧  ¬(∃i : 1 ≤ i ≤ K : empty(P_i))) do
          move := {};
          for i ∈ all - ac and j ∈ all do
               if head(P_i) < head(P_j) then
                    move := move ∪ {i};
                    bigger[i] := j;
               end;
               if head(P_j) < head(P_i) then
                    move := move ∪ {j};
                    bigger[j] := i;
               end;
          endfor
          for i ∈ move do
               dest := FindQ(G,i,bigger[i]);
               x := deletehead(P_i);
               insert(Q_dest, x);
          endfor
          ac := all - move;
     endwhile
     if (∃i : empty(P_i)) then
          FinishMerge(G, P_1, ... P_K, Q_1, ... Q_{K-1});
          return (Q_1, ..., Q_{K-1});
     else
          return (P_1, ..., P_K);// merge not possible
     end
end
```

Figure 4.3: Generalized Merge Procedure for deposets

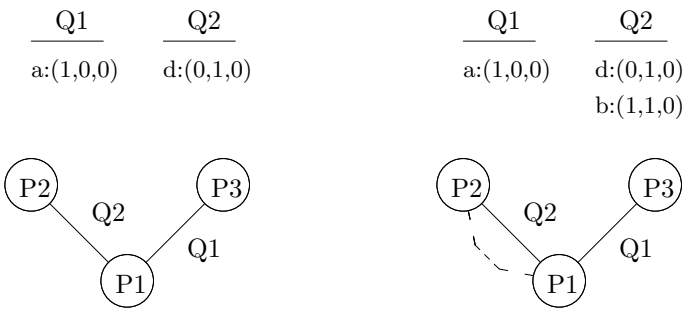



Figure 4.4: Using a queue insert graph to find the output queue

We now return to describing how *FindQ* works. The formal description of *FindQ* is given in Figure 4.5. Given the queue which contains the element to be moved, and the queue with which this element was compared (it must be smaller than the head of another queue in order to move it), the procedure *FindQ* determines which output queue to use. The *FindQ* function takes three parameters:

| | | |
|---|---|---|
| $G$ | : | an undirected graph called *queue insert graph* |
| $i$ | : | the input queue from which the element $x$ is to be deleted |
| $j$ | : | the queue in which all elements are bigger than $x$ |

A "queue insert graph" is used to deduce the queue in which the next element is inserted. It has $K$ vertices and exactly $K-1$ edges. An edge corresponds to an output queue and a vertex corresponds to an input queue. Therefore, each edge $(i,j)$ has a label, $label(i,j) \in \{1,\ldots,K-1\}$, which identifies the output queue it corresponds with. No labels are duplicated in the graph, thus each output queue is represented exactly once. Similarly, each input queue is represented exactly once.

An edge $(i,j)$ between vertex $i$ and vertex $j$ means that the heads of $P_i$ and $P_j$ are both bigger than the tail of $Q_{label(i,j)}$. The goal is to ensure that for any input queue (i.e. any vertex) there exists an output queue (i.e. an edge) in which the head of the input queue can be inserted. This constraint is equivalent to the requirement that every vertex has at least one edge adjacent to it. It is also equivalent to the requirement that the graph is a tree (i.e., acyclic) since there are $K$ nodes and $K-1$ edges.

*FindQ* uses the queue insert graph as follows. Consider the function call $FindQ(G,i,j)$. The element to be deleted is $e_i = head(P_i)$, and it is smaller than $e_j = head(P_j)$ (i.e., $bigger[i] = j$). *FindQ* adds the edge $(i,j)$ to $G$. Since $G$ was a tree before adding edge $(i,j)$, it now contains exactly one cycle which includes $(i,j)$. Let $(i,k)$ be the other edge incident on vertex $i$ which is part of this cycle (it is possible that $k = j$). *FindQ* deletes $(i,k)$ and adds $(i,j)$, and then sets the label of $(i,j)$ equal to the label of $(i,k)$. *FindQ* then returns the label associated with the new edge. This label indicates which queue $e_i$ will be inserted in.

Consider our previous example with the naive algorithm. It made a bad decision when it placed element $(1,1,0)$ on $Q_1$. Figure 4.4 shows the state of the queues and of the graph before and after the correct decision made by *FindQ*. Element $(1,1,0)$ is in $P_1$ and is less than the head of $P_2$. Thus the edge $(1,2)$ is added to the graph (dashed line in figure 4.4). It forms a cycle with the

other edge $(1, 2)$ which is labeled with $Q_2$. We copy this label to the new edge, delete the old edge and return $Q_2$, indicating that $(1, 1, 0)$ should be inserted in $Q_2$.

An important invariant of the queue insert graph is that given any edge $(i, k)$ and the output queue associated with it, the queue is empty or the tail of the queue is less than all elements in input queues $P_i$ and $P_k$. This is stated and proven in Lemma 4.2 and is used later to show that the output queues are always sorted.

**Lemma 4.2** *If $(i, k) \in G$, then*

$$empty(Q_{label(i,k)}) \ \vee \ (\forall e : e \in P_i \cup P_k : tail(Q_{label(i,k)}) \leq e)$$

**Proof:** Initially, the lemma holds since all output queues are empty. Now assume that the lemma holds and we need to insert $e_i = head(P_i)$ into output queue $Q_l$ where $l = FindQ(G, i, j)$ and $j = bigger[i]$. Since $j = bigger[i]$, we know $e_i \leq head(P_j)$. Since $P_i$ and $P_j$ are sorted, we know $e_i \leq e$ for any element $e \in P_i \cup P_k$. After moving $e_i$ to $Q_l$, the tail of $Q_l$ will be $e_i$ and the lemma will still hold.

■

```
function FindQ(G: graph; i,j:1..K): label;
    add edge (i, j) to G;
    (i, k) := the edge such that (i, j) and (i, k) are part of the same cycle in G;
    remove edge (i, k) from G;
    label(i, j) := label(i, k);
    return label(i, j);
end
```

Figure 4.5: function FindQ that finds the output queue to insert an element

The merge operation must produce sorted output queues. Lemma 4.3 proves that our algorithm meets this requirement.

**Lemma 4.3** *If the elements are inserted in the output queues using FindQ, then all output queues are always sorted.*

**Proof:** Initially all output queues are empty. Now assume each output queue is sorted and we need to move $e_i = head(P_i)$ to an output queue. The $FindQ$ procedure will return $Q_{label(i,j)}$, where $(i, j)$ is some edge in $G$. By Lemma 4.2, the tail of this queue is less than or equal to $e_i$. Thus after inserting $e_i$ in $Q_l$, $Q_l$ is still sorted. No other output queues are modified, thus the lemma still holds.

■

### 4.3.1 Overhead Analysis

In this section, we analyze the complexity based on the number of comparisons required by the algorithm, that is, the number of times the heads of two queues are compared in the *Merge* function. We prove an upper bound and a lower bound. The lower bound is proven by defining an adversary which produces a set of input queues that forces the merge algorithm to use at least $KMN$ comparisons, where $M$ is the number of elements in the largest input queue.

#### An Upper Bound

**Theorem 4.4** *The maximum number of comparisons required by the above algorithm is* $KMN(K+\log_\rho N)$ *where* $\rho = K/(K-1)$.

**Proof:** We first calculate the complexity of merging $K$ queues into $K-1$ queues. From the *Merge* algorithm it is clear that each element must be in *ac* before it passes to an output queue and it requires $K$ comparisons to be admitted to *ac*. Thus, if the total number of elements to be merged is $l$, then $l$ elements must pass through *ac* on their way to the output queue for a total of $Kl$ comparisons.

Initially $l \leq KM$ but the queues grow for each successive call to *Merge*. At this point, our technique of rotating the list of queues to be merged is useful. Let $level.i$ denote the maximum number of merge operations that any element in $P_i$ has participated in. The algorithm rotates *Qlist* to ensure that in each iteration the $K$ queues with the smallest level numbers will be merged. Initially, there are $N$ queues at level 0. Each of the first $N/K$ merge operations reduces $K$ queues with level 0 to $K-1$ queues with level 1. This pattern continues until there are $2K$ queues left, at which time the maximum level will be $log_\rho N$ where $\rho$ is the reducing factor and equals $K/(K-1)$. Merging the remaining $2K$ queues into $K-1$ queues adds $K$ more levels. Thus, the maximum level of any final output queue is $K + log_\rho N$. So there are at most $MN$ elements, each of which participates in at most $K + log_\rho N$ merge operations at a cost of $K$ comparisons per element per merge. Therefore the maximum number of comparisons required by the above algorithm is $KMN(K + log_\rho N)$.

■

Note that for the special case when $K = 2$, the complexity is $O(MN \log N)$. Further, if $M = 1$ and $K = 2$, this reduces to the well-known merge sort algorithm with the complexity of $O(N \log N)$ comparisons. Another noteworthy special case is when $K = N$. In this case, the complexity becomes $O(MN^2)$ which is also optimal.

#### A Lower Bound

In this section, we provide a lower bound on the number of comparisons required by any algorithm to solve the above problem.

**Theorem 4.5** *Let* $(P, <)$ *be any partially ordered finite set of size* $MN$. *We are given a decomposition of* $P$ *into* $N$ *sets* $P_1, ...P_N$ *such that each* $P_i$ *is a chain of size* $M$. *Any algorithm which determines if there exists an antichain of size* $K$ *must make at least* $\Omega(KMN)$ *comparisons.*

**Proof:** We use an adversary argument. Let $P_i[s]$ denote the $s^{th}$ element in the queue $P_i$. The adversary will give the algorithm $P_i$'s with the following characteristic:

$$(\forall i, j, s : P_i[s] < P_j[s+1])$$

Formally, on being asked to compare $P_i[s]$ and $P_j[t]$, $(s \neq t)$ the adversary uses:

**if** $(s < t)$ **then** return $P_i[s] < P_j[t]$
**if** $(t < s)$ **then** return $P_j[t] < P_i[s]$

Thus, the above problem reduces to $M$ independent instances of the problem which checks if a poset of $N$ elements has a subset of size $K$ containing pairwise incomparable elements. If the algorithm does not completely solve one instance then the adversary chooses that instance to show a poset consistent with all its answers but different in the final outcome.

We now show that the number of comparisons to determine whether any poset of size $N$ has an antichain of size $K$ is at least $N(K-1)/2$. The adversary will give that poset to the algorithm which has either $K-1$ or $K$ chains such that any pair of elements in different chains are incomparable. In other words, the poset is a simple union of either $K-1$ or $K$ chains. The adversary keeps a table *numq* such that *numq*[$x$] denotes the number of questions asked about the element $x$. The algorithm for the adversary is shown in Fig. 4.6.

```
var
        numq: array[1, ..., N] of integers, initially, ∀i : 1,...,N, numq[i] = 0;
/* number of questions asked about element x */

function  compare (x,y:elements)
        numq[x]++; numq[y]++;
        if (numq[x] = K − 1) then
            chain[x] := chain in which no element has been compared with x so far;
        if (numq[y] = K − 1) then
            chain[y] := chain in which no element has been compared with y so far;
        if (numq[x] < K − 1) or (numq[y] < K − 1) then
            return x||y;
        if (chain[x] ≠ chain[y]) then
            return x||y;
        else
            if x inserted earlier than y then return (x < y);
            else return (y < x);
end
```

Figure 4.6: Algorithm for the Adversary

If the algorithm does not ask $K-1$ questions about any element $x$, the adversary can produce a poset inconsistent with the answer of the algorithm. If the algorithm answered that no anti-chain of size $K$ exists, then the adversary can produce an antichain which includes one element from each of the $K-1$ chains and the element $x$. On the other hand, if the algorithm answered that an

antichain exists, then the adversary could put $x$ and all other elements for which $K-1$ questions have not been asked in $K-1$ chains.

Since each comparison involves two elements, we get that the algorithm must ask at least $N(K-1)/2$ questions for each level. Thus, overall, any algorithm must make at least $MN(K-1)/2$ comparisons.

■

It is easy to see that the lower bound is not tight. If we choose $M = 1$ and $K = 2$, we get the lower bound of $N/2$. However, the lower bound of $N \log N$ is well known for this case.

## 4.4 Bibliographic Remarks

The algorithm to merge $k$ chains into $k-1$ chains is taken from [TG97].

# Chapter 5

# Lattices

## 5.1 Introduction

Given a poset, the two most fundamental derived operations on the elements of the poset are the join and the meet operations. As we have seen earlier, join and meet operations may not exist for all subsets of the poset. This observation motivates the notion of lattices which occur in many different contexts. Let us recall the definition of a lattice from Chapter **??**.

**Definition 5.1 (Lattice)** *A poset $(X, \leq)$ is a* **lattice** *iff $\forall x, y \in X : x \sqcup y$ and $x \sqcap y$ exist.*

The first two posets in Figure 5.1 are lattices, whereas the third one is not. Note that in the third poset $b \sqcup c$ and $d \sqcap e$ do not exist. Although both $d$ and $e$ are upper bounds on the set $\{b, c\}$, neither $d$ nor $e$ is the least upper bound.

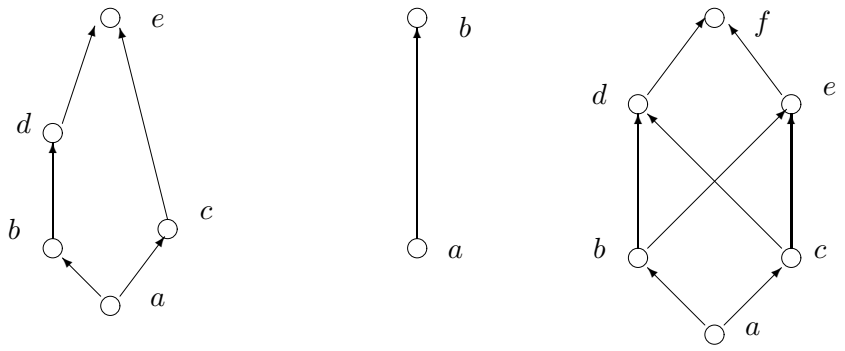



Figure 5.1: Only the first two posets are lattices.

If $\forall x, y \in X : x \sqcup y$ exists, then we call it a *sup semilattice*. If $\forall x, y \in X : x \sqcap y$ exists, then we call it an *inf semilattice*.

In our definition of lattice, we have required existence of *lub*'s and *glb*'s for sets of size two. This is equivalent to the requirement of existence of *lub*'s and *glb*'s for sets of finite size by using induction (see Problem 5.1).

We now give many examples of lattices that occur naturally.

- The set of natural numbers under the relation *divides* forms a lattice. Given any two natual numbers, the greatest common divisor (gcd) and the least common multiple (lcm) of those two numbers correspond to the sup and inf respectively.
- The family of all subsets of a set $X$, under the relation $\subseteq$ (i.e., the poset $(2^X, \subseteq)$) forms a lattice. Given any two subsets $Y, Z$ of $X$, the sets $Y \cup Z$ and $Y \cap Z$ (corresponding to sup and inf) are always defined.
- The set of rationals or reals forms a lattice with the $\leq$ relation. This is a totally ordered set and sup and inf correspond to the *max* and *min* for a finite subset.
- The set of n-dimenstional vectors over $\mathbb{N}$ with component-wise $\leq$ forms a lattice. Componentwise $\leq$ of two vectors $(x_1, x_2, ...x_n)$, $(y_1, y_2, ...y_n)$ is defined as:

  $$(x_1, x_2, ...x_n) \leq (y_1, y_2, ...y_n) \equiv \forall i : \ x_i \leq y_i.$$

  The sup and inf operations correspond to component-wise maximum and minimum.

It is important to note that a poset may be a lattice, but it may have sets of infinite size for which sup and inf may not exist. A simple example is that of the set of natural numbers under the natual $\leq$ relation. There is no sup of the set of even numbers, even though sup and inf exist for any finite set. Another example is the set of rational numbers, $Q$. Consider the set $\{x \in Q \mid x \leq \sqrt{2}\}$. The least upper bound of this set is $\sqrt{2}$ which is not a rational number. A lattice for which any subset has lub and glb defined is called a **complete** lattice. An example of a complete lattice is the set of real numbers extended with $+\infty$ and $-\infty$. We will discuss complete lattices in greater detail in Chapter **??**.

## 5.2 Sublattices

S is a sublattice of a given lattice $L = (X, \leq)$ iff it is non-empty, and:

$$\forall a, b \in S : sup(a, b) \in S \wedge inf(a, b) \in S.$$

Note that the $sup$ and $inf$ of any two elements in the sublattice $S$ must be the same as the $sup$ and $inf$ of those elements in the original lattice $L$.

For $S$ to be a sublattice, $S$ being a subset of a lattice is not sufficient. In addition to $S$ being a subset of a lattice, $sup$ and $inf$ operations must be inherited from the lattice.

### Examples

1. In Figure 5.2 the shaded elements in lattices $(i)$, and $(ii)$ form sublattices, while those in $(iii)$, and $(iv)$ do not.
2. Any one-element subset of a lattice is a sublattice. More generally, any chain in a lattice is a sublattice.
3. A subset $M$ of a lattice $\langle L; \leq \rangle$ may be a lattice in its own right without being a sublattice of $L$; see Figure 5.2$(iii)$ for an example. The shaded elements in $(iii)$ is not a sublattice because $a, b \in S$, however $a \sqcup b \notin S$.

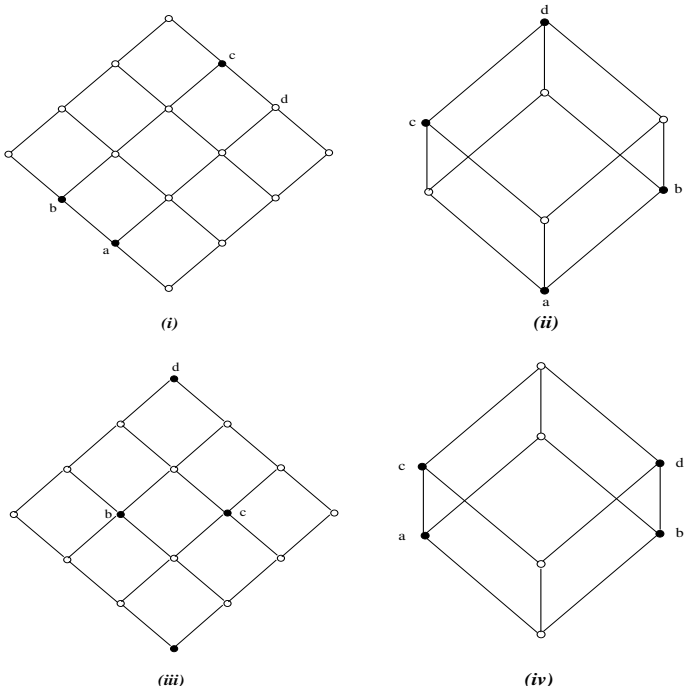



Figure 5.2: Sublattice Examples

## 5.3 Lattices as Algebraic Structures

We have so far looked at a lattice as a special type of poset, $P = (X, \leq)$, where the operator $\leq$ is reflexive, antisymmetric and transitive. We have defined the operations of $\sqcup$ and $\sqcap$ on lattices based on the given $\leq$ relation.

An alternative method of studying lattices is to start with a set equipped with $\sqcup$ and $\sqcap$ operator and define $\leq$ relation based on these operations. Consider any set $X$ with two algebraic operators $\sqcup$ and $\sqcap$. Assume that the operators satisfy the following properties:

| | | |
|---|---|---|
| $(L1)$ | $a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$ | (associativity) |
| $(L1)^\delta$ | $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$ | |
| $(L2)$ | $a \sqcup b = b \sqcup a$ | (commutativity) |
| $(L2)^\delta$ | $a \sqcap b = b \sqcap a$ | |
| $(L3)$ | $a \sqcup (a \sqcap b) = a$ | (absorption) |
| $(L3)^\delta$ | $a \sqcap (a \sqcup b) = a$ | |

**Theorem 5.2** *Let $(X, \sqcup, \sqcap)$ be a nonempty set with the operators $\sqcup$ and $\sqcap$ which satisfy $(L1)-(L3)$ above. We define the operator $\leq$ on $X$ by:*

$$a \leq b \equiv (a \sqcup b = b)$$

*Then,*

| | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| a | a | a ⊓ b | a ⊓ c | a ⊓ d | a ⊓ e | a ⊓ f | a ⊓ g |
| b | b ⊔ a | b | b ⊓ c | b ⊓ d | b ⊓ e | b ⊓ f | b ⊓ g |
| c | c ⊔ a | c ⊔ b | c | c ⊓ d | c ⊓ e | c ⊓ f | c ⊓ g |
| d | d ⊔ a | d ⊔ b | d ⊔ c | d | d ⊓ e | d ⊓ f | d ⊓ g |
| e | e ⊔ a | e ⊔ b | e ⊔ c | e ⊔ d | e | e ⊓ f | e ⊓ g |
| f | f ⊔ a | f ⊔ b | f ⊔ c | f ⊔ d | f ⊔ e | f | f ⊓ g |
| g | g ⊔ a | g ⊔ b | g ⊔ c | g ⊔ d | g ⊔ e | g ⊔ f | g |

Figure 5.3: Table notation for the algebra $(X, \sqcup, \sqcap)$.

- $\leq$ *is reflexive, antisymmetric and transitive*
- $sup(a, b) = a \sqcup b$
- $inf(a, b) = a \sqcap b$.

**Proof:** Left as an exercise.

■

The algebra $(X, \sqcup, \sqcap)$ can be represented by an $n \times n$ table, where $|X| = n$. It may seem as though two $n \times n$ tables are needed, one for $\sqcup$ and one for $\sqcap$. However, a single $n \times n$ table suffices, with one half used for $\sqcup$ and the other for $\sqcap$, as shown in Figure 5.3. By commutativity, $a \sqcup b = b \sqcup a$, and $a \sqcap b = b \sqcap a$, so populating both halves of the table for the same operation is redundant. Also, by absorption, $a \sqcup a = a \sqcap a = a$, so the diagonal elements of the table agree for $\sqcup$ and $\sqcap$.

## 5.4 Bounding the Size of the Cover Relation of a Lattice

In this section, we give a bound on the number of edges in the cover relation of a lattice. Our description follows the reference [**?**]. Consider a poset with $n$ elements. Let $e_<$ be the number of edges in the cover relation and $e_\leq$ be the number of edges in the poset relation.

Then, it is clear that $e_< \leq e_\leq \leq n^2$.

**Theorem 5.3** *For a lattice L,*

$$n - 1 \leq e_< \leq n^{3/2}$$

We prove that $e_< \leq n^{3/2}$.

**Proof:** The lower bound is clear because every element in the lattice has at least one lower cover (except the smallest element). We show the upper bound.

Let $L$ be an inf-semilattice. Consider two distinct elements $x$, $y \in L$. Let $B(x)$ denote the set of elements covered by $x$. $B(x) \bigcap B(y)$ cannot have more than one element because that would violate inf-semilattice property. Let $B'(x) = B(x) \bigcup \{x\}$. Hence,

$$|B'(x) \bigcap B'(y)| \leq 1$$

Let

$$L = \{x_0, \ldots, x_{n-1}\}$$

$$b_i = |B(x_i)|$$

Because there is no pair in common between $B'(x)$ and $B'(y)$ for distinct $x$ and $y$, we get

$$\sum_{i=0}^{n-1} \binom{b_i + 1}{2} \leq \binom{n}{2}$$

Simplifying, we get

$$\sum_{i=0}^{n-1} (b_i^2 + b_i) \leq n^2 - n$$

Dropping $b_i$ from the left hand side and $-n$ from the right hand side, we get

$$\sum_{i=0}^{n-1} b_i^2 < n^2$$

Since $e_< = \sum b_i$, we get

$$\left(\frac{e_<}{n}\right)^2 = \left(\frac{\sum b_i}{n}\right)^2 \leq \frac{\sum b_i^2}{n} \leq \frac{n^2}{n} = n$$

The first inequality follows from Cauchy-Schwarz inequality.

Therfore,

$$e_<^2 \leq n^3$$

■

## 5.5 Join-Irreducible Elements Revisited

We have defined join-irreducible elements of a poset in Chapter **??**. Since lattices are posets with some additional properties, the same definition carries over. In fact, the concept of join-irreducible elements is more natural in the setting of lattices where the join operator between elements is always defined. In this section, we revisit join-irreducibles in lattices to show some of their useful properties.

For finite lattices, the elements with only one lower cover are join-irreducible elements (see Problem 5.4)

**Example 1:** In the lattice shown in Figure 5.4, $x$ and $y$ are the only join-irreducible elements.

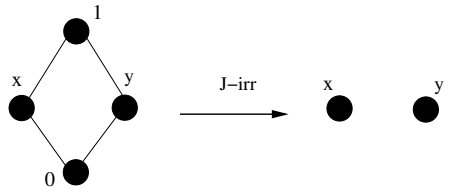



Figure 5.4: Join-irreducible elements: $J(L) = \{x, y\}$

**Example 2:** In a chain, all elements except the least element are join-irreducible.

**Example 3:** In a boolean lattice only the atoms (elements that cover 0) are join-irreducible elements. As an exercise, prove that if $x \in L$ covers 0, then it is join-irreducible.

**Lemma 5.4** *Let $L$ be a finite lattice. The following two are equivalent.*

1. $\forall a, b \in L : x = (a \sqcup b) \Rightarrow (x = a) \vee (x = b)$
2. $\forall a, b \in L : (a < x) \wedge (b < x) \Rightarrow (a \sqcup b < x)$

**Proof:** ($\Rightarrow$) Assume $x$ is join-irreducible. In general, we know:
$\forall a, b \in L : (a < x) \wedge (b < x) \Rightarrow (a \sqcup b \leq x) \Rightarrow (a \sqcup b < x) \vee (a \sqcup b = x)$

If $(a \sqcup b = x)$, then since $x$ is join-irreducible, $x = a$ or $x = b$ must be true, which is clearly a contradiction to our assumption in (2). Therefore, $(a \sqcup b < x)$ has to be true.

($\Leftarrow$) Assume $(x = a \sqcup b)$. This means, $x \geq a \wedge x \geq b$. It is given that $(x > a) \wedge (x > b) \Rightarrow (x > a \sqcup b)$,
which is a contradiction to our assumption in (1). So either $(x = b)$ or $(x = a)$ must be true.

■

**Proof:**

**Lemma 3** *For a finite lattice $L$, $a \leq b$ is equivalent to $\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b$.*

**Proof**: For the forward direction, $a \leq b$ and $x \leq a$ implies, for any $x$, $x \leq b$ by transitivity. For the reverse direction, denote by $h(z)$ the *height* of $z$, i.e. the length (number of edges) of the *longest* path from $z$ to $\inf L$ in the cover graph (well-defined, since $L$ is a finite lattice). We will prove the property

$$P(a) \;:=\; \forall b : ((\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b) \Rightarrow a \leq b)$$

for all $a \in L$ by *strong* induction on $h(a)$. Given $a$, consider an arbitrary $b$ and assume

(LHS) $\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b$ and (IH) $P(c)$ holds for all $c$ with $h(c) < h(a)$.

(1) If $h(a) = 0$, then $a = \inf L \leq b$, and $P(a)$ is vacuously true.
(2) If $a$ is join-irreducible, then, using $x := a$ in (LHS), $a \leq b$ follows, and $P(a)$ is again vacuously true.
(3) Now assume $h(a) > 0$ and $a$ not join-irreducible. Then there exist $c \neq a, d \neq a$ such that $a = c \sqcup d$. Since $c \neq a$, we can conclude that $h(a) \geq h(c) + 1$ ($h$ measures *longest* paths!). By (IH), $P(c)$ holds, i.e. $\forall b : ((\forall x : x \in J(L) : x \leq c \Rightarrow x \leq b) \Rightarrow c \leq b)$. We will use $P(c)$ to show $c \leq b$: assume $x \in J(L)$ with $x \leq c$, then $x \leq c \sqcup d = a$, hence $x \leq a$, thus, by (LHS), $x \leq b$. Property $P(c)$ delivers that $c \leq b$. Similarly, one can derive $d \leq b$, hence $c \sqcup d \leq b$, and with $a = c \sqcup d$ we obtain $a \leq b$.

■

Notice that we previously had a similar lemma with posets : $a \leq b \equiv \forall z : z \leq a \Rightarrow z \leq b$.
Obviously, the advantage of the previous lemma is that it only requires comparisons between join-irreducible elements and not all elements $\leq a$.

The following result strengthens the statement that $a = \sqcup_{x \le a} x$:

**Lemma 5.5** *For a finite lattice $L$ and any $a \in L$,*

$$a = \bigsqcup \{x \in J(L) : x \le a\}.$$

**Proof:** Let $T = \{x \in J(L) : x \le a\}$. We have to show that $a = \text{lub}(T)$.

Since any $x \in T$ satisfies $x \le a$, $a$ is an upper bound on $T$. Consider *any* upper bound $u$:

$$\begin{array}{ll} & u \text{ is an upper bound on } T \\ \Leftrightarrow & \langle \text{Definition of upper bound} \rangle \\ & x \in T \Rightarrow x \le u \\ \Leftrightarrow & \langle \text{Definition of } T \rangle \\ & (x \in J(L) \wedge x \le a) \Rightarrow x \le u \\ \Leftrightarrow & \langle \text{Elementary propositional logic: } (a \wedge b) \Rightarrow c \equiv a \Rightarrow (b \Rightarrow c) \rangle \\ & x \in J(L) \Rightarrow (x \le a \Rightarrow x \le u) \\ \Leftrightarrow & \langle \text{Lemma 3} \rangle \\ & a \le u, \end{array}$$

so $a$ is in fact the least upper bound on $T$.

■

Notice one special case: since $a := \inf L$ is not considered join-irreducible, the lemma confirms that $\inf L = \bigsqcup \emptyset$.

Finally, the following lemma shows that the requirement of join-irreducible elements can be stated in terms of $\le$ rather than $=$ for distributive lattices.

**Lemma 5.6** *In a finite distributive lattice (FDL) $L$, an element $x$ is join-irreducible if and only if*

$$x \neq \inf L \quad \textit{and} \quad \forall a, b \in L : x \le a \sqcup b \Rightarrow (x \le a \vee x \le b). \tag{5.1}$$

**Proof:** If $x$ is join-irreducible, then $x \neq \inf L$ by definition, and

$$\begin{array}{ll} & x \le a \sqcup b \\ \Leftrightarrow & \langle \text{property of } \le \text{ and } \sqcap \rangle \\ & x = x \sqcap (a \sqcup b) \\ \Leftrightarrow & \langle L \text{ distributive} \rangle \\ & x = (x \sqcap a) \sqcup (x \sqcap b) \\ \Rightarrow & \langle x \text{ join-irreducible} \rangle \\ & x = x \sqcap a \vee x = x \sqcup b \\ \Leftrightarrow & \langle \text{property of } \le \text{ and } \sqcap \rangle \\ & x \le a \vee x \le b. \end{array}$$

Conversely, assume (9.2), and let $x = a \sqcup b$. Then $x \le a \sqcup b$, hence $x \le a \vee x \le b$. On the other hand, $x = a \sqcup b$ implies $x \ge a \wedge x \ge b$. From the last two, since $\vee$ distributes over $\wedge$, it follows that $x = a \vee x = b$.

■

**Corollary 5.7a** *In a finite distributive lattice $L$, an element $x$ is join-irreducible if and only if*

$$\forall V \subset L: \qquad V = \emptyset \quad \vee \quad \left( x \leq \bigsqcup V \Rightarrow \exists v : v \in V : x \leq v \right). \tag{5.2}$$

**Proof:** If $x$ is join-irreducible, then for $V \subset L$, property (9.3) is proven by induction on $|V|$, the base case ($|V| = 2$) making use of lemma 9.2. Conversely, if (9.3) holds, consider a two-element set, say $V = \{a, b\}$. Lemma 9.2 immediately delivers that $x$ is join-irreducible.

■

5.1. A poset $(X, P)$ is a lattice iff for all finite subsets $Y \subseteq X$, $\sup Y$ and $\inf Y$ exist.

5.2. Show that absorption implies idempotency.

5.3. Show that the bound for $e_<$ can be improved when $L$ is distributive, i.e., show that if $L$ is distributive then

$$e \leq \frac{nlog_2\ n}{2}$$

5.4. Show that in finite lattices an element is join-irreducible iff it has only one lower cover.

# Chapter 6

# Lattice Completion

## 6.1 Introduction

We have seen that lattices are nicer structures than general posets because they allow us to take the meet and the join for any pair of elements in the set. What if we wanted to take the join and the meet of arbitrary sets? Complete lattices allow us to do exactly that. All finite lattices are complete, so the concept of complete lattices is important only for infinite lattices. In this chapter we first discuss complete lattices and show many ways in which complete lattices arise in mathematics and computer science. In particular, *topped* $\cap$*–structures* and *closure operators* give us complete lattices.

Next we consider the question: What if the given poset is not a complete lattice or even a lattice? Can we embed it into a complete lattice? This brings us to the notion of lattice completion which is useful for both finite and infinite posets.

### 6.1.1 Complete Lattices

We begin with the definition of a complete lattice.

**Definition 6.1** *A poset $(X, P)$ is a* ***complete lattice*** *iff for all subsets $Y \subseteq X$, sup $Y$ and inf $Y$ exist.*

For finite sets, the notion of lattices and complete lattices coincide, i.e., all finite lattices are always complete. When $X$ is infinite, then $(X, P)$ may be a lattice but not a complete lattice. Note that we require $\sup Y$ and $\inf Y$ to exist for all subsets (not just finite subsets). For example, consider the set of natural numbers under the usual order. This poset is a lattice but not a complete lattice because *sup* is not defined for subsets $Y$ such as the set of all odd numbers. Note that in some cases, a lattice can be easily made complete by adding a few "artificial" elements. For example, we can add a special element "$\infty$" to the set of natural numbers. The set $\mathbb{N} \cup \{\infty\}$ is a complete lattice. It may not be enough to add finite number of elements to complete a lattice. For example, the set of rationals with the elements $\infty$ and $-\infty$ is not a complete lattice. Consider the subset $Y = \{x \in X | x^2 \leq 2\}$. This set has no supremum in the set of rationals.

In the definition of complete lattices, note that when $Y$ is the empty set, the requirement that $\inf(Y)$ exist corresponds to existence of the $\top$ element. Similarly, when $Y$ is empty set $\sup(Y)$ equals the $\bot$ element.

The following lemma provides us a way of showing that a lattice is complete by only proving that the $inf$ exists, saving us half the work.

**Lemma 6.2 (Half-work Lemma)** *A poset $P$ is a complete lattice iff $inf(S)$ exists for every $S \subseteq P$.*

**Proof:** The forward direction ($\Rightarrow$) is easy.

To show $\Leftarrow$, we need to prove that $sup(S)$ exists for every $S$. To do so, we will formulate $sup(S)$ in terms of the $inf$ of some other set (see Figure 6.1).

Consider the set $T$ of upper bounds of $S$, i.e.,

$$T = \{x \in X : \forall s \in S : s \leq x\}.$$

Now let $a = inf(T)$. We claim that $a = sup(S)$. From the definition of $T$, we get that $\forall s \in S : \forall t \in T : s \leq t$. Since, $a = \inf(T)$, it follows that $\forall s \in S : s \leq a$. Thus, $a$ is an upper bound of $S$.

Further, for any upper bound $t$ of $S$, we know that $t \in T$. Therefore, $a \leq t$ because $a = \inf(T)$. Thus, $a = sup(S)$.

■

Note that the set $T$ in the proof may be empty. In that case, $a$ would be the top element of $P$.

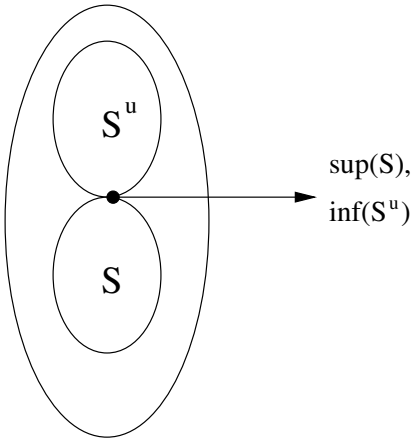



Figure 6.1: Illustration for the Half-Work Lemma.

## 6.2 Alternate Definitions for Complete Lattices

We find from experience that good structures tend to have multiple, equivalent ways of defining them. This is good for us in at least two ways. First, it provides multiple ways of characterizing the structure, hence offering us more flexibility in doing proofs. In addition, it may provide us efficient algorithms for dealing with the structures. Here we study two alternative definitions for complete lattices, and then show their equivalence.

### 6.2.1 Closure Operators

Given a set $X$, we denote the set of all subsets of $X$ (the power set of $X$) by $2^X$. We now define closure operators on the set $2^X$.

**Definition 6.3 (Closure Operator)** *Any map $C : 2^X \mapsto 2^X$ is a closure operator iff it satisfies:*

1. $\forall A \subseteq X, A \subseteq C(A)$ *(increasing)*
2. $\forall A, B \subseteq X, A \subseteq B \Rightarrow C(A) \subseteq C(B)$ *(monotone)*
3. $\forall A \subseteq X, C(C(A)) = C(A)$ *(idempotent)*

1. Given a poset $P = (X, \leq)$ and any subset $Y \subseteq X$, we define

$$C(Y) = \{x \in X : \exists y \in Y : x \leq y\}$$

. $C(Y)$ adds to $Y$ all elements that are smaller than some element in $Y$. It is easy to verify that $C$ is a closure operator.

2. Our next example is from theory of formal languages. Let $\Sigma$ be any alphabet and $L \subseteq \Sigma^*$ be a language (set of finite strings) defined over $\Sigma$. Then, **Kleene closure** of $L$ consists of all finite strings that can be obtained by concatenation of strings in $L$. Thus, Kleene closure for a set of languages is $C : \pm^* \mapsto \pm^*$ such that

$C(L) = \{x.y : x \in L \wedge y \in L\}$, for $L \in \mathcal{L}$

where the operator . means concatenation. It is easy to verify that Kleene closure satisfies all requirements of the closure operator.

Before we show the relationship between complete lattices and closure operators, we present yet another alternate definition, called *topped $\cap$–structures*.

### 6.2.2 Topped $\cap$–Structures

**Definition 6.4 (Topped $\cap$–Structure)** *For a set $X$, let $\mathcal{L} \subseteq 2^X$. $\mathcal{L}$ is a topped $\cap$–structure iff the following hold:*

1. $X \in \mathcal{L}$
2. $[\forall i \in I : A_i \in \mathcal{L}] \Rightarrow (\bigcap_{i \in I} A_i \in \mathcal{L})$, *where $I$ is any indexing set.*

Thus a topped $\cap$–structure is a set of sets that contains the original set, and is closed under arbitrary intersection. From the half-work lemma it is clear that topped $\cap$–structures form a complete lattice under the relation $\subseteq$.

We now show that closure operators are equivalent to topped $\cap$–structures.

**Theorem 6.5** *Let $C$ be a closure operator defined on $2^X$. Let $\mathcal{L}(C) = \{A \subseteq X : C(A) = A\}$ be a family of subsets in $2^X$. Then $\mathcal{L}(C)$ is a topped $\cap$–structure.*

*Conversely, let $\mathcal{L}$ be a topped $\cap$–structure. Then*

$$C(\mathcal{L})(A) = \cap\{B \in \mathcal{L} | A \subseteq B\}$$

*is a closure operator.*

**Proof:** Left as an exercise.

■

## 6.3 Dedekind–MacNeille Completion

For any poset $P$, *sup* or *inf* may not be defined for all subsets of elements of $P$. We would like to be able to find a complete lattice that has $P$ embedded in it. We now turn to one such completion, the *Dedekind–MacNeille completion* (also called *normal completion* or *completion by cut*).

To define the notion of cuts, we first giev definitions of down sets, up sets, upper bounds, and lower bounds.

$$y \in D(x) \quad \equiv \quad y \leq x \qquad \text{(down set)} \tag{6.1}$$

$$y \in U(x) \quad \equiv \quad x \leq y \qquad \text{(up set)} \tag{6.2}$$

$$y \in A^u \quad \equiv \quad \forall x \in A : x \leq y \qquad \text{(upper bounds)} \tag{6.3}$$

$$y \in A^l \quad \equiv \quad \forall x \in A : y \leq x \qquad \text{(lower bounds)} \tag{6.4}$$

Note that $D[x]$ is an order ideal. It is called the *principal ideal* of $x$. Similarly, $U[x]$ is an order filter. It is called the *principal filter* of $x$.

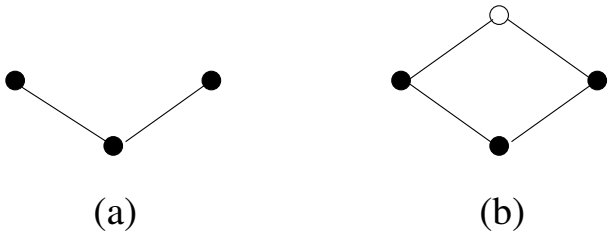



Figure 6.2: (a) The original poset. (b) A completion, where the unshaded vertex is the added element.

*Remark:* The Dedekind–MacNeille completion will henceforth be referred to as the $DM$ completion; similarly, the $DM$ completion of a specific poset $P$ will be denoted as $DM(P)$.

The $DM$ completion is based on a closure operator. Since we know that closure operators are equivalent to complete lattices, we know that applying the $DM$ completion to $P$ will give us our

desired result. Before continuing, let us recall that $A^u$ is the set of all upper bounds of $A$, $A^l$ the set of all lower bounds of $A$, and $A^{ul} = (A^u)^l$. We now define a *cut*.

**Definition 6.6** *For a poset $P = (X, \leq)$, a subset $A \subseteq X$ is a* **cut** *if $A^{ul} = A$.*

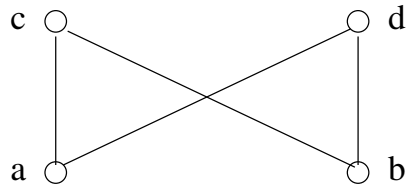



Figure 6.3: A poset that is not a complete lattice.

Consider the poset in Figure 6.3. For $A = \{a, b, c\}$, we have $A^u = \{c\}$ and $A^{ul} = \{a, b, c\}$, so $A$ is a cut. On the other hand, for $A = \{b, c\}$, we have $A^u = \{c\}$ and $A^{ul} = \{a, b, c\} \neq A$, so $A$ is not a cut.

We are now ready to define $DM(P)$.

**Definition 6.7 (Dedekind–MacNeille Completion of a Poset)** *For a given poset $P$, the Dedekind–MacNeille completion of $P$ is the poset formed with the set of all the cuts of $P$ under the set inclusion. Formally,*

$$DM(P) = (\{A : A^{ul} = A\}, \subseteq).$$

For the poset in Figure 6.3, the set of all cuts is:

{{}, {a}, {b}, {a,b}, {a,b,c}, {a,b,d}, {a,b,c,d} }.

The poset formed by these sets under the $\subseteq$ relation is shown in Figure 6.4. This new poset is a complete lattice. Our original poset $P$ is embedded in this new structure. We also note that our set of cuts forms a topped $\cap$–structure.

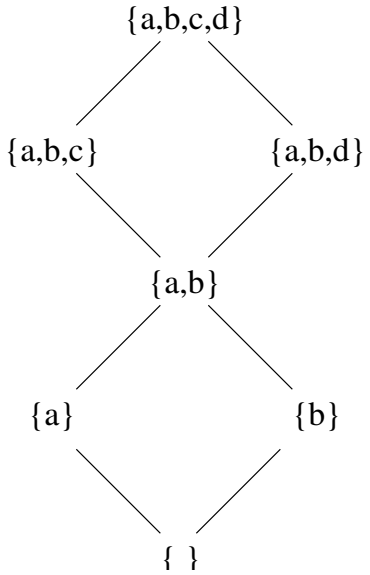



Figure 6.4: The $DM$ completion of the poset from Figure 6.3.

Figure 6.5 illustrates some more $DM$ completions.

There are other ways to embed a poset $P$ in a complete lattice. One way is via ideals; Figure 6.6 shows the complete lattice that embeds the poset from Figure 6.5 (b). Notice that the embedding by ideals yields a larger lattice than that by $DM$ completion. This is an important property of the $DM$ completion: $DM(P)$ results in the *smallest* complete lattice that embeds $P$.

## 6.4 Structure of $DM(P)$ of a poset $P = (X, \leq)$

In this section, we explore the structure of $DM(P)$ and exhibit a mapping that show embedding of $P$ into $DM(P)$. We show calculational style of proofs in this section. To facilitate such proofs, we give rules to eliminate or introduce *inf* and *sup* symbols as follows.

**Rule 6.8** *Assuming* $inf(A)$ *exists:* $x \leq inf(A) \equiv \forall a \in A : x \leq a$

**Rule 6.9** *Assuming* $sup(A)$ *exists:* $x \geq sup(A) \equiv \forall a \in A : x \geq a$

To manipulate formulae, it is also desirable to remove the "$\forall$" symbols:

**Rule 6.10** $(x \leq y) \equiv (\forall z : z \leq x \Rightarrow z \leq y)$

The forward direction holds due to transitivity $\leq$. The backward direction can be shown by substituting $x$ for $z$ and reflexivity of $\leq$

We are now ready to prove results about the structure of $DM(P)$. We first show that for any $x \in P$, $D[x]$ is a cut and therefore belongs to $DM(P)$.

**Lemma 6.11** $\forall x \in P : D[x]^{ul} = D[x]$.

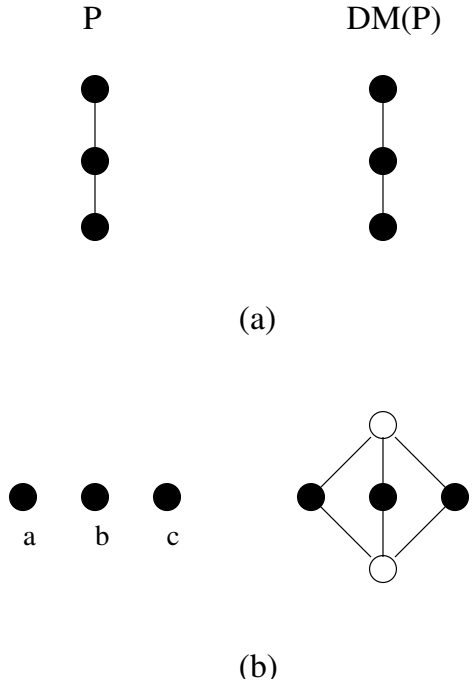



Figure 6.5: Two posets and their $DM$ completions.

**Proof:**
We show that

$$D[x]^u = U[x] \tag{6.5}$$

By duality:

$$U[x]^l = D[x] \tag{6.6}$$

Therefore, the result follows from (6.5) and (6.6). To prove (6.5):

$$\begin{aligned}
& y \in D[x]^u \\
\equiv & \{\text{definition of upper bound - Rule } \textbf{??}\} \\
& \forall z : z \in D[x] : z \leq y \\
\equiv & \{\text{definition of D - Rule } \textbf{??}\} \\
& \forall z : z \leq x : z \leq y \\
\equiv & \{\text{rules of predicate calculus}\} \\
& \forall z : z \leq x \Rightarrow z \leq y \\
\equiv & \{\text{property of } \leq\} \\
& x \leq y \\
\equiv & \{\text{definition of U - Rule } \textbf{??}\} \\
& y \in U[x]
\end{aligned}$$

■

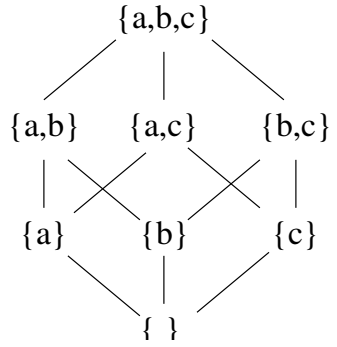



Figure 6.6: The complete lattice that embeds the poset from Figure 6.5 (b).

Therefore, $D[x]$ is a map from $P$ to $DM(P)$. We now show that $D[.]$ preserves all least upper bounds and greatest lower bounds in $P$.

**Lemma 6.12** *Assume $inf(A)$ exists. Then:*

$$\bigcap_{\forall a \in A} D[a] = D[inf(A)].$$

**Proof:**

$$\begin{aligned}
& x \in \bigcap_{a \in A} D[a] \\
\equiv & \{\cap = \forall\ \} \\
& \forall a \in A : x \in D[a] \\
\equiv & \{\text{Rule } \mathbf{??}\} \\
& \forall a \in A : x \leq a \\
\equiv & \{\text{Rule 6.8}\} \\
& x \leq inf(A) \\
\equiv & \{\text{Rule } \mathbf{??}\} \\
& x \in D[inf(A)]
\end{aligned}$$

■

## 6.5 Problems

1. Assume $sup(A)$ exists in $P$. Show that

$$A^{ul} = D[sup(A)]$$

2. Consider a poset $(X, \leq)$. Assume that for $S \subseteq X$ , $infS$ exists. Let $y = infS$. $\forall x \in X : x \leq y \Leftrightarrow \forall s \in S : x \leq s$

3. Consider $S, T \subseteq X$, where X is a poset. SHow that $sup(S) \leq inf(T) \Rightarrow \forall s \in S, \forall t \in T : s \leq t$.

4. Show that the completion of the set of rational numbers using cuts results in the set of real numbers.

# Chapter 7

# Morphisms

## 7.1 Introduction

In this chapter we study homomorphisms between lattices. Many of the concepts and results are analogous to the results typically studied in group and ring theory. For example, the concept of lattice homomorphism is analogous to that of group homomorphism. These concepts have natural applications in the theory of distributed systems. In particular, we show applications in predicate detection.

## 7.2 Lattice Homomorphism

**Definition 7.1 (Lattice homomorphism)** *Given two lattices $L_1$ and $L_2$, a function $f : L_1 \to L_2$ is a* **lattice homomorphism** *if $f$ is* **join-preserving** *and* **meet-preserving**, *that is, $\forall x, y \in L_1$ :*

$$f(x \sqcup y) = f(x) \sqcup f(y)\,, \quad \textit{and} \quad f(x \sqcap y) = f(x) \sqcap f(y)$$

The property of preserving joins (or meets) is stronger than that of preserving the partial order. This fact is shown in the next lemma.

**Lemma 7.2** *If $f$ is join-preserving, then $f$ is monotone.*

**Proof:**

$$\begin{array}{ll} & x \leq y \\ \equiv & \{ \text{ Connecting Lemma } \} \\ & (x \sqcup y) = y \\ \Rightarrow & \{ \text{ applying f on both sides } \} \\ & f(x \sqcup y) = f(y) \\ \Rightarrow & \{ \text{ f is join-preserving } \} \\ & (f(x) \sqcup f(y)) = f(y) \\ \equiv & \\ & f(x) \leq f(y) \end{array}$$

■

**Exercise**

1. Prove that if $f$ is meet-preserving, then $f$ is monotone.

### 7.2.1 Lattice Isomorphism

**Definition 7.3 (Lattice isomorphism)** *Two lattices $L_1$ and $L_2$ are* **isomorphic** *(denoted by $L_1 \approx L_2$) iff there exists a bijection $f$ from $L_1$ to $L_2$ which is a lattice homomorphism.*

**Theorem 7.4** *$L_1$ and $L_2$ are isomorphic iff there exists a bijective $f : L_1 \to L_2$ such that both $f$ and $f^{-1}$ are monotone.*

**Proof:** We leave the forward direction as an exercise. We prove the backward direction. Let $f$ be a bijection from $L_1$ to $L_2$ such that $f$ and $f^{-1}$ are monotone. We show that $f$ preserves joins.

**1.** We first show that $\forall a, b \in L_1 : f(a) \sqcup f(b) \leq f(a \sqcup b)$

$$
\begin{array}{ll}
 & true \\
\Rightarrow & \{ \text{ definition of join } \} \\
 & (a \leq (a \sqcup b)) \wedge (b \leq ((a \sqcup b)) \\
\Rightarrow & \{ \text{ monotone } f \ \} \\
 & f(a) \leq f(a \sqcup b) \wedge f(b) \leq f(a \sqcup b) \\
\Rightarrow & \{ \text{ definition of join } \} \\
 & f(a) \sqcup f(b) \leq f(a \sqcup b)
\end{array}
$$

**2.** We now show that $\forall a, b \in L_1 : f(a) \sqcup f(b) \geq f(a \sqcup b)$.
It is sufficient to show that for any $u \in L_2$,
$(f(a) \sqcup f(b)) \leq u \Rightarrow f(a \sqcup b) \leq u$

$$
\begin{array}{ll}
 & (f(a) \sqcup f(b)) \leq u \\
\Rightarrow & \{ \text{ definition of join } \} \\
 & (f(a) \leq u) \wedge (f(b) \leq u) \\
\Rightarrow & \{ \ f^{-1} \text{ is monotone } \} \\
 & (a \leq f^{-1}(u)) \wedge (b \leq f^{-1}(u)) \\
\Rightarrow & \{ \text{ definition of join } \} \\
 & (a \sqcup b) \leq f^{-1}(u) \\
\Rightarrow & \{ \text{ monotone } f \ \} \\
 & f(a \sqcup b) \leq u
\end{array}
$$

The proof for $\sqcap$ is dual.

■

### 7.2.2 Lattice Congruences

Lattice homomorphisms can also be understood using the notion of lattice congruences. Informally, a congruence on a lattice (or any algebra) is an equivalence relation that preserves lattice operations (operations of the algebra, respectively). Recall that a binary relation $\theta$ on a set $X$ is an equivalence relation iff $\theta$ is reflexive, symmetric, and transitive.

The general structure of equivalence relation is easy to understand because of the well-known correspondence between an equivalence relation and partitions of a set. An equivalence relation partitions a set into disjoint subsets called **equivalence classes** or **blocks**. A block $[x]_\theta$ consists of all elements in $X$ that are related to $x$. For example, consider the following relation on the set of natural numbers:

$$\theta_n = \{(x, y) | x \ = \ y \ mod \ n\} \text{ for } n \geq 1.$$

This relation partitions the set of natural numbers into $n$ blocks. Conversely, given a partition of a set into a union of non-empty disjoint subsets, one can easily define an equivalence relation whose equivalence classes are the blocks in the partition.

## Exercises

7.1. Let $Eq(X)$ be the set of all equivalence relations defined on a set $X$. For any two elements $\theta_1, \theta_2$ of $Eq(X)$, we define $\theta_1$ to be less than or equal to $\theta_2$ iff $\theta_1 \subseteq \theta_2$. Show that $Eq(X)$ forms a complete lattice under this order. (Hint: Show that it is a topped $\cap$-closed structure.)

An equivalence relation is called a **congruence** if it preserves operations of the algebra. An equivalence relation on a lattice $L$ which is compatible with both join and meet operations is called a congruence on $L$. Formally,

**Definition 7.5 (Lattice Congruence)** *An equivalence relation $\theta$ on a lattice $L$ is a congruence iff $x \equiv y (\mathrm{mod}\, \theta)$ implies that*

1. $\forall z : (z \sqcup x) \equiv (z \sqcup y) \ (\mathrm{mod}\, \theta)$
2. $\forall z : (z \sqcap x) \equiv (z \sqcap y) \ (\mathrm{mod}\, \theta)$

The following lemma is left as an exercise.

**Lemma 7.6** *Let $\theta$ be a congruence of lattice $L$; then*

1. $(a \equiv b (\mathrm{mod}\, \theta) \wedge (a \leq c \leq b)) \Rightarrow a \equiv c (\mathrm{mod}\, \theta)$.
2. $a \equiv b (\mathrm{mod}\, \theta)$ *iff* $(a \sqcap b) \equiv (a \sqcup b) (\mathrm{mod}\, \theta)$.

### 7.2.3 Quotient Lattice

Given a congruence $\theta$ on a lattice $L$, we can define quotient of $L$ with respect to $\theta$ as $L/\ \theta = \{[a]_\theta | a \in L\}$. The join and meet operations between the blocks of $\theta$ are defined naturally as:
$[a]_\theta \sqcup [b]_\theta = [a \sqcup b]_\theta$.
Similarly,
$[a]_\theta \sqcap [b]_\theta = [a \sqcap b]_\theta$.
The set $L/\theta$ together with the meet and join operations defined above is called the quotient lattice or the reduced lattice.

### 7.2.4 Lattice Homomorphism and Congruence

We now show that given a lattice homomorphism, one can define a lattice congruence and vice versa.

**Lemma 7.7** *Let $L$ and $K$ be lattices and $f : L \to K$ be a lattice homomorphism. Define the equivalence relation $\theta$ on $L$ as follows : $u \equiv v(\mathrm{mod}\,\theta) \iff f(u) = f(v)$. Then $\theta$, called the* **kernel** *of $f$ and denoted as $ker f$, is a congruence.*

**Proof:** We need to show that if $a \equiv b(\mathrm{mod}\,\theta)$ and $c \equiv d(\mathrm{mod}\,\theta)$ then $a \sqcup c \equiv b \sqcup d(\mathrm{mod}\,\theta)$

$f(a \sqcup c)$
$=\{$ $f$ is a lattice homomorphism $\}$
$f(a) \sqcup f(c)$
$= \{a \equiv b,\, c \equiv d\}$
$f(b) \sqcup f(d)$
$=\{$ $f$ is a lattice homomorphism $\}$
$f(b \sqcup d)$

Therefore, $a \sqcup c \equiv b \sqcup d(\mathrm{mod}\,\theta)$. The proof for $\sqcap$ is dual.

■

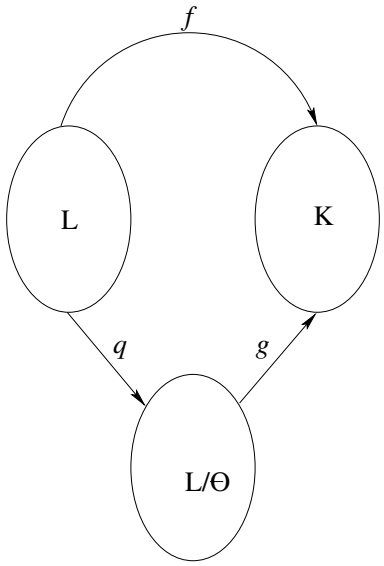



Figure 7.1: Fundamental homomorphism theorem

The following is a standard algebraic result.

**Theorem 7.8 (Fundamental homomorphism theorem)** *Let $L$ and $K$ be lattices and $f : L \to K$ be a lattice homomorphism. Define $\theta = ker\ f$. The mapping $g : L/\theta \to K$ given by $g([a]_\theta) = f(a)$ is well defined and $g$ is an isomorphism between $L/\theta$ and $K$.*

**Proof:** Left as an exercise.

■

## 7.3 Properties of Lattice Congruence Blocks

In this section we explore the structure of a congruence block. We will show that every congruence block is a quadrilateral-closed convex sublattice. To this end, we first define a quadrilateral in a lattice.

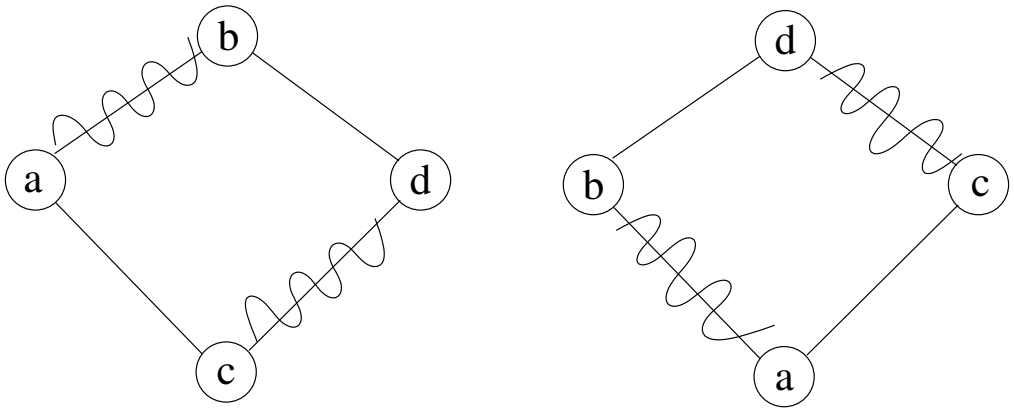



Figure 7.2: Quadrilateral closed structures

The elements $a$, $b$, $c$ and $d$ form a **quadrilateral** $< a, b; c, d >$ iff

1. $a < b$ and $c < d$

2. $(a \sqcup d = b) \wedge (a \sqcap d = c)$ OR
   $(b \sqcup c = d) \wedge (b \sqcap c = a)$

Hence a block (congruence class) of $\theta$ is quadrilateral-closed if; $a$, $b \in$ some block of $\theta \Rightarrow$ $c$, $d \in$ some block of $\theta$, where $< a, b; c, d >$ is a quadrilateral.

**Theorem 7.9** *$\theta$ is a congruence iff*

1. *Each block of $\theta$ is a sublattice*

2. *Each block is convex*

3. *The blocks are quadrilateral-closed*

**Proof:**

1. Each block $A$ is a sublattice
   We need to show that $\forall a, b \in A : (a \sqcup b \in A) \wedge (a \sqcap b \in A)$
   $$a \equiv_\theta b$$
   $$\equiv$$
   $$a \sqcup a \equiv_\theta b \sqcup a$$
   $$\equiv$$
   $$a \equiv_\theta b \sqcup a$$

2. Each block is convex
   We show that $\forall a,\ b \in A : (a \leq c \wedge c \leq b) \Rightarrow c \in A$
   $a \equiv_\theta b$
   $\equiv$
   $a \sqcup c \equiv_\theta b \sqcup c$
   $\equiv$
   $c \equiv_\theta b$

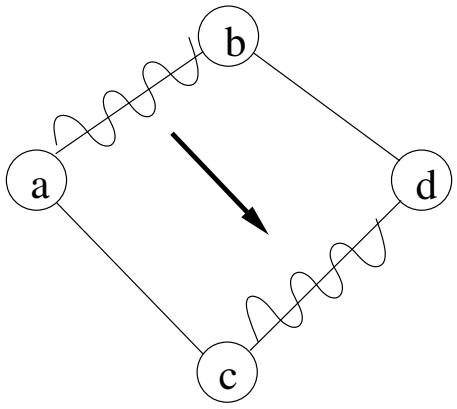



Figure 7.3: Quadrilateral closed

3. The blocks are quadrilateral-closed.
   It is sufficient to show that $a \equiv_\theta b \Rightarrow c \equiv_\theta d$, where $< a, b; c, d >$ is a quadrilateral.
   $a \sqcap d = c$
   $b \sqcap d = d$
   Hence a $\equiv_\theta b \Rightarrow c \equiv_\theta d$
   The converse, i.e, conditions (1) ,(2) and (3) imply that $\theta$ is a congruence class is left as an exercise.

■

### 7.3.1 Congruence Lattice

The congruences of a lattice form another lattice which we call the *congruence lattice*. Thus $Con\ L \equiv$ set of all congruences on lattice $L$. The proof of the following result can be found in [**?**].

**Theorem 7.10** *For any finite lattice $L$, $Con\ L$ is a finite distributive lattice. Conversely, given any finite distributive lattice $L$, there exists a lattice $K$ such that $Con\ K = L$.*

### 7.3.2 Principal Congruences

For a pair of elements,$(a, b)$, find all the congruences $\theta$ such that $a \equiv_\theta b$. Then, the *principal congruence* generated by $(a, b)$ is defined as

$$\theta(a, b) = \bigcap \{\theta \in Con\ L | (a, b) \in \theta\}$$

Hence $\theta(a, b)$ is the smallest congruence permitting us to collapse elements $a$ and $b$.

## 7.4 Model Checking on Reduced Lattices

We now show an application of congruences to checking temporal logic formulas on Kripke structures (state transition diagrams) that form a lattice. System properties and specifications are assumed to be expressed in a restricted version of the *Computation Tree Logic (CTL)* [**?**] which does not have the *next-time* operator $X$. Next-time is not preserved by state reductions, and hence we focus on the remaining portion of the temporal logic, denoted $CTL_{-X}$. This contains the operators $EF$, $AF$, $EG$, $AG$, $EU$ and $AU$. Almost all the properties (like safety and liveness) that are of interest in distributed programs (especially in the asynchronous environment) can be expressed in $CTL_{-X}$. For instance, in a mutual exclusion algorithm, the property "once a process requests a lock, it eventually gets the lock", can be expressed as $AG(request \Rightarrow AF(lock))$. As another example, "a ring of $n$ processes always has exactly one token" can be expressed as $AG(token_1 + ... + token_n = 1)$.

We are interested in grouping together global states which have the same state with respect to the property that we wish to verify. For example, if we are interested in detecting the property $(x^2 + y > 10)$, then it would simplify the problem of detection if we can group together states that have the same $x$ and $y$ values. Doing this will induce an equivalence relation on the lattice and partition it into equivalence classes. However the structure formed by collapsing together the equivalence class elements may not preserve the temporal logic formula. We are interested in exact reductions in the sense that if a property holds in the original state lattice then it holds in the reduced state lattice and vice versa. If a property does not hold in the original state lattice then it also does not hold in the reduced state lattice and vice versa. Collapsing a "contiguous" set of states may not give exact reduction. We demonstrate this by a simple example of a global state graph in Figure 7.4. The black nodes represent states at which $\Phi$ holds. The property we are trying to verify is $AF : \Phi$. It is clear from the figure that $AF : \Phi$ holds in the original graph on the left but not on the reduced graph on the right.

We will show that if the equivalence class forms a congruence, then one can indeed use the reduced lattice. Consider a global state lattice $L$, the congruence $\theta$ and the reduced lattice $L/\theta$. Then corresponding to every path $P$ from bottom to top in $L/\theta$, there is a corresponding path in the original lattice $L$ from its bottom to top and vice versa. In the following we use $\prec_L$ to denote the cover relation in a lattice $L$. To prove the equivalence of paths between $L$ and $L/\theta$ we need Lemma 7.12 which states that two states in the reduced graph have a cover relation between them if and only if they contain states which have a cover relation between them. Using this, we prove the equivalence of paths in Lemma 7.13. We first need the following lemma:

**Lemma 7.11** *Given two congruence classes $A$ and $B$ in $L/\theta$, let $(a_\perp, a_\top)$ and $(b_\perp, b_\top)$ be the bottom and top element pairs of $A$ and $B$ respectively. If $B$ covers $A$ in $L/\theta$, then there exists a path from $a_\top$ to $b_\top$ in $L$ consisting only of nodes in $A$ and $B$.*

**Proof:**

Since $B$ covers $A$ in $L/\theta$, there exist elements $c \in A$ and $d \in B$ such that $c \leq d$ in $L$ (Figure 7.5). As $a_\perp \leq c$ and $d \leq b_\top$, we get

$$
\begin{aligned}
& a_\perp \leq b_\top \\
\Rightarrow \quad & \{property\ of\ meet\} \\
& a_\perp \sqcap a_\top = a_\top \sqcap b_\top \\
\equiv \quad & \{a_\perp \sqcap b_\top = a_\perp\}
\end{aligned}
$$

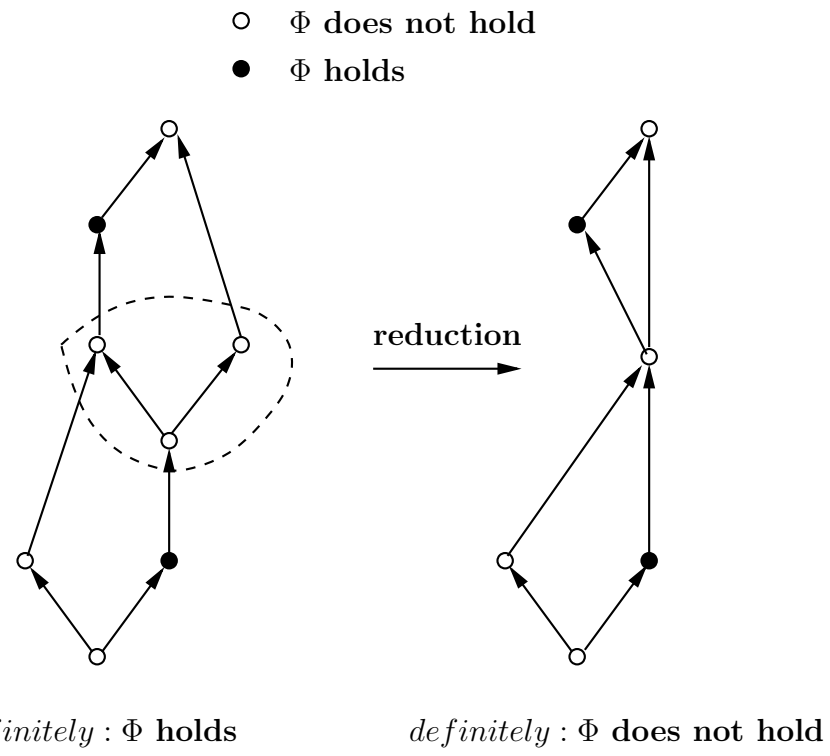



Figure 7.4: Simple reduction of state graph does not preserve path based formulas.

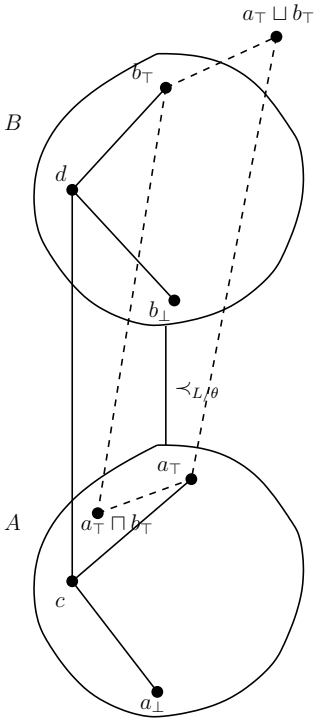



Figure 7.5: Proof of Lemma 7.11.

$$
\begin{array}{ll}
& a_\bot \leq a_\top \sqcap b_\top \\
\equiv & \{property\ of\ meet\} \\
& a_\bot \leq a_\top \sqcap b_\top \leq a_\top \\
\Rightarrow & \{A\ is\ convex\} \\
& a_\top \sqcap b_\top \in A
\end{array}
$$

Therefore is shown in Figure 7.5 $< b_\top, a_\top \sqcup b_\top; a_\top \sqcap b_\top, a_\top >$ forms a quadrilateral in $L$. Since $a_\top \sqcap b_\top \equiv a_\top (mod\ \theta)$, from the quadrilateral-closed property of congruences we have, $b_\top \equiv a_\top \sqcup b_\top (mod\ \theta)$. Therefore, $a_\top \sqcup b_\top \in B$. By definition, $b_\top$ is the top element of $B$ and $b_\top \leq a_\top \sqcup b_\top$ implies that $b_\top = a_\top \sqcup b_\top$. Therefore $a_\top \leq b_\top$ (Connecting Lemma).Hence there exists a path from $a_\top$ to $b_\top$ in $L$.

It remains to be shown that the path consists only of nodes belonging to $A$ or $B$. Pick any $e \in L$ such that it is on the path from $a_\top$ to $b_\top$. Thus $a_\top \leq e \leq b_\top$ and since $A \prec_{L/\theta} B$, hence by property of the covering relation either $e \in A$ or $e \in B$ which yields the desired result.

■

**Lemma 7.12** *Let $A$, $B \in L/\theta$, then*

$$A \prec_{L/\theta} B$$

*if and only if there exist $m$, $n \in L$ such that*

$$m \in A,\ n \in B \text{ and } n \prec_L n$$

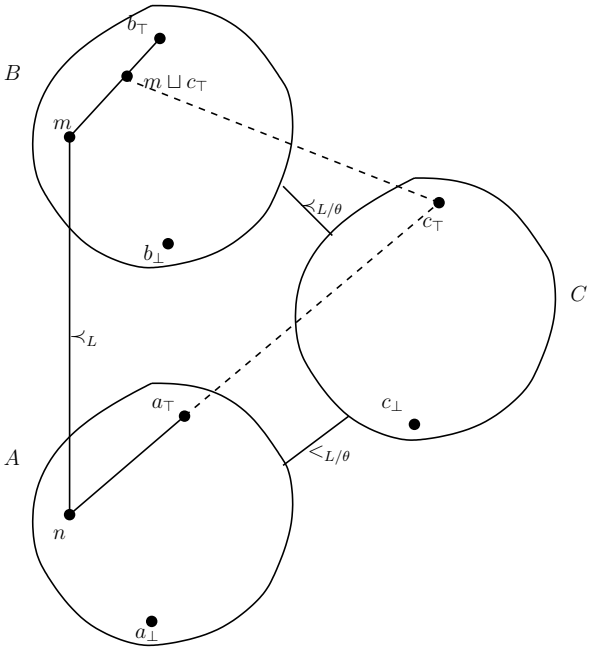



*Figure* 8(1)

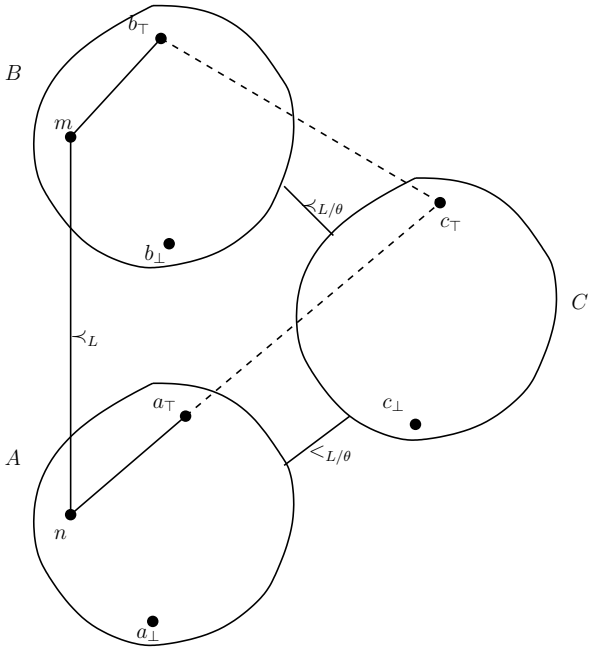



*Figure* 8(2)

Figure 7.6: Proof of Lemma 7.12.

**Proof:**

The forward direction of the proof follows from Lemma 7.11 as follows: We assume that $A \prec_{L/\theta} B$. Thus by Lemma 7.11 there exists a path from $a_\top$ to $b_\top$ in $L$ consisting only of nodes in $A$ and $B$. The first element in the path is $a_\top$ and let the second element on the path be $b$. Clearly $b \in B$ and $a_\top \prec_L b$. Thus there exist $m$, $n \in L$ such that $n \in A,\ m \in B$ *and* $n \prec_L m$.

To prove the converse, we assume that $A,\ B \in L/\theta$ and there exist $m,\ n \in L$ such that $n \in A,\ m \in B$ *and* $n \prec_L m$. Let us assume that there exists $C \in L/\theta$ such that $A <_{L/\theta} C \prec_{L/\theta} B$ (Figure 7.6). Because $n \prec_L m$, we cannot have $c_\top \leq m$ as there can only be one path from $n$ to $m$ in the covering graph (Figure 7.6(1)). Thus $\neg(c_\top \leq m)$ (Figure 7.6(2)). We first prove that $< m, m \sqcup c_\top; n, c_\top >$ forms a quadrilateral in $L$ and then show that this contradicts our assumption about the existence of $C$.

We first prove that $m \parallel c_\top$:

$C \prec_{L/\theta} B$
$\Rightarrow$ $\{Lemma\ 7.11\}$
$m \in B, b_\top \in B, c_\top < b_\top$
$\Rightarrow$ $\{Convexity\ of\ sublattice\ B, c_\top \notin B\}$
$\neg(m \leq c_\top)$
$\equiv$ $\{\neg(c_\top \leq m)\}$
$m \parallel c_\top$

Since $n \leq m$ and $m \leq b_\top$, by transitivity $n \leq b_\top$. Therefore, $< m, m \sqcup c_\top; n, c_\top >$ forms a quadrilateral in $L$. Now we show that $m \sqcup c_\top \in B$. From the fact that $B$ is a sublattice and

Lemma 7.11, we get:

$$
\begin{array}{ll}
& m \leq b_\top,\ c_\top \leq b_\top \\
\equiv & \{property\ of\ join\} \\
& m \sqcup c_\top \leq b_\top \\
\equiv & \{property\ of\ join\} \\
& m \leq m \sqcup c_\top \leq b_\top \\
\equiv & \{convexity\ of\ sublattice\ B\} \\
& m \sqcup c_\top \in B
\end{array}
$$

Since $m \sqcup c_\top \equiv m (mod\ \theta)$, hence from the quadrilateral -closed property of congruences we have, $n \equiv c_\top (mod\ \theta)$. Therefore, $c_\top \in A$. This contradicts our assumption that there exists $C \in L/\theta$ such that $A <_{L/\theta} C \prec_{L/\theta} B$.

■

Now we can prove that there is a one-to-one correspondence between paths of $L$ and $L/\theta$ with respect to the values of the variables that are relevant to the properties that we are trying to detect. The first part of the lemma says that for any path in $L/\theta$, if we look at the pre-images of the nodes on the path (corresponding to inverse of the lattice homomorphism function $f$), then there exists a subset of these pre-image nodes which also forms a path in $L$. Since $f$ is defined to preserve values of variables relevant to the property being detected, this enables us to prove that detecting temporal formula in $L$ is equivalent to detecting temporal formulae in $L/\theta$. Similarly, the second part of the lemma proves that for every path in $L$, if we look at the image of the nodes on the path then they form a corresponding path in $L/\theta$.

**Lemma 7.13** *[Equivalence of Paths] Let $f$ be the lattice homomorphism corresponding to $\theta$.*

1. *If $P = (P_1, ..., P_k)$ is a path from bottom to top in $L/\theta$, then there exists a path $Q$ in $L$ such that $Q \subseteq \{q \in L : \exists i \in [1,k] : q \in f^{-1}(P_i)\}$.*
2. *If $Q = (q_1, ..., q_k)$ is a path from bottom to top in $L$, then the set $P = \{f(q_1), ..., f(q_k)\}$ forms a path from bottom to top in $L/\theta$ (Figure 7.7 illustrates the lemma).*

**Proof:**

1. Consider the set $f^{-1}(P_i)$. For any $i \in [1,k]$, let $x_i$ and $y_i$ be the bottom and top elements of $f^{-1}(P_i)$ respectively ($f^{-1}(P_i)$ is a sublattice). Note that $x_1$ and $y_k$ are the bottom and top elements of $L$. From Lemma 7.11, we get that there is a path from $y_1 \rightarrow y_2 \rightarrow ... \rightarrow y_k$. Also since $f^{-1}(P_1)$ is a sublattice, hence there is a path $x_1 \rightarrow y_1$. Therefore there is a path from $x_1$ (bottom) to $y_k$ (top) in $L$.
2. Let $f(q_{i,1}), ..., f(q_{i,l})$ be the sequence obtained from $P$ after removing duplicate nodes. From the definition of $f(q_{i,k})$ and $f(q_{i,k+1})$, there exists $m,\ n \in L$ such that $m \in f(q_{i,k}),\ n \in f(q_{i,k+1})$ *and* $m \prec_L n$. Then Lemma 7.12 gives us that $f(q_{i,k}) \prec_{L/\theta} f(q_{i,k+1})$ and in general $f(q_{i,1}) \prec_{L/\theta} ... \prec_{L/\theta} f(q_{i,l})$ and thus this forms the desired path.

■

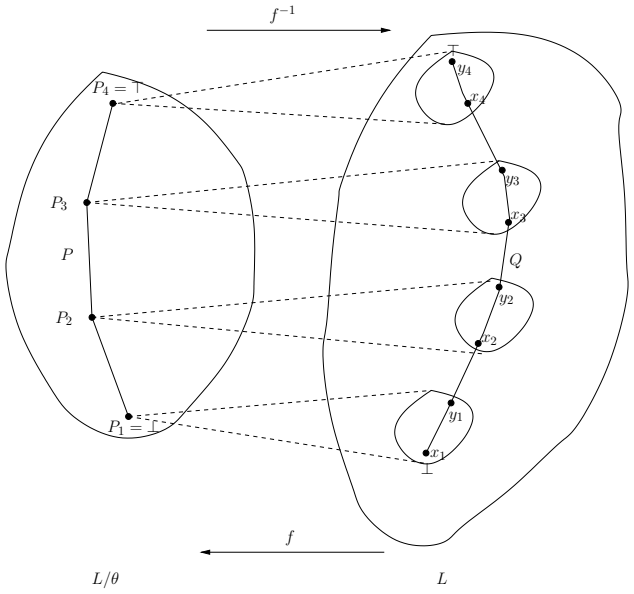



Figure 7.7: Proof of Lemma 7.13.

As a result, we get the following theorem.

**Theorem 7.14** *[?] A formula $\phi$ in $CTL_{-X}$ holds in $L$ iff it holds in $L/\theta$. where $\theta$ is the congruence class corresponding to all non-temporal subformulas in $\phi$.*

**Proof:** We can use induction on the structure of the formula and Lemma 7.13.

■

## 7.5 Problems

7.1. Let $L_1$ and $L_2$ be lattices and $f : L_1 \to L_2$ be a map.

(a) The following are equivalent:

i. $f$ is order-preserving;
ii. $(\forall a, b \in L) f(a \sqcup b) \geq f(a) \sqcup f(b)$;
iii. $(\forall a, b \in L) f(a \sqcap b) \geq f(a) \sqcap f(b)$.

In particular, if $f$ is a homomorphism, then $f$ is order preserving.

## 7.6 Bibliographic Remarks

The discussion on lattice homomorphisms and congruence lattices is quite standard. The reader is referred to [Grä03] for various proofs. The application of lattice congruences to model checking is taken from [**?**].

# Chapter 8

# Modular Lattices

## 8.1 Introduction

We describe a special class of lattices called modular lattices. Modular lattices are numerous in mathematics; for example, the lattice of normal subgroups of a group is modular, the lattice of ideals of a ring is modular, and so is the finite-dimensional vector space lattice. Distributive lattices are a special class of modular lattices. The, the set of all consistent global states in a distributed computation forms a distributive lattice and is therefore a modular lattice.

In this chapter we first introduce both modular and distributive lattices to show the relationship between them. Later, we focus on modular lattices. Distributive lattices are considered in detail in Chapter **??**.

## 8.2 Modular Lattice

**Definition 8.1** *$L$ is* **modular** *if* $\forall a, b, c : a \geq c \Rightarrow a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c$

The definition says that if $c \leq a$, then one can bracket the expression $a \sqcap b \sqcup c$ either way. Recall the definition of a distributive lattice :

**Definition 8.2** *$L$ is distributive if* $\forall a, b, c : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$

In the definition of distributive lattices, the equality can be replaced by $\leq$ because of the following observation.

**Lemma 8.3** *For all lattices,* $a \sqcap (b \sqcup c) \geq (a \sqcap b) \sqcup (a \sqcap c)$

**Proof:** $a \sqcap (b \sqcup c) \geq (a \sqcap b)$

$a \sqcap (b \sqcup c) \geq (a \sqcap c)$

Combining, we get

$a \sqcap (b \sqcup c) \geq (a \sqcap b) \sqcup (a \sqcap c)$

■

A similar observation follows from the definition of modular lattices.

**Lemma 8.4** *For all lattices* $a \geq c \Rightarrow a \sqcap (b \sqcup c) \geq (a \sqcap b) \sqcup c$

**Proof:**

$a \geq c \Rightarrow (a \sqcap c) = c$

Using Lemma 8.3 it follows that $a \sqcap (b \sqcup c) \geq (a \sqcap b) \sqcup c$.

■

**Theorem 8.5** *$L$ is distributive implies that $L$ is modular.*

**Proof:**

$L$ is distributive
$\equiv$ { definition of a distributive lattice }
$\forall a, b, c : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$
$\Rightarrow$ { $a \geq c$ is equivalent to $a \sqcap c = c$ }
$\forall a, b, c : a \geq c \Rightarrow a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c$
$\equiv${ definition of a modular lattice }
$L$ is modular.

■

## 8.3 Characterization of Modular Lattices

We first define two special lattices - the diamond lattice and the pentagon lattice. The **diamond lattice** ($M_3$) shown in Figure 8.1(b). $M_3$ is modular. It is, however, not distributive. To see this, note that in the diagram of $M_3$ we have:

$a \sqcap (b \sqcup c) = a \sqcap 1 = a$

and $(a \sqcap b) \sqcup (a \sqcap c) = 0 \sqcup 0 = 0$.

Since $a \neq 0$ , $M_3$ is not distributive. All lattices of four elements or less are modular. The smallest lattice which is not modular is the **pentagon** ($N_5$) which is shown in Figure 8.1(a).

In the figure, $a \geq c$ holds. However, $a \sqcap (b \sqcup c) = a \sqcap 1 = a$ but $(a \sqcap b) \sqcup c = 0 \sqcup c = c$.

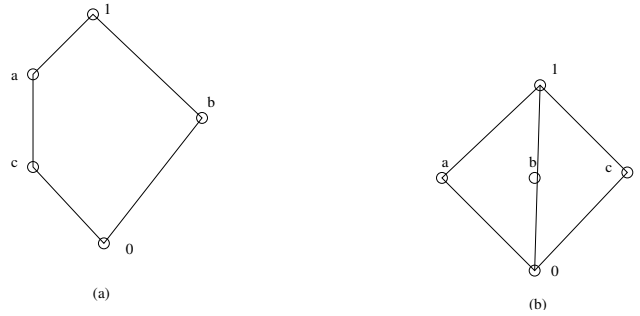



Figure 8.1: Diamond($M_3$ and Pentagon($N_5$).

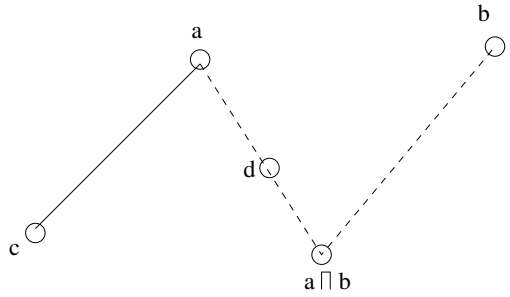



Figure 8.2:

We now focus on modular lattices and list some theorems which characterize modular lattices.

In the definition of modular lattices, if $c$ satisfies $(a \sqcap b) \leq c \leq a$, then we get that $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c = c$. The following theorem shows that to check modularity it is sufficient to consider $c$'s that are in the interval $[a \sqcap b, a]$.

**Theorem 8.6** *A lattice $L$ is modular iff* $\forall a, b \in L$

$$d \in [a \sqcap b, a] \Rightarrow a \sqcap (b \sqcup d) = d \tag{8.1}$$

**Proof:**

First, we show that (8.1) implies that $L$ is modular.

Suppose $c \leq a$ but $a \sqcap b \leq c$ is false. We define $d = c \sqcup (a \sqcap b)$. Clearly $d \in [a \sqcap b, a]$.

Consider $a \sqcap (b \sqcup d)$. Replacing the value of $d$ from (**??** we get

$$a \sqcap (b \sqcup (c \sqcup (a \sqcap b))) = c \sqcup (a \sqcap b) \tag{8.2}$$

Now consider $a \sqcap (b \sqcup c)$. Using the fact that $c \leq c \sqcup (a \sqcap b)$ we get

$$a \sqcap (b \sqcup c) \leq a \sqcap (b \sqcup (c \sqcup (a \sqcap b))) \tag{8.3}$$

Combining (8.2) and (**??** we get $a \sqcap (b \sqcup c) \leq c \sqcup (a \sqcap b)$. Rewriting the equation gives us $a \sqcap (b \sqcup c) \leq (a \sqcap b) \sqcup c$.

Since $a \sqcap (b \sqcup c) \geq (a \sqcap b) \sqcup c$ holds for all lattices (Lemma 8.4), we get

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c$$

Hence $L$ is modular, which is what we wanted to show.

Showing the other side, that modularity implies condition (8.1) is trivial, and is left as an exercise.

■

The following lemma is useful in proving the next theorem.

**Lemma 8.7** *For all lattices $L$, $\forall a, b, c$ : let $v = a \sqcap (b \sqcup c))$ and $u = (a \sqcap b) \sqcup c$. Then, $v > u \Rightarrow [(v \sqcap b) = (u \sqcap b)] \wedge [(v \sqcup b) = (u \sqcup b)]$.*

**Proof:** We show the first conjunct. The proof for the second is similar.
$a \sqcap b$
$=$
$\quad (a \sqcap b) \sqcap b$
$\leq$
$\quad [(a \sqcap b) \sqcup c)] \sqcap b$
$=$ { definition of $u$}
$\quad u \sqcap b$
$\leq \{v > u\}$
$\quad v \sqcap b$
$=$ { definition of $v$}
$\quad (a \sqcap (b \sqcup c)) \sqcap b$
$= \{b \leq b \sqcup c\}$
$\quad (a \sqcap b)$

■

We are now ready for another characterization of modular lattices.

**Theorem 8.8** *A lattice $L$ is modular iff it does not contain a sublattice isomorphic to $N_5$.*

**Proof:** If $L$ contains $N_5$, then it clearly violates modularity as shown earlier. Now assume that $L$ is not modular. Then, there exist $a, b, c$ such that $a > c$ and $a \sqcap (b \sqcup c) > (a \sqcap b) \sqcup c$. It can be verified that $b||a$ and $b||c$. For example, $b \leq a$ implies $(a \sqcap b) \sqcup c = b \sqcup c \leq a \sqcap (b \sqcup c)$. The other cases are similar.

We let $v = a \sqcap (b \sqcup c)$ and $u = (a \sqcap b) \sqcup c)$. We know that $v > u$. It is easy to verify that $b||u$ and $b||v$. For example, $b < v \equiv b < a \sqcap (b \sqcup c) \Rightarrow (b < a)$.

From Lemma 8.7, $u \sqcup b = v \sqcup b$ and $u \sqcap b = v \sqcap b$.

Thus, $u, v, b, u \sqcup b$, and $u \sqcap b$ form $N_5$.

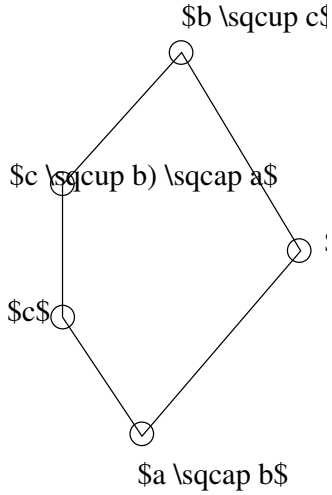



Figure 8.3: Proof of $N_5$ Theorem

■

**Theorem 8.9** *[Shearing Identity] A lattice $L$ is modular iff* $\forall x, y, z : x \sqcap (y \sqcup z) = x \sqcap ((y \sqcap (x \sqcup z)) \sqcup z)$.

**Proof:**

We first prove that modularity implies shearing. We use the fact that $x \sqcup z \geq z$.

$$\begin{aligned} & (y \sqcap (x \sqcup z)) \sqcup z \\ = \; & z \sqcup (y \sqcap (x \sqcup z)) \\ = \; & (z \sqcup y) \sqcap (x \sqcup z) \qquad \text{by modularity} \\ = \; & (y \sqcup z) \sqcap (x \sqcup z) \end{aligned}$$

Now consider the right hand side of the Shearing identity.

$$\begin{aligned} & x \sqcap ((y \sqcap (x \sqcup z)) \sqcup z) \\ = \; & x \sqcap ((y \sqcup z) \sqcap (x \sqcup z)) \\ = \; & x \sqcap (y \sqcup z) \qquad \text{since } x \leq x \sqcup z \end{aligned}$$

Showing that shearing implies modularity is left as an exercise.

■

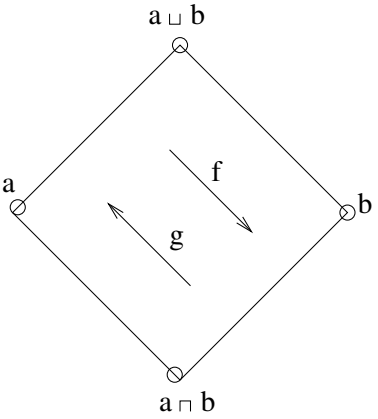



Figure 8.4:

**Theorem 8.10** *A lattice $L$ is modular iff $\forall a, b : f$ and $g$ as defined below are isomorphic.*

*Consider two incomparable elements $a$ and $b$ as shown in Figure 9.5. Then*

$$f : x \mapsto x \sqcap b$$
$$g : y \mapsto y \sqcup a$$

**Proof:**

First we show that modularity implies $f$ and $g$ are isomorphic. In order to show that $f$ and $g$ are isomorphic, we need to show that:

1. $f$ and $g$ are homomorphic.
2. they are bijective.

In order to show $f$ is a lattice homomorphism we need to show that $f$ is join-preserving and meet-perserving. That is

$$f(x_1 \sqcup x_2) = f(x_1) \sqcup f(x_2) \text{ and } f(x_1 \sqcap x_2) = f(x_1) \sqcap f(x_2)$$

**Claim**: $f$ and $g$ are monotone functions, that is, if $x_1 \leq x_2$ then $f(x_1) \leq f(x_2)$.

We will first show that $f$ is join-preserving, that is, $f(x_1 \sqcup x_2) = f(x_1) \sqcup f(x_2)$. Consider the right hand side

$$\begin{aligned} & f(x_1) \sqcup f(x_2) \\ &= (x_1 \sqcap b) \sqcup (x_2 \sqcap b) \text{ since } x_1 \leq x_2 \text{ and } f \text{ is monotone} \\ &= x_2 \sqcap b \end{aligned}$$

Now taking the left hand side, we get

$$\begin{aligned} & f(x_1 \sqcup x_2) \\ & = (x_1 \sqcup x_2) \sqcap b \text{ using } x_1 \leq x_2 \\ & = x_2 \sqcap b \end{aligned}$$

Hence, $f$ is join preserving. We now show that it is meet preserving too.

$$\begin{aligned} & f(x_1 \sqcap x_2) \\ & = (x_1 \sqcap x_2) \sqcap b \text{ using } x_1 \leq x_2 \\ & = x_1 \sqcap b \end{aligned}$$

Now working with the other side of the equation,

$$\begin{aligned} & f(x_1) \sqcap f(x_2) \\ & = (x_1 \sqcap b) \sqcap (x_2 \sqcap b) \text{ using } x_1 \leq x_2 \\ & = x_1 \sqcap b \end{aligned}$$

Therefore $f$ is both join-preserving and meet-preserving. We can show that $g$ is join-preserving and meet-preserving in a similar fashion. Hence $f$ and $g$ are lattice homomorphisims.

We now show that $f$ and $g$ are inverses of each other. Consider

$$\begin{aligned} & f \circ g(x) \\ & = g(f(x)) \\ & = (x \sqcap b) \sqcup a \\ & = x \sqcap (b \sqcup a) \text{ from modularity} \\ & = x \end{aligned}$$

Therefore $f = g^{-1}$. Hence $f$ and $g$ are bijective. This concludes our proof that $f$ and $g$ are lattice isomorphisms.

We will leave it as an exercise to show the reverse implication.

■

**Theorem 8.11** *A lattice $L$ is modular iff it satisfies the Upper Covering Condition, i.e., if $x$ and $y$ both cover $x \sqcap y$, then $x \sqcup y$ covers both $x$ and $y$.*

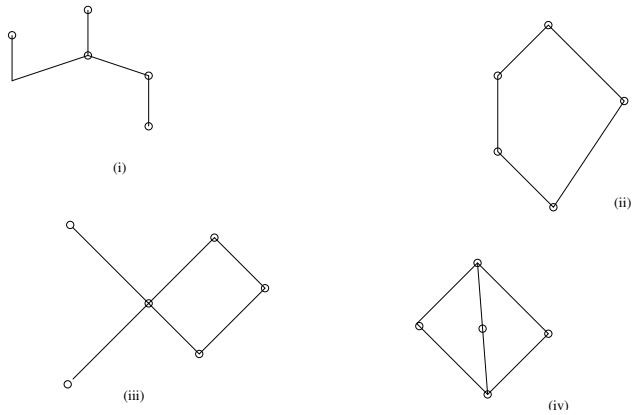



Figure 8.5: (i) a ranked poset, (ii) a poset which is not ranked, (iii) a ranked and graded poset, (iv) a ranked and graded lattice

**Proof:** We leave this proof as an exercise.

■

We define a ranked poset and a graded poset. Some examples are shown in Figure 8.5.

**Definition 8.12 (Ranked poset)** *A poset is ranked if there exists a ranking function* $r : P \mapsto N$ *such that* $r(x) = r(y) + 1$ *whenever* $x$ *covers* $y$*.*

**Definition 8.13 (Graded poset)** *A ranked poset is graded if all maximal chains have the same length.*

**Theorem 8.14** *A lattice* $L$ *is modular iff* $L$ *is graded and the ranking function satisfies*

$$r(x) + r(y) = r(x \sqcap y) + r(x \sqcup y)$$

**Proof:** The proof of the theorem is left as an exercise.

■

## 8.4 Problems

8.1. Show that if $L$ is a graded lattice, then

(a) $r(x) + r(y) \geq r(x \sqcap y) + r(x \sqcup y)$ is equivalent to the Upper Covering Condition.

(b) $r(x) + r(y) \leq r(x \sqcap y) + r(x \sqcup y)$ is equivalent to the Lower Covering Condition.

# Chapter 9

# Distributive Lattices

## 9.1 Distributive Lattices

A lattice $L$ is distributive if:

$$\forall a, b, c \in L : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c) \tag{9.1}$$

Given a modular lattice, the following theorem is useful in determining if the lattice is distributive.

**Theorem 9.1 (Diamond)** *A modular lattice is distributive iff it does not contain a diamond as a sublattice.*

**Proof:** ($\Rightarrow$): if the lattice contains a diamond, it cannot be distributive because elements of a diamnod do not satisfy distributve law. In Figure 9.1, $x \sqcap (y \sqcup z) = x \sqcap 1 = x$ however, $(x \sqcap y) \sqcup (x \sqcap z) = 0 \sqcup 0 = 0$.

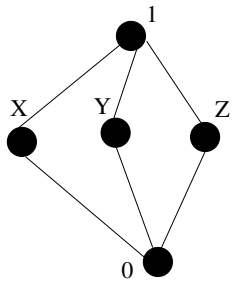



Figure 9.1: Diamond: non-distributive lattice

($\Leftarrow$) It is sufficient to prove that every modular non-distributive lattice contains $M_3$ as a sublattice. Let $x, y, z$ be such that $x \sqcap (y \sqcup z) > (x \sqcap y) \sqcup (x \sqcap z)$.

Define elements $a, b, c, d, e$ as follows: $a := (x \sqcap y) \sqcup (y \sqcap z) \sqcup (z \sqcap x)$
$b := (x \sqcup y) \sqcap (y \sqcup z) \sqcap (z \sqcup x)$
$c := (x \sqcap b) \sqcup a$
$d := (y \sqcap b) \sqcup a$
$e := (z \sqcap b) \sqcup a$

We leave it for the reader to verify that these elements form a diamond.

■

The above theorem along with the pentagon theorem suggests that a lattice is modular and distributive iff it neither contains a pentagon nor a diamond. This fact gives us a brute force algorithm to determine if a lattice is distributive or modular by enumerating all its sublattices of size 5.

## 9.2 Properties of Join-Irreducibles

The following lemma is used in the proof of Birkhoff's theorem and gives an an alternate characterization of join-irreducibles in distributive lattices.

**Lemma 9.2** *In a finite distributive lattice (FDL) $L$, let $x$ be any nonzero element. Then, $x$ is join-irreducible if and only if*

$$\forall a, b \in L : x \leq a \sqcup b \Rightarrow (x \leq a \vee x \leq b). \tag{9.2}$$

**Proof:**

$$\begin{array}{ll} & x \leq a \sqcup b \\ \Leftrightarrow & \langle \text{property of } \leq \text{ and } \sqcap \rangle \\ & x = x \sqcap (a \sqcup b) \\ \Leftrightarrow & \langle L \text{ distributive} \rangle \\ & x = (x \sqcap a) \sqcup (x \sqcap b) \\ \Rightarrow & \langle x \text{ join-irreducible} \rangle \\ & (x = x \sqcap a) \vee (x = x \sqcap b) \\ \Leftrightarrow & \langle \text{property of } \leq \text{ and } \sqcap \rangle \\ & (x \leq a) \vee (x \leq b). \end{array}$$

Conversely, assume (9.2), and let $x = a \sqcup b$. Then $x \leq a \sqcup b$, hence $x \leq a \vee x \leq b$. On the other hand, $x = a \sqcup b$ implies $x \geq a \wedge x \geq b$. From the last two, it follows that $x = a \vee x = b$.

■

Note the the backward direction does not require distributivity of $L$.

**Corollary 9.3** . *In a finite distributive lattice $L$, an element $x$ is join-irreducible if and only if*

$$\forall nonemptyV \subseteq L : x \leq \bigsqcup V \Rightarrow \exists v : v \in V : x \leq v. \tag{9.3}$$

**Proof:** If $x$ is join-irreducible, then for any nonempty $V \subseteq L$, property (9.3) is proven by induction on $|V|$. The case when $V$ is singleton is trivial and the case ($|V| = 2$) makes use of lemma 9.2. Converse holds trivially.

■

## 9.3 Birkhoff's Representation Theorem for FDLs

The main theorem of this chapter relates FDLs to posets:

**Theorem 9.4** *Let $L$ be a FDL. Then the following mapping from $L$ to the set of ideals of $J(L)$ is an isomorphism:*

$$f : L \to O(J(L)), \qquad a \in L \mapsto f(a) = \{x \in J(L) : x \leq a\}.$$

**Proof:** First note that $f(a) \in O(J(L))$ because $\leq$ is transitive.

Secondly, $f$ is injective (one-one). If $f(a) = f(b)$, then $\bigsqcup f(a) = \bigsqcup f(b)$, hence $a = b$ by lemma 5.5.

Thirdly, $f$ is surjective (onto). Let $V \in O(J(L))$. Consider $a := \bigsqcup V$; we will show that $f(a) = V$. We first show that $f(a) \subseteq V$.

$x \in f(a)$
$\equiv$ { definition of $f(a)$ }
$x$ is join-irreducible and $x \leq a \equiv$ { definition of $a$ }
$x$ is join-irreducible and $x \leq \bigsqcup V$
$\Rightarrow$ { Corollary 9.3 }
$x$ is join-irreducible and $\exists y \in V : x \leq y$
$\Rightarrow$ { $V$ is an order ideal }
$x \in V$

We now show that $V \subseteq f(a)$. $v \in V$
$\Rightarrow$ { definition of $V$, and $\bigsqcup$ }
$v$ is join-irreducible and $v \leq \bigsqcup V$
$\Rightarrow$ { definition of $a$ }
$v$ is join-irreducible and $v \leq a$
$\Rightarrow$ { definition of $f(a)$ }
$v \in f(a)$.

Lastly, we show that $f$ preserves the order: if $a \leq b$, then $x \leq a$ implies $x \leq b$, hence $f(a) \subseteq f(b)$. Conversely, $f(a) \subseteq f(b)$ implies $\bigsqcup f(a) \leq \bigsqcup f(b)$, which by lemma 5.5 reduces to $a \leq b$.

■

## Interpretation and Examples

Birkhoff's theorem establishes an isomorphism relationship between a given finite distributive lattice $L$ and the set of order ideals of a particular poset, $P$, namely the poset of join-irreducible elements of the lattice. In general, the poset $P$ is obtained from $L$ as the set of $L$'s join-irreducibles, and $L$ is recovered from $P$ *up to isomorphism* by taking the set of ideals of elements from $P$. The significance of this theorem lies in the fact that the poset of join-irreducibles of $L$ can be exponentially more succinct than the lattice itself.

Figure 9.2 shows a five-element lattice with three join-irreducibles $a$, $b$, $c$. The set of ideals of the join-irreducibles on the right is isomorphic to the original lattice, which means in particular that the two cover graphs are isomorphic in the graph-theory sense. To recover the names of elements in $L$

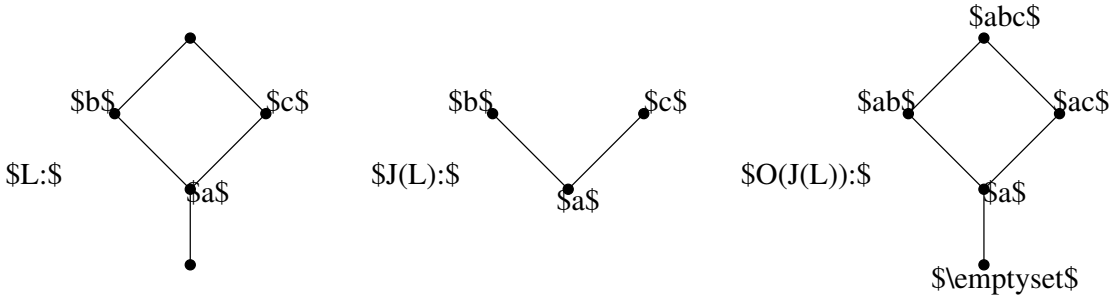



Figure 9.2: A lattice $L$, its set of join-irreducibles $J(L)$, and their ideals $O(J(L))$

from those in $O(J(L))$, one computes the join in $J(L)$ of the corresponding ideal, i.e. $f^{-1} = \bigsqcup$. For example, $f^{-1}(ab) = \bigsqcup\{a, b\} = b$. If $\bigsqcup Z$ is undefined in $J(L)$ for an ideal $Z$, then the corresponding element $f^{-1}(Z)$ of $L$ is not join-irreducible and hence does not appear in $J(L)$, so its name cannot be recovered.

Another example is given in figure 9.3.

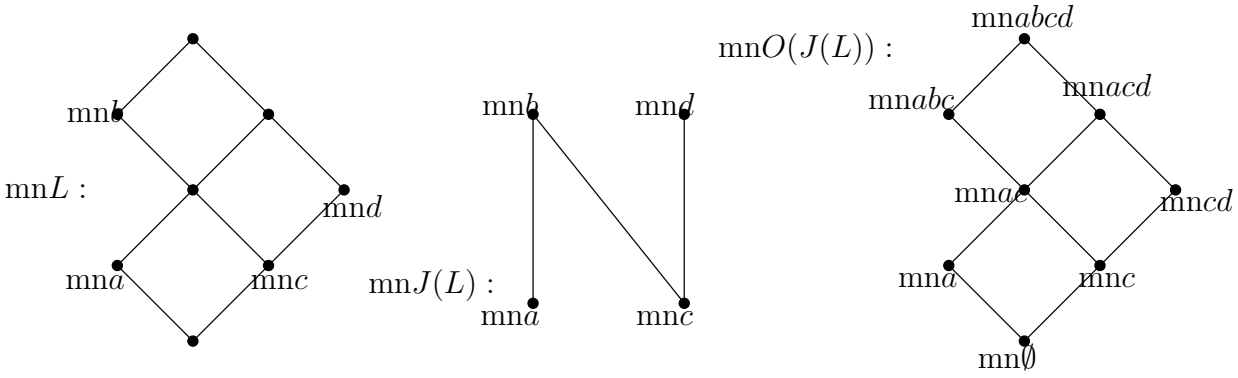



Figure 9.3: A lattice with the "N-poset" as set of join-irreducibles

Birkhoff's Theorem also makes it possible to translate the numerous properties of FDL $L$ into those of poset $P$, and vice-versa. For example, the dimension of $L$ is equal to the width of a poset $P$, and the height of $L$ equals the cardinality of $P$. It can also be shown easily that $L$ is a boolean

lattice iff $P$ is an antichain, and $L$ is a chain iff $P$ is a chain. We have left such claims as exercises to the reader.

Although we have used join-irreducibles in Birkhoff's theorem, the argument can also be carried out with meet-irreducibles. We leave the following lemma as an exercise.

**Lemma 9.5** *For a finite distributive lattice $L$, the sub-posets containing $J(L)$ and $M(L)$ are isomorphic.*

Figure 9.4 (b) shows that the statement is not true for a non-distributive lattice: not even the cardinalities of $J(L)$ and $M(L)$ are equal. The lattice in (a), on the other hand, is distributive, and indeed $J(L)$ and $M(L)$ are isomorphic.

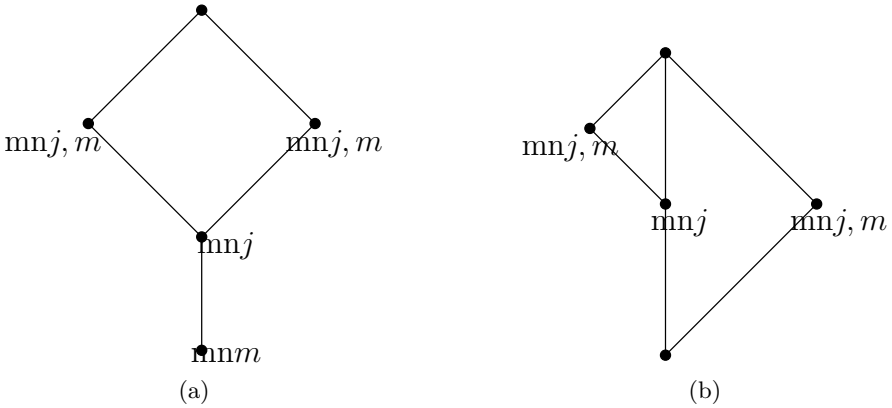



Figure 9.4: (a) Join- ("$j$") and meet- ("$m$") irreducible elements; (b) Example for lemma 9.5

## 9.4 Finitary Distributive Lattices

Birkhoff's theorem requires the distributive lattice to be finite. In an infinite distributive lattices there may not be any join-irreducible elements. For example, the lattice of open subsets on real-line does not have any join-irreducible elements. In the infinite case, prime ideals serve as the "basis" elements. We refer to [DP90] to reader for details on that approach.

In computer science applications (such as distributed computing), it is more common to get finitary distributive lattices. A finitary distributive lattice is a locally finite distributive lattice with the bottom element. By following the proof of Birkhoff's theorem it can be readily verified that it can be extended to finitary lattices in the following sense.

**Theorem 9.6** *Let $P$ be a poset such that every principal order ideal is finite. Then the poset of finite order ideals of $P$ is a finitary distributive lattice. Conversely, if $L$ is a finitary distributive lattice and $P$ is its subposet of join-irreducibles, then every principal order ideal of $P$ is finite and $L$ is isomorphic to the lattice of all finite order ideals of $P$.*

## 9.5 Problems

1. We saw in Chapter **??** that join-irreducible elements may not always be dissectors. Show that for distributive lattices all join-irreducible elements are also dissectors.

2. Show that the poset of join-irreducible elements of a FDL $L$ is isomorphic to the poset of meet-irreducible elements of $L$.

3. Show that height of a FDL $L$ equals the size of the subposet $J(L)$.

4. Show that for a finite lattice $L$, the following are equivalent:

   (1) $L$ is distributive.

   (2) $L$ is isomorphic to the set of ideals of its join-irreducibles: $L \approx O(J(L))$.

   (3) $L$ is isomorphic to a lattice of sets (see definition below).

   (4) $L$ is isomorphic to a sub-lattice of a boolean lattice.

# Chapter 10

# Slicing

In this chapter, we are interested in producing structures that generate subsets of the finite distributive lattice. It is more convenient to use directed graphs instead of posets for this purpose because we can get sublattices by simply adding edges to the original directed graph (this will be shown later).

The notion of order ideals of a poset can be extended to graphs in a straightforward manner. A subset of vertices, $H$, of a directed graph is an *ideal* if it satisfies the following condition: if $H$ contains a vertex $v$ and $(u, v)$ is an edge in the graph, then $H$ also contains $u$. We will continue to use $P$ for the directed graph. Observe that an ideal of $P$ either contains all vertices in a strongly connected component or none of them. Let $\mathcal{I}(P)$ denote the set of ideals of a directed graph $P$. Observe that the empty set and the set of all vertices trivially belong to $\mathcal{I}(P)$. We call them **trivial** ideals. The following theorem is a slight generalization of the result that the set of ideals of a partially ordered set forms a distributive lattice.

**Theorem 10.1** *Given a directed graph $P$, $(\mathcal{I}(P); \subseteq)$ forms a distributive lattice.*

Observe that when the directed graph has no edges (i.e., the poset is an antichain), the set of ideals correspond to the boolean lattice on the set of vertices. At the other extreme, when the graph is strongly connected, there are only trivial ideals. Since trivial ideals are always part of $\mathcal{I}(P)$, it is more convenient to deal only with nontrivial ideals of a graph. It is easy to convert a graph $P$ to $P'$ such that there is one-to-one correspondence between all ideals of $P$ and all nontrivial ideals of $P'$. We construct $P'$ from $P$ by adding two additional vertices $\bot$ and $\top$ such that $\bot$ is the smallest vertex and $\top$ is the largest vertex (i.e., there is a path from $\bot$ to every vertex and a path from every vertex to $\top$). It is easy to see that any nontrivial ideal will contain $\bot$ and not contain $\top$. As a result, every ideal of $P$ is a nontrivial ideal of the graph $P'$ and vice versa. We will deal with only nontrivial ideals from now on and an ideal would mean nontrivial ideal unless specified otherwise. The directed graph representation of Figure 10.1(a) is shown in Figure 10.1(c).

Figure 10.2 shows a directed graph and its nontrivial ideals. The directed graph in Figure 10.2(a) is derived from Figure 10.1(a) by adding an edge from $c$ to $b$ and adding two additional vertices

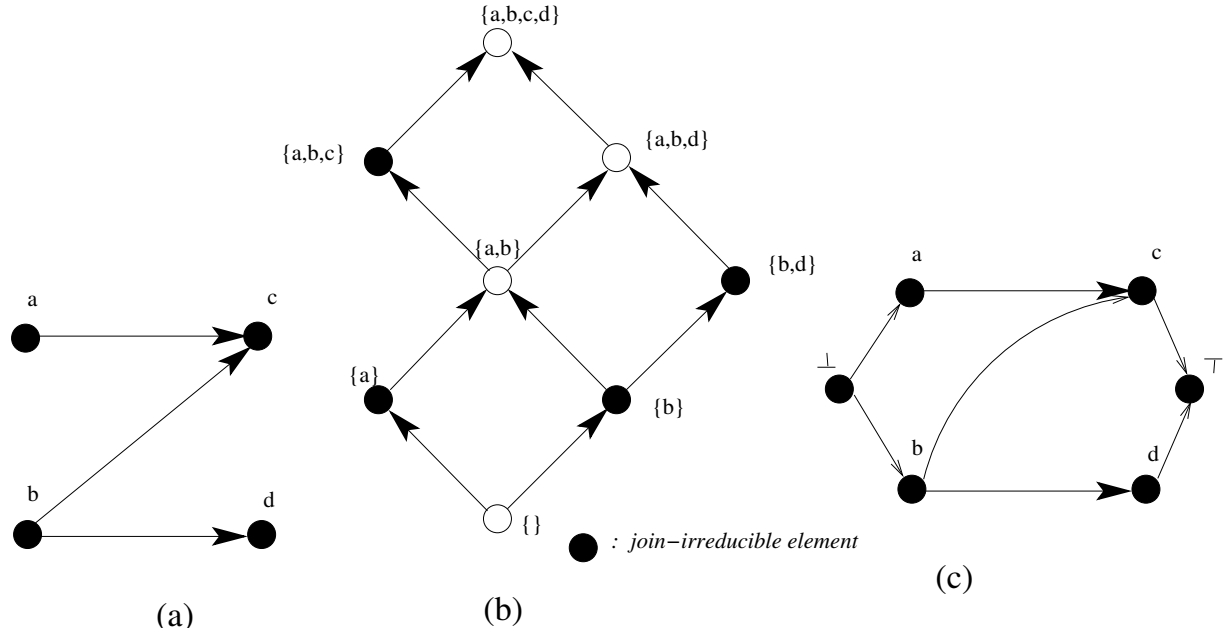



Figure 10.1: (a) a partial order (b) the lattice of ideals. (c) the directed graph

$\perp$ and $\top$. The resulting set of nontrivial ideals is a sublattice of Figure 10.1(b). In the figure, we have not shown $\perp$ in the ideals because it is implicitly included in every nontrivial ideal.

We use the graph in Figure 14.8 as a running example in the paper and denote it by $\mathcal{K}$ . Let $X = \{1..n\}$ and $k$ be a number between 1 and $n$ (inclusive). Ignoring the vertices $\top$ and $\perp$, $\mathcal{K}$ has $k$ chains and each chain has $n - k$ elements. We call these chains $P_1, \ldots, P_k$. In the graph, there are two types of edges. For every chain there is an edge from $j^{th}$ element to $(j + 1)^{th}$ element, and an edge from $j^{th}$ element in $P_{i+1}$ to $j^{th}$ element in $P_i$. Therefore, if an ideal of $P$ contains $j$ elements of $P_i$, then it also contains at least $j$ elements of $P_{i+1}$. We show that the ideals of $\mathcal{K}$ are in 1-1 correspondence with all the subsets of $X$ of size $k$. The correspondence between subsets of $X$ and ideals can be understood as follows. If chain $P_i$ has $t$ elements in the ideal, then the element $t + i$ is in the set $Y$. Thus chain $P_1$ chooses a number from $1 \ldots n - k + 1$ (because there are $n - k$ elements); chain $P_2$ chooses the next larger number and so on. Figure 14.8(b) gives an example of the graph for subsets of size 3 of the set [6]. The ideal, shown corresponds to the subset $\{1, 3, 4\}$. It can also be easily verified that there are $\binom{n}{k}$ ideals of this poset.

## 10.1 Predicates on Ideals

Our technique is crucially based on the notion of predicates on the set of ideals. A predicate is simply a boolean function from the set of all ideals to $\{0, 1\}$. Equivalently, a predicate specifies a subset of the ideals in which the boolean function evaluates to 1. In the poset $\mathcal{K}$ , "does not contain consecutive numbers" is a predicate which is either true or false for any ideal. For example, it is true for $\{1, 3, 5\}$ but false for $\{1, 3, 4\}$. In the boolean lattice of all subsets of $[n]$, $\mathcal{B}_n$, "has size $k$" is a predicate which is true if the subset has size $k$ and false otherwise.

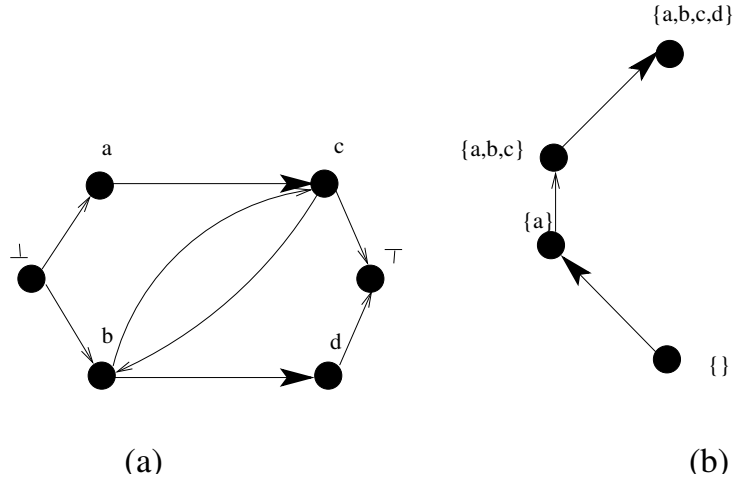



Figure 10.2: (a) A directed graph (b) the lattice of its nontrivial ideals.

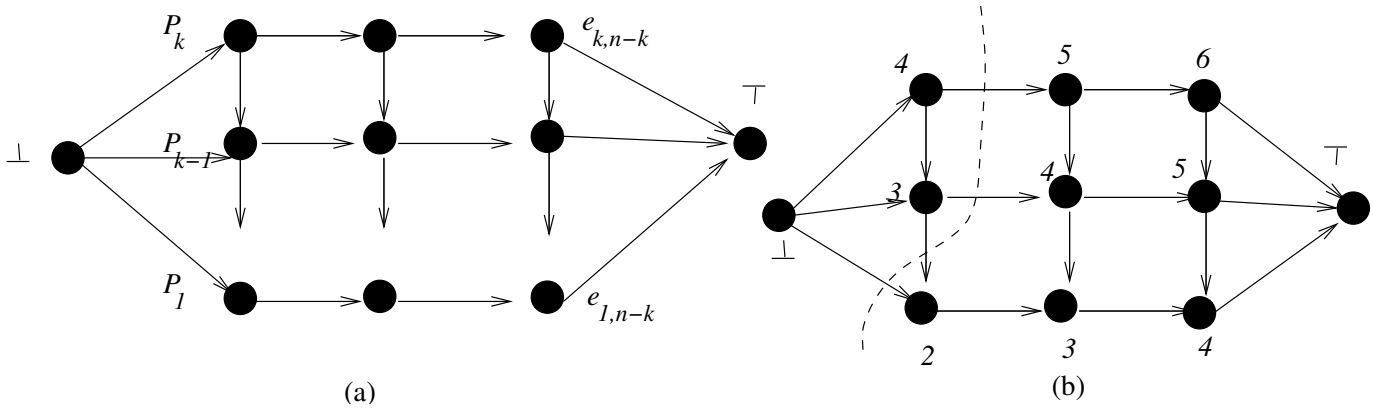



Figure 10.3: (a) The poset or directed graph $\mathcal{K}$ for generating subsets of $X$ of size $k$. (b) The ideal denoting the subset $\{1, 3, 4\}$.

We now define various classes of predicates. The class of meet-closed predicates are useful because they allow us to compute the least ideal that satisfies a given predicate.

**Definition 10.2 (meet-closed Predicates)** *A predicate $B$ is meet-closed for a poset or a graph $P$ if*

$$\forall G, H \in \mathcal{I}(P) : B(G) \wedge B(H) \Rightarrow B(G \sqcap H)$$

The predicate "does not contain consecutive numbers" in the poset $\mathcal{K}$ is meet-closed whereas the predicate "has size $k$" in the poset $\mathcal{B}_n$ is not.

It follows from the definition that if there exists any ideal that satisfies a meet-closed predicate $B$, then there exists the least one. Note that the predicate *false* which corresponds to the empty subset and the predicate *true* which corresponds to the entire set of ideals are meet-closed predicates.

As another interesting example, define a $n$-size subset of $[2n]$ to satisfy the predicate Catalan if by putting left parentheses on the specified subset and right parentheses in remaining positions,

one gets a well-balanced string of parentheses. For example, the subset $\{1, 3, 4\}$ corresponds to the string "()(())" where left parenthesis are at positions 1, 3 and 4. This subset satisfies Catalan predicate whereas the subset $\{1, 4, 5\}$ does not. We will use $G[i]$ to denote the $i$th largest element in the set. To formalize a well-balanced expression, note that if $G[i]$ equals $k$ for any $i \in [n]$, then there are $i - 1$ left parentheses and $k - i$ right parentheses to the left of position $k$. Therefore $G$ satisfies Catalan predicate iff for all $i$

$$G[i] - i \leq i - 1.$$

Or, equivalently

$$G[i] \leq 2i - 1.$$

It is now clear that Catalan is a meet-closed (and join-closed) predicate. If $G[i] \leq 2i - 1$ and $H[i] \leq 2i - 1$, then $min(G[i], H[i]) \leq 2i - 1$.

We now give another characterization of such predicates that will be useful for computing the least ideal that satisfies the predicate. To that end, we first define the notion of a crucial element for an ideal.

**Definition 10.3 (Crucial Element)** *Let $(E, \leq)$ be a poset. For an ideal $G \subsetneqq E$ and a predicate $B$, we define $e \in E - G$ to be crucial for $G$ as:*

$$crucial(G, e, B) \stackrel{\text{def}}{=} \forall \text{ ideals } H \supseteq G : (e \in H) \vee \neg B(H).$$

Intuitively, this means that any ideal $H$, that is at least $G$, cannot satisfy the predicate unless it contains $e$.

**Definition 10.4 (Linear Predicates)** *A predicate $B$ is linear iff for all ideals $G \subsetneqq E$,*

$$\neg B(G) \Rightarrow \exists e \in E - G : crucial(G, e, B).$$

Now, we have

**Theorem 10.5** *A predicate $B$ is linear iff it is meet-closed.*

**Proof:** First assume that $B$ is not closed under meet. We show that $B$ is not linear. Since $B$ is not closed under meets, there exist two ideals $H$ and $K$ such that $B(H)$ and $B(K)$ but not $B(H \sqcap K)$. Define $G$ to be $H \sqcap K$. $G$ is a strict subset of $H \subseteq E$ because $B(H)$ but not $B(G)$. Therefore, $G$ cannot be equal to $E$. We show that $B$ is not linear by showing that there does not exist any crucial element for $G$. A crucial element $e$, if it exists, cannot be in $H - G$ because $K$ does not contain $e$ and still $B(K)$ holds. Similarly, it cannot be in $K - G$ because $H$ does not contain $e$ and still $B(H)$ holds. It also cannot be in $E - (H \cup K)$ because of the same reason. We conclude that there does not exist any crucial element for $G$.

Now assume that $B$ is not linear. This implies that there exists $G \subsetneqq E$ such that $\neg B(G)$ and none of the elements in $E - G$ is crucial. We first claim that $E - G$ cannot be a singleton. Assume if possible $E - G$ contains only one element $e$. Then, any ideal $H$ that contains $G$ and does not contain $e$ must be equal to $G$ itself. This implies that $\neg B(H)$ because we assumed $\neg B(G)$. Therefore, $e$ is

crucial contradicting our assumption that none of the elements in $E-G$ is crucial. Let $W = E-G$. For each $e \in W$, we define $H_e$ as the ideal that contains $G$, does not contain $e$ and still satisfies $B$. It is easy to see that $G$ is the meet of all $H_e$. Therefore, $B$ is not meet-closed because all $H_e$ satisfy $B$, but not their meets.

■

**Example 10.6** Consider the poset of $n$ elements $\{1..n\}$ in which all elements are incomparable. Figure 14.7(a) shows the graph for this poset. It is clear that the ideals of this graph correspond to subsets of $[n]$ and thus generate the boolean lattice. Now consider the predicate $B$ defined to be true on $G$ as "If $G$ contains any odd $i < n$, then it also contains $i + 1$." It is easy to verify that $B$ is meet-closed. Given any $G$ for which $B$ does not hold, the crucial elements consist of

$$\{i | i \ even, 2 \leq i \leq n, i-1 \in G, i \notin G\}$$

**Example 10.7** Consider the poset $\mathcal{K}$ . Let the predicate $B$ be "G does not contain any consecutive numbers." Consider any $G$ for which $B$ does not hold. This means that it has some consecutive numbers. Let $i$ be the smallest number such that $i-1$ is also in $G$. If $i$ is on chain $P_j$, then the next element in $P_j$ is crucial. For example, the ideal $\{1, 3, 4\}$ does not satisfy $B$. The smallest element whose predecessor is also in $G$ is 4 which is on the chain $P_3$. Therefore, the second element in $P_3$ is crucial. In other words any ideal that satisfies $B$ and is bigger than $\{1, 3, 4\}$ is at least $\{1, 3, 5\}$. If $i$ is the last element in $P_j$, then any element in $E - G$ can serve as a crucial element because in this case for any $H$ that contains $G$, $B(H)$ is false.

Our interest is in detecting whether there exists an ideal that satisfies a given predicate $B$. We assume that given an ideal, $G$, it is efficient to determine whether $B$ is true for $G$ or not. On account of linearity of $B$, if $B$ is evaluated to be false in some ideal $G$, then we know that there exists a crucial element in $E-G$. We now make an additional assumption called the efficient advancement property.

**(Efficient Advancement Property)** There exists an efficient (polynomial time) function to determine the crucial element.

We now have

**Theorem 10.8** *If $B$ is a linear predicate with the efficient advancement property, then there exists an efficient algorithm to determine the least ideal that satisfies $B$ (if any).*

**Proof:** An efficient algorithm to find the *least* cut in which $B$ is true is given in Figure 10.4. We search for the least ideal starting from the ideal $\{\perp\}$. If the predicate is false in the ideal, then we find the crucial element using the efficient advancement property and then repeat the procedure.

■

```
(1) boolean function detect(B:boolean_predicate, P:graph)
(2) var
(3)       G: ideal initially G = {⊥};
(4)
(5) while (¬B(G) ∧ (G ≠ P)) do
(6)       Let e be such that crucial(e, G, B) in P;
(7)       G := G ∪ {e}.
(8) endwhile;
(9) if B(G) return true;
(10)else return false;
```

Figure 10.4: An efficient algorithm to detect a linear predicate

Assuming that $crucial(e, G, B)$ can be evaluated efficiently for a given graph, we can determine the least ideal that satisfies $B$ efficiently even though the number of ideals may be exponentially larger than the size of the graph. As an example, to find the least ideal in $\mathcal{K}$ that satisfies "does not contain consecutive numbers," we start with the ideal $\{1, 2, 3\}$. Since 1 and 2 are consecutive, we advance along $P_2$ to the ideal $\{1, 3, 4\}$ which still does not satisfy the predicate. We now advance along $P_3$ to the ideal $\{1, 3, 5\}$ which is the smallest ideal that satisfies the given predicate.

So far we have focused on meet-closed predicates. All the definitions and ideas carry over to join-closed predicates. If the predicate $B$ is join-closed, then one can search for the largest ideal that satisfies $B$ in a fashion analogous to finding the least ideal when it is meet-closed.

Predicates that are both meet-closed and join-closed are called regular predicates.

**Definition 10.9 (Regular Predicates [GM01])** *A predicate is regular if the set of ideals that satisfy the predicate forms a sublattice of the lattice of ideals.*

Equivalently, a predicate $B$ is *regular* with respect to $P$ if it is closed under $\sqcup$ and $\sqcap$, i.e.,

$$\forall G, H \in \mathcal{I}(P) : B(G) \wedge B(H) \Rightarrow B(G \sqcup H) \wedge B(G \sqcap H)$$

The set of ideals that satisfy a regular predicate forms a sublattice of the lattice of all ideals. Since a sublattice of a distributive lattice is also distributive, the set of ideals that satisfy a regular predicate forms a distributive lattice. From Birkhoff's theorem we know that a distributive lattice can be equivalently represented using the poset of its join-irreducible elements. The poset of join-irreducible elements provides a compact way of enumerating all ideals that satisfy the predicate and will be useful in efficient solving of combinatorial problems.

There are two important problems with this approach. First, what if the predicate is not regular? Is it still possible to represent the set of ideals satisfying $B$ compactly? Second, we need to calculate the structure that captures the set of ideals satisfying $B$ efficiently. These problems are addressed in the next section.

The notion of slice will be used to represent the subset of ideals that satisfy $B$ in a concise manner. The slice of a directed graph $P$ with respect to a predicate $B$ (denoted by $slice(P, B)$) is a graph

derived from $P$ such that all the ideals in $\mathcal{I}(P)$ that satisfy $B$ are included in $\mathcal{I}(slice(P, B))$. Note that the slice may include some additional ideals which do not satisfy the predicate. Formally,

**Definition 10.10 (Slice [MG01])** *A slice of a graph $P$ with respect to a predicate $B$ is the directed graph obtained from $P$ by adding edges such that:*
*(1) it contains all the ideals of $P$ that satisfy $B$ and*
*(2) of all the graphs that satisfy (1), it has the least number of ideals.*

We first show that given a distributive lattice $L$ generated by the graph (poset) $P$, every sublattice of $L$ can be generated by a graph obtained by adding edges to $P$.

**Theorem 10.11** *Let $L$ be a finite distributive lattice generated by the graph $P$. Let $L'$ be any sublattice of $L$. Then, there exists a graph $P'$ that can be obtained by adding edges to $P$ that generates $L'$.*

**Proof:** Our proof is constructive. We show an algorithm to compute $P'$. For every vertex $e \in P$, let $I(e)$ be the set of ideals of $P$ containing $e$. Since $L$ is the set of all ideals of $P$, we can view $I(e)$ as a subset of $L$. In rest of the proof, we will not distinguish between an ideal of the graph $P$ and the corresponding element in lattice $L$. Let $I(e, L')$ be the subset that is contained in $L'$, i.e., $I(e, L') = I(e) \cap L'$. Based on the set $I(e, L')$, we define $J(e, L')$ as follows. If $I(e, L')$ is empty then $J(e, L')$ is defined to be the trivial ideal of $P$ that includes the $\top$ element. Otherwise, $J(e, L')$ is defined as the least element of $I(e, L')$. Since $L'$ is a sublattice, it is clear that if the set of ideals that include $e$ are in $L'$, then their intersection (meet) also includes $e$ and is in $L'$. Thus, $J(e, L')$ is well defined.

Note that $J(\bot, L')$ corresponds to the least element of $L'$ and $J(\top, L')$ corresponds to the trivial ideal that includes $\top$.

Now we add the following edges to the graph $P$. For every pair of vertices $e, f$ such that $J(e, L') \leq J(f, L')$, we add an edge from $e$ to $f$. We now claim that the resulting graph $P'$ generates $L'$.

Pick any nontrivial ideal $G$ of $P'$, i.e., an ideal that includes $\bot$ and does not include $\top$. We show that $G = \cup_{e \in G} J(e, L')$. This will be sufficient to show that $G \in L'$ because $G$ is a union of ideals in $L'$ and $L'$ is a lattice. Since $e \in J(e, L')$ it is clear that $G \subseteq \cup_{e \in G} J(e, L')$. We show that $G \supseteq \cup_{e \in G} J(e, L')$. Let $f \in J(e, L')$ for some $e$. This implies that $J(f, L') \subseteq J(e, L')$ because $J(f, L')$ is the least ideal containing $f$ in $L'$. By our algorithm, there is an edge from $f$ to $e$ in $P'$, and since $G$ is an ideal of $P'$ that includes $e$, it also includes $f$. Thus $G \supseteq \cup_{e \in G} J(e, L')$.

Conversely, pick any element $G$ of $L'$. We show that $G$ is a nontrivial ideal of $P'$. Since $L' \subseteq L$ and $L$ corresponds to nontrivial ideals of $P$, it is clear that $G$ is a nontrivial ideal of $P$. Our obligation is to show that it is a nontrivial ideal of $P'$ as well. Assume, if possible, $G$ is not a nontrivial ideal of $P'$. This implies that there exists vertices $e, f$ in $P'$ such that $f \in G$, $e \notin G$ and $(e, f)$ is an edge in $P'$. The presence of this edge in $P'$, but not in $P$ implies that $J(e, L') \subseteq J(f, L')$. Since $f \in G$ and $G \in L'$, from definition of $J(f, L')$, we get that $J(f, L') \subseteq G$. But this implies $J(e, L') \subseteq G$, i.e., $e \in G$, a contradiction.

■

Now, the following result is easy.

**Theorem 10.12** *For any $P$ and $B$, slice$(P, B)$ exists and is unique.*

**Proof:** First note that intersection of sublattices is also a sublattice. Now given any predicate $B$ consider all the sublattices that contain all the ideals that satisfy $B$. The intersection of all these sublattices gives us the smallest sublattice that contains all the ideals. From Theorem 10.11, we get that there exists a graph that generates this sublattice.

■

The procedure outlined in the above proof is not efficient because it requires us to take intersection of all bigger sublattices. We now show how to efficiently compute slices for the predicates for which there exists an efficient detection algorithm. The slicing algorithm is shown in Figure **??**. It takes as input a graph $P$ and a boolean predicate $B$. The algorithm constructs the slice by adding edges to the graph $P$. For this purpose, it initializes in line (3) a graph $R$ as $P$. In rest of the function, edges are added to $R$ which is finally returned.

The addition of edges is done as follows. For every pair of vertices, $e$ and $f$, in the graph $P$, the algorithm constructs $Q$ from $P$ by adding edges from $f$ to $\bot$ and $\top$ to $e$. Due to these edges in $Q$, all nontrivial ideals of $Q$ contain $f$ and do not contain $e$. We now invoke the detection algorithm on $Q$. If the detection algorithm returns false, then we know that there is no nontrivial ideal of $P$ that contains $f$ but does not contain $e$. Therefore, all ideals that satisfy $B$ have the property that if they include $f$, they also include $e$. Hence, we add an edge from $e$ to $f$ in the graph $R$. We continue this procedure for all pairs of vertices. Theorem 10.13 shows the correctness of this procedure.

**Theorem 10.13** *Let $P$ be a directed graph. Let $R$ be the directed graph output by the algorithm in Figure* **??** *for a predicate $B$. Then $R$ is the slice of $P$ with respect to $B$.*

**Proof:** Let $\mathcal{I}(P, B)$ denote the set of ideals of $P$ that satisfy $B$. We first show that $\mathcal{I}(R) \supseteq \mathcal{I}(P, B)$. Adding an edge $(e, f)$ in $R$ eliminates only those ideals of $P$ that contain $f$ but do not contain $e$. But all those ideals do not satisfy $B$ because the edge $(e, f)$ is added only when $detect(B, Q)$ is false. Thus, all the ideals of $P$ that satisfy $B$ are also the ideals of $R$.

Next we show that the $\mathcal{I}(R)$ is the smallest sublattice of $\mathcal{I}(P)$ that includes $\mathcal{I}(P, B)$. Let $M$ be a graph such that $\mathcal{I}(M) \supseteq \mathcal{I}(P, B)$. Assume if possible $\mathcal{I}(M)$ is strictly smaller than $\mathcal{I}(R)$. This implies that there exists two vertices $e$ and $f$ such that there is an edge from $e$ to $f$ in $M$ but not in $R$. Since $R$ is output by the algorithm, $detect(B, Q)$ is true in line (7); otherwise, an edge would have been added from $e$ to $f$. But, this means that there exists an ideal in $P$ which includes $f$, does not include $e$, and satisfies $B$. This ideal cannot be in $\mathcal{I}(M)$ due to the edge from $e$ to $f$ contradicting our assumption that $\mathcal{I}(P, B) \subseteq \mathcal{I}(M)$.

■

Theorem 10.13 allows us to compute slice for any predicate that can be detected efficiently. For example, relational predicates can be detected efficiently using max-flow techniques ([Gar96] Chapter 6).

When the predicate is known to be linear, then we can use a more efficient and direct algorithm to compute the slice as shown in Figure 10.5. The algorithm determines for every element $e \in P$, the least order ideal of $P$ that includes $e$ and satisfies $B$. Since $B$ is a linear predicates and the condition "includes e" is also linear, it follows that least is well-defined whenever there are satisfying cuts. The correctness of this algorothm is left as an exercise.

```
(1) poset function computeSlice(B:linear_predicate, P: poset)
(2) var
(4) begin
(5)     for every pair of node e in P do
(6)         output the least order ideal of P that satisfies B and includes e;
(7)     endfor
(8) end;
```

Figure 10.5: An efficient algorithm to compute the slice for a linear predicate $B$

## 10.2 Problems

1. Show that whenever $B$ is linear, the order ideal $J_B(e)$ defined as the least order ideal that satisfies $B$ and includes $e$ is a join-irreducible element of the slice of order ideals that satisfy $B$. Conversely, every join-irreducible elements of the slice is of the form $J_B(e)$ for some $e$.

## 10.3 Bibliographic Remarks

The definition of slicing is from [**?**, **?**]. The algorithm in Figure **??** is from [**?**].

# Chapter 11

# Applications to Combinatorics

## 11.1 Introduction

A combinatorial problem usually requires enumerating, counting or ascertaining existence of structures that satisfy a given property $B$ in a set of structures $L$. We now show that slicing can be used for solving such problems mechanically and efficiently. Specifically, we give an efficient (polynomial time) algorithm to enumerate, count or detect structures that satisfy $B$ when the total set of structures is large but the set of structures satisfying $B$ is small. We illustrate our techniques by analyzing problems in integer partitions, set families, and set of permutations.

Consider the following combinatorial problems:

(Q1) Count the number of subsets of the set $[n]$ (the set $\{1 \dots n\}$) which have size $m$ and do not contain any consecutive numbers.

(Q2) Enumerate all integer partitions less than or equal to $(\lambda_1, \lambda_2, \dots, \lambda_n)$ in which $\lambda_1$ equals $\lambda_2$.

(Q3) Give the number of permutations of $[n]$ in which $i$ less than or equal to $j$ implies that the number of inversions of $i$ is less than or equal to the number of inversions of $j$.

Our goal in this paper is to show how such problems can be solved *mechanically* and *efficiently* for any fixed values of the parameters $n$ and $m$.

It is important to note that someone trained in combinatorics may be able to solve all of these problems efficiently (and the reader is encouraged to solve these problems before reading further). Our emphasis is on techniques that can be applied mechanically. On the other hand, note that for the fixed values of $n$ and $m$, all the sets above are finite and therefore all the problems can be solved mechanically. Our emphasis is on *efficiency*. To be more precise, let $L$ be a large set of combinatorial structures (for example, all subsets of $\{1 \dots n\}$ of size $m$, all permutations of $[n]$ etc.) Each combinatorial problem requires enumerating, counting, or searching the subset of structures that satisfy a given property $B$. Call this set $L_B \subseteq L$. For example, in the problem (Q1), $L$ is

the set of all subsets of $[n]$ of size $m$ and $L_B$ is the set of all subsets of $[n]$ of size $m$ that do not contain any consecutive numbers. For any fixed set of parameters $m$ and $n$, the size of $L$ is large but finite, enabling one to enumerate all possible structures and then to check each one of them for the property $B$. This approach results in an algorithm that requires time proportional to the size of the set $L$ which is exponential in $n$ (or $m$). This paper proposes a technique that provides answers to some combinatorial problems in polynomial time and for others, such as those involving enumeration, in time proportional to the size of $L_B$ (and not $L$).

To explain our technique, we use the term *small* to mean polynomial in $n$ and $m$, and *large* to mean exponential in $n$ or $m$. Thus, the set $L$ is large. We first build a *small* structure $P$ such that all elements of $L$ can be generated by $P$. Second, we compute a *slice* of $P$ with respect to $B$, denoted by $P_B$, such that $P_B$ generates all elements of $L_B$. $P_B$ is a small structure and can be efficiently analyzed to answer questions about $L_B$ or enumerate all elements of $L_B$.

### 11.1.1 Counting Ideals

For counting the number of elements in $L$ and its sublattices, we use $N(P)$ to denote the number of ideals of the poset $P$. Since our interest is in efficient calculation of $N(P)$, we will restrict the *dimension* of the partial order generating the lattice. For any poset $(X, P)$, the dimension of $(X, P)$, denoted by $dim(X, P)$, is the least positive integer $t$ for which there exists a family $\{C_1, C_2, \ldots, C_t\}$ of linear extensions of $P$ (total orders compatible with $P$) such that $P$ is equal to the intersection of $C_1, \ldots, C_t$. Determining whether a poset $P$ with $n$ points is 2-dimensional and isomorphism testing for 2-dimensional orders can be done in $O(n^2)$ time [**?**]. All the posets used in this paper are 2-dimensional. The reader is referred to [**?**] for the dimension theory of posets. The following lemma shows that the number of ideals of a poset can be calculated efficiently for series-parallel posets (a special case of 2-dimensional posets) [**?**]. For generalization to counting ideals of two dimensional posets see [**?**].

**Lemma 11.1 (Counting Lemma)**

1. *If $Q$ is an extension of $P$ then $N(Q) \leq N(P)$.*
2. *(Parallel) Let $P + Q$ be the disjoint union (or direct sum) of posets $P$ and $Q$ (see [DP90]). Then, $N(P + Q) = N(P)N(Q)$.*
3. *(Series) Let $P \oplus Q$ be the ordinal sum of posets $P$ and $Q$[DP90]. Then, $N(P \oplus Q) = N(P) + N(Q) - 1$.*
4. *Assume that $P$ can be decomposed into the least number of chains $C_1, C_2, \ldots C_n$. Then*

$$N(P) \leq \prod_{i=1}^{n} (|C_i| + 1).$$

*When each chain is at most $m$ in length, we get that $N(P) \leq (m + 1)^n$.*

For some examples, instead of enumerating all ideals of a poset we may be interested in enumerating or counting ideals in a certain *level set*. To define *level sets*, first define a poset to be *ranked* if for

each element $e \in P$, one can assign a non-negative integer, $rank(e)$, such that if $f$ covers $e$, then $rank(f) = rank(e) + 1$. The set of all elements in $P$ with rank $i$ are called its *level set* with rank $i$. Every distributive lattice is a ranked poset [**?**].

### 11.1.2 Boolean Algebra and Set Families

Let $X$ be a ground set on $n$ elements. Assume that we are interested in the sets of subsets of $X$. By using $\subseteq$ as the order relation, we can view it as a distributive lattice $L$. This lattice has $n+1$ levels and each level set of rank $k$ in the boolean lattice corresponds to $\binom{n}{k}$ sets of size $k$. $L$ is generated by the directed graph in Figure 14.7(a). It is easy to verify that there is a bijection between every nontrivial ideal of the graph and a subset of $X$.

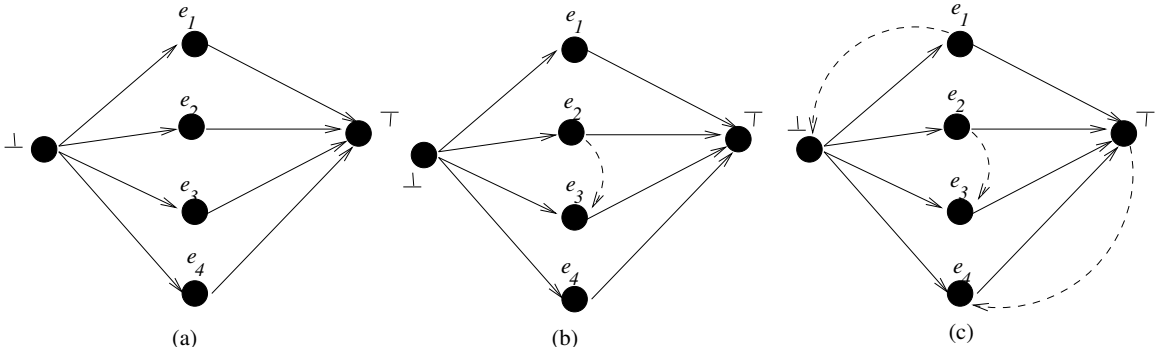



Figure 11.1: Graphs and slices for generating subsets of $X$ when $|X| = 4$

Now consider all subsets of $X$ such that if they include $e_i$ then they also include $e_j$. To obtain the slice with respect to this predicate, we just need to add an edge from $e_j$ to $e_i$. Figure 14.7(b) shows the slice with respect to the predicate "if $e_3$ is included then so is $e_2$." To ensure the condition that $e_i$ is always included, we simply add an edge from $e_i$ to $\perp$ and to ensure that $e_i$ is never included in any subset, we add an edge from $\top$ to $e_i$. Figure 14.7(c) shows the slice which gives all subsets that always contain $e_1$, never contain $e_4$ and contain $e_2$ whenever they contain $e_3$.

As an application, we now solve some combinatorial problems. Let $n$ be even. We are required to calculate the total number of subsets of $[n]$ which satisfy the property that if they contain any odd integer $i$, then they also contain $i+1$ (or equivalently, compute the number of ways to select groups from $n/2$ couples such that a wife is always accompanied by her husband in the group although a husband may not be accompanied by his wife). Although this problem can be solved directly by a combinatorial argument, we will show how our method can be applied. We first construct the poset which generates all the subsets of $[n]$. It is Figure 14.7(a) in the case of 4. We now define the subset of interest by a predicate $B$. For any subset $G$ of $[n]$, we let $B(G)$ to be true if $G$ contains $i+1$ whenever it contains any odd integer $i$. From our discussion of regular predicates, it is clear that $B$ is regular. To compute the slice, it is sufficient to have a predicate detection algorithm. Detecting the least ideal that satisfies $B$ can be done efficiently because it satisfies efficient advancement property. If an ideal does not satisfy $B$, then there is some unaccompanied wife and therefore the element corresponding to the husband is crucial. By applying predicate detection algorithm repeatedly, we can determine all the edges that need to be added to get the slice. In this example, it is sufficient to add an edge from $e_{i+1}$ to $e_i$ for odd $i$. The slice consists of $n/2$ chains each with

exactly 2 elements (ignoring $\perp$ and $\top$). From the counting lemma (Lemma 11.1), it follows that the total number of ideals is $(2+1)^{n/2} = 3^{n/2}$. The reader should note that for any fixed value of $n$, the problem can be solved by a computer automatically and efficiently (because the slice results in a series-parallel poset).

### 11.1.3 Set families of Size $k$

It is important to note that regularity of $B$ is dependent upon the lattice structure of $L$. For example, in many applications of set families, we are interested in sets of a fixed size $k$. The predicate $B$ that the ideal is of size $k$ is not regular. However, by considering alternative posets, this set family can still be analyzed.

Now let us apply our theory to the first combinatorial problem (Q1) mentioned in the introduction. Assume that we are interested in counting all subsets of $n$ of size $k$ which do not have any consecutive numbers. In this example, $G$ satisfies $B$ if whenever $P_i$ has $t$ elements in $G$, $P_{i+1}$ has at least $t+1$ elements in $G$. This condition is linear (in fact regular) and we can use the algorithm in Figure **??** to compute the slice. Figure 14.8 shows the original graph with $k$ chains, each with $n-k$ elements. Let us call this graph $\mathcal{K}\ (k, n-k)$. Figure 11.2 shows the slice which includes precisely those subsets that do not contain consecutive numbers. By collapsing all strongly connected components and by removing the transitively implied edges we get a graph which is isomorphic to $\mathcal{K}\ (k, n-k-(k-1))$ (i.e., the original graph when there are $k$ chains and each chain has only $n-k-(k-1)$ elements). Therefore, the total number of order ideals is $\binom{n-k+1}{k}$. Again one can come up with a combinatorial argument to solve the problem (for example, see Theorem 13.1 and Example 13.1 in [**?**]), but the slicing approach is completely mechanical.

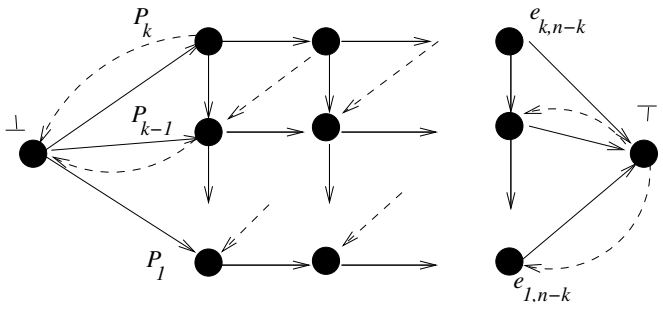



Figure 11.2: Slice for the predicate "does not contain consecutive numbers"

As another example, Catalan predicate is regular and we can compute the slice automatically for all $n$ size subsets of $2n$ that are Catalan. The slice for $n$ equal to 5 after simplification is shown in Figure 11.3.

All the above constructions can be generalized to multidimensional grids to obtain results on multinomials instead of binomials.

## 11.2 Integer Partitions

**Definition 11.2** *$\lambda = (\lambda_1, ..., \lambda_k)$ is an unordered partition of the integer $n$ if*

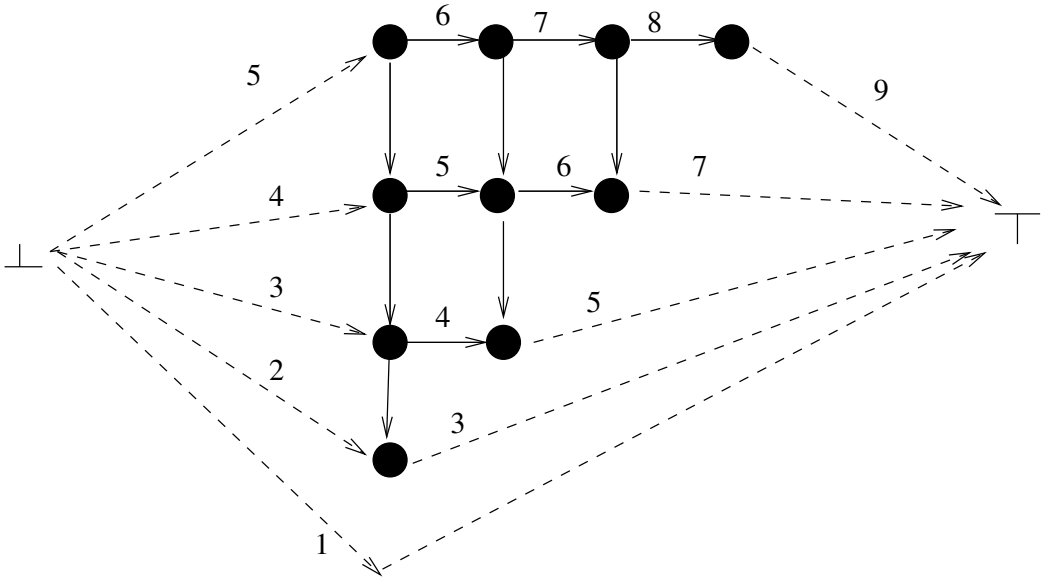
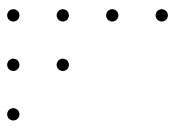



Figure 11.3: Slice for the predicate "Catalan numbers"



$$7 = 4 + 2 + 1$$

Figure 11.4: Ferrer's diagram for the integer partition $(4, 2, 1)$ for 7.

1. $\forall i \in [1, k-1] : \lambda_i \geq \lambda_{i+1}$
2. $\sum_{i=1}^{k} \lambda_i = n$

**Definition 11.3** *$p(n)$ is the number of unordered partitions of the integer $n$.*

**Definition 11.4** *A Ferrer's diagram for an integer partition $\lambda = (\lambda_1, ..., \lambda_k)$ of integer $n$ is a matrix of dots where the $i^{th}$ row contains $\lambda_i$ dots (example Figure 11.4). Thus each row represents the size of a partition and the number of rows represents the number of parts in the partition.*

**Exercise 11.1** *Show that the number of partitions of $n$ with $k$ parts and largest part equal to $m$ is equal to the number of partitions of $n$ with $m$ parts and largest part equal to $k$.*

**Solution.** The two numbers can be shown to be equal by noting that there exists a one to one correspondence between a partition $\lambda_a$ of $n$ with $k$ parts such that the largest part equal to $m$ and

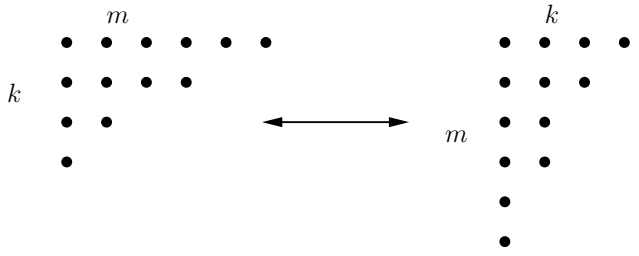



Figure 11.5: Solution for Exercise 11.1.

a partition $\lambda_b$ of $n$ with $m$ parts such that the largest part equal to $k$. This is because $\lambda_a$ can be transformed to a partition of the later type by rotating its Ferrer's diagram by $90^o$ and vice versa (Figure 11.5). □

Note the difference between the problems of integer partition and set partition. The set partition problem says that given a set $S$, find a partition of $S$ into subsets $S_i$ such that $\cup_i S_i = S$ and $\forall i \neq j S_i \cap S_j = \phi$. As opposed to this, the integer partition problem asks to partition a given positive integer into positive integers so that they sum up to the given integer.

For getting the integer partitions, we listed the following methods :

- Ferrer's diagram
- Generating function
- Slicing

## 11.3 Partitions using generating functions

**Definition :** $p(n, k)$ = number of partitions of $n$ with largest part $\leq k$.
We also refer to $p(n, n)$ as $p(n)$
The recurrence for $p(n, k)$ is given by

$$p(n, k) = p(n - k, k) + p(n, k - 1)$$

This recurrence can be justified in the following way. The term $p(n, k - 1)$ gives the number of partitions of $n$ where each part is $\leq k - 1$ and the term $p(n - k, k)$ gives the number of partitions of $n$ where atleast one part is $k$. These two cases together account for all the partitions of $n$. The base case for this recurrence is given by $p(n, 0) = 0$.
$p(n)$ can also be viewed in terms of the number of solutions of the following Diophantine equation

$$1.x_1 + 2.x_2 + 3.x_3 + \ldots + n.x_n = n$$

In one solution of this equation $x_i$ would give the number of $i$ present in the partition. We will try to get solutions to this equation using the generating functions.

### 11.3.1 Generating functions

Generating functions in very broad terms can be defined as the polynomials with "interesting" properties. For example, in the polynomial $(1+z)^n$, coefficient of $x^i$ gives $\binom{n}{i}$. Generating function for $p(n)$ is given by:

$$P(z) = (1 + z + z^2 + \ldots)(1 + z^2 + z^4 + \ldots)(1 + z^3 + z^6 + \ldots)\ldots$$

The first term in this polynomial represents the number of partitions which have value 1. Similarly, the second term in this polynomial represents the number of partitions which have value 2. So in $P(z)$ the coefficient of $z^n$ gives $p(n)$.$P(z)$ can be simplified in the following way:

$$\begin{aligned} P(z) &= (1 + z + z^2 + \ldots)(1 + z^2 + z^4 + \ldots)(1 + z^3 + z^6 + \ldots)\ldots \\ &= (\frac{1}{1-z})(\frac{1}{1-z^2})(\frac{1}{1-z^3})\ldots \\ &= \frac{1}{\prod_{k=1}^{\infty}(1-z^k)} = \sum_{n=0}^{\infty} p(n)z^n \end{aligned}$$

This formula is attributed to Euler.

Now we look at generating functions for partitions with added restrictions.
**Definition :** $q(n)$ = number of partitions of $n$ that doesn't contain 1.
The generating function for $q(n)$ is given by :

$$\begin{aligned} \sum_{n=0}^{\infty} q(n)z^n = Q(z) &= (1 + z^2 + z^4 + \ldots)(1 + z^3 + z^6 + \ldots)\ldots \\ &= \frac{1}{\prod_{k=2}^{\infty}(1-z^k)} \\ &= (1-z)P(z) \\ &= \sum_{n=0}^{\infty} p(n)z^n - \sum_{n=0}^{\infty} p(n)z^(n+1) \\ &= \sum_{n=0}^{\infty} p(n)z^n - \sum_{n=0}^{\infty} p(n-1)z^(n)\{\text{define p(-1) = 0}\} \\ &= \sum_{n=0}^{\infty} (p(n) - p(n-1))z^(n) \end{aligned}$$

Therefore, $q(n) = p(n) - p(n-1)$
**Definition :** $p_d(n)$ = number of partitions of $n$ with distinct partitions.
**Definition :** $p_o(n)$ = number of partitions of $n$ with odd partitions.

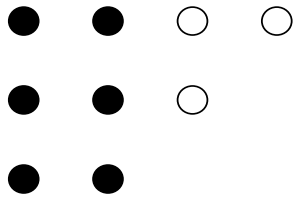


Figure 11.6: Ferrer's diagram for $(4,3,2)$ shown to contain $(2,2,2)$

The generating function for $p_d(n)$ is given by :

$$\begin{aligned}\sum_{n=0}^{\infty} p_d(n)z^n = P_d(z) &= (1+z)(1+z^2)(1+z^3)\dots \\ &= \frac{1-z^2}{1-z}.\frac{1-z^4}{1-z^2}.\frac{1-z^6}{1-z^3}\dots \\ &= \frac{1}{\prod_{k=1,3,5,\cdots}^{\infty}(1-z^k)} \\ &= P_o(z) = \sum_{n=0}^{\infty} p_o(n)z^(n)\end{aligned}$$

Therefore, $p_d(n) = p_o(n)$.

## 11.4 Young's lattice

We define an order on the partitions in the following way: Given two partitions $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_m), \delta = (\delta_1, \delta_2, \dots, \delta_n)$, we say that $\lambda \geq \delta$ iff $m \geq n$ and $\forall 1 \leq i \leq n, \lambda_i \geq \delta_i$. This can also be viewed in terms of containment in the Ferrer's diagram i.e. $\lambda > \delta$ if the Ferrer's diagram for $\delta$ is contained in the Ferrer's diagram of $\lambda$. For example, consider the partitions $(4,3,2)$ and $(2,2,2)$. The Ferrer's diagram for $(2,2,2)$ is contained in the Ferrer's diagram for $(4,3,2)$ as shown in figure 11.6. Hence, $(4,3,2) > (2,2,2)$.

**Definition:** Given a partition $\lambda$, *Young's lattice* $Y_\lambda$ is the lattice of all partitions that are less than or equal to $\lambda$.
The Young's lattice for $(3,3,3)$ is shown in figure 14.9. Note that partitions less than a partition are not necessarily the partitions of the same integer.

We claim that $Y_\lambda$ is the set of ideals of the poset corresponding to the Ferrer's diagram of $\lambda$. This can be easily seen in the light of the fact that for any ideal $\delta$ of the Ferrer's diagram, the components satisfy $\forall i > j, \delta_i > \delta_j$. An example is shown in figure 11.8 As a result of this claim, some corollaries immediately follow.

**Corollary 11.5** *$Y_\lambda$ is distributive*

**Proof:** This immediately follows from the fact the set of ideals of a poset form a distributive lattice

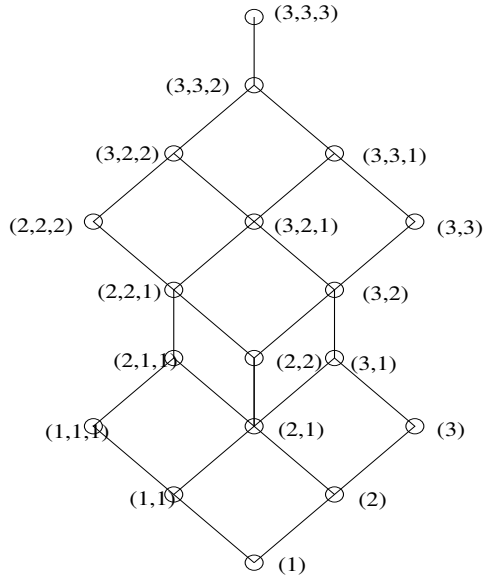



Figure 11.7: Young's lattice for $(3, 3, 3)$

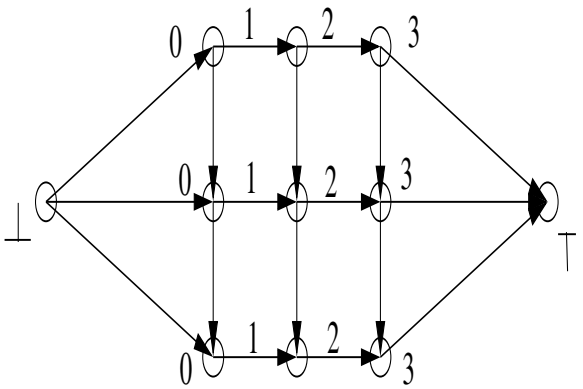



Figure 11.8: The poset corresponding to Ferrer's diagram of $(3, 3, 3)$

■

**Corollary 11.6** *Number of Ferrer's diagram that can fit in a $m \times k$ box is $\binom{m+k}{k}$*

**Proof:** This follows from the result we proved earlier about $L_{m,k}$.

■

**Corollary 11.7** *All partitions of $n$ are at the $n^{th}$ level of $\underbrace{Y_{(n,n,\ldots,n)}}_{n}$.*

**Proof:** This follows from the fact that all partitions of $n$ are present in $Y_(n, n, \ldots, n)$ and are incomparable. Note that all the partitions of numbers less than $n$ are also present. An example of this can be seen in figure 14.9.

■

**Definition:** $q(l, k, n-k)$ = number of partitions of $l$ such that their Ferrer's diagram fits in $k \times n-k$ box.

**Corollary 11.8**

$$\binom{n}{k} = \sum_{l=0}^{n-k} q(l, k, n-k)$$

**Proof:** The result follows from definition of $q$ and corollary 11.6.

■

We now examine the partitions which have some additional conditions imposed on them. These additional conditions can then be viewed as predicates and the slicing method could be used to give us a graph which could generate all such partitions. Some very natural and interesting partitions are :
All partitions that are less than or equal to $\lambda$

- with odd parts
- with distinct parts
- with even parts
- with $(\lambda_1 = \lambda_2)$
- with $(\lambda_k = 1)$

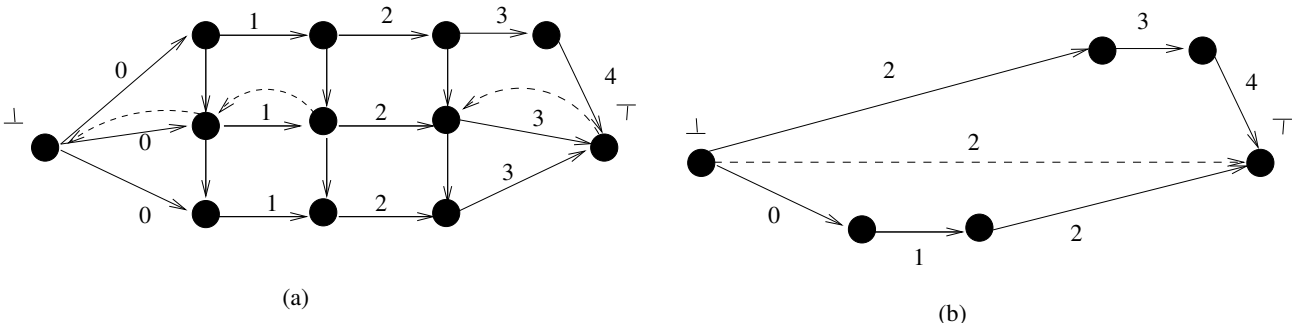



Figure 11.9: Slice for $\delta_2 = 2$

Interestingly, it turns out that all these predicates are regular !! The reader is encouraged to prove that this indeed is true.

Assume that we are interested in all those partitions such that their second component is some fixed value say $b$. It is easy to verify that partitions $\delta \in Y_\lambda$ such that $\delta_2 = b$ form a sublattice and therefore the condition $\delta_2 = b$ is a regular predicate.

Figure 11.9(a) gives the slice of partitions in which $\delta_2 = 2$. Since the second part must be 2, additional edges ensure that the ideal of the graph has exactly two elements from $P_2$. On collapsing the strongly connected components and transitively reducing the graph we get the series-parallel graph Figure 11.9(b). By applying counting lemma, we get that there are $(2+1)(2+1) = 9$ such partitions which can all be enumerated automatically using Figure 11.9(b). They are:

$$\{220, 221, 222, 320, 321, 322, 420, 421, 422\}$$

As another example assume that we are interested in all partitions less than $\lambda$ which have distinct parts. Figure 11.10(a) gives the slice and Figure 11.10(b) gives the graph after simplification. The graph is equivalent to that of subsets of size 3 from [5]. Hence, there are $\binom{5}{3}$ such partitions. These partitions can also be enumerated from the figure. They are:

$$\{210, 310, 410, 320, 420, 430, 321, 421, 431, 432\}.$$

Some other subsets of partitions discussed in the literature are "partitions with odd number of parts," "partitions with distinct odd parts," "partitions with even number of parts" etc. These are also regular predicates. Recently, Ferrari and Rinaldi have given generating functions for the classes of integer partitions whose Ferrers diagram fit into a hook shape[**?**]. They consider integer partitions with restrictions such as $k^{th}$ part is at most $h$. Such restrictions are also regular.

Now the reader may also see the solution for the second problem (Q2) mentioned in the introduction—enumerating all partitions in the Young's lattice $Y_\lambda$ with first part equal to the second part. We simply define the predicate $B$ on a partition $\delta$ to be true when $\delta_1$ equals $\delta_2$. It is clear that the predicate is closed under joins and meets and is therefore a regular predicate. One can draw the slice and conclude that the number of partitions $\delta$ in $Y_\lambda$ satisfying $\delta_1 = \delta_2$ is equal to the number of partitions in $Y_\delta$ where $\delta = (\lambda_2, \lambda_3, \ldots, \lambda_k)$. The slice can also be used to enumerate all required partitions.

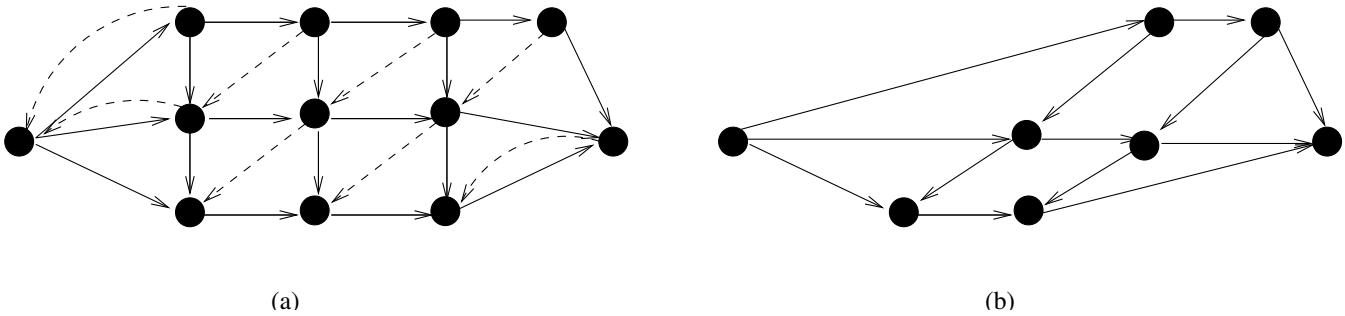



Figure 11.10: Slice for "distinct parts"

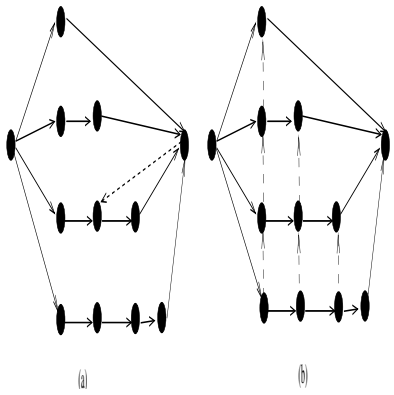



Figure 11.11: Poset for generating all 4! permutations

Note that the level set of rank $N$ of $Y_\lambda$ (where $\lambda = (\lambda_1, \lambda_2 \ldots, \lambda_t)$) corresponds to all partitions of $N$ with at most $t$ parts and the largest part at most $\lambda_1$. It follows that all partitions of $N$ can be enumerated as the elements in level set of rank $N$ of $Y_{(N,N,..N)}$.

## 11.5 Permutations

The set of permutations on $n$ numbers = $n!$. This number is very large as compared to $n$ itself. So here also, we would like to use some representation which could generate all the permutations or permutations which obey certain properties. Suppose the $n$ numbers are labelled $0, 1, \ldots, n-1$. Then consider the poset given in figure 11.13. We claim that the given poset will generate all the 4! permutations. For this we define the concept of number of inversions of $i$.

**Definition:** Number of inversions of $i$ = number of elements smaller than $i$ that appear after $i$.
A consistent cut of the poset would give us an ordered set $(a_1, \ldots, a_(n-1))$. In this set $a_i$ gives us the number of inversions of $i$ in the permutation. Since 0 always has 0 inversions, we don't include it in the set. Each set of inversions gives rise to a permutation and similarly, corresponding to each permutation, there is a unique set of inversions. For example, in figure 11.13 the cut gives the set $(1, 2, 1)$. This maps to the permutation $(2, 1, 3, 0)$. Similarly, the permutation $(0, 1, 3, 2)$ maps to $(0, 0, 1)$ as the set of inversions.

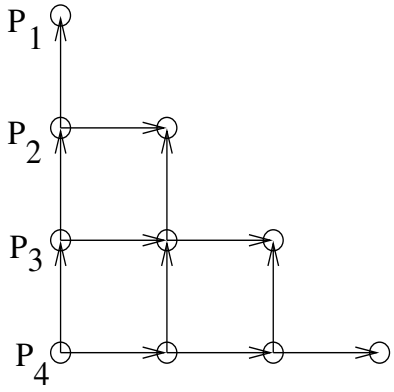



Figure 11.12: Poset for generating poset with additional constraint

Here also we can add additional constraints on the permutations to be generated and model them as predicates. For example, consider the following set of permutations.
All permutations with $i \leq j \Rightarrow i$ has atmost as many inversions as $j$.
This can be modeled as the poset shown in figure 11.12. Since this poset is the same as the one we used for generating the partitions, it follows that the required number of permutations is equal to the number of integer partitions $\leq (n-1, n-2, \ldots, 1)$. Hence this method can offer insight into relationship between two seemingly different problems.

It is easy to show that the following predicates are regular. Further by computing the slice, we can also calculate the number of permutations satisfying $B$.

**Lemma 11.9** *All the following properties of permutations are regular.*

1. *The symbol $m < n$ has* at most *$j$ inversions (for $j < m$). The total number of such permutations is $\frac{n!(j+1)}{m}$.*
2. *$i \leq j$ implies that $i$ has at most as many inversions as $j$. The total number of such permutations is same as the number of integer partitions less than $(n-1, n-2, ..., 1)$ which is equal to $n$th Catalan number.*

**Proof: Lemma 11.9** For the first part, note that it is sufficient to add an edge from $\top$ to element $e_j$ in chain $P_{m+1}$. This ensures that symbol $m$ cannot have more than $j$ inversions. Figure 11.13(a) shows the slice for set of permutations on [5] such that the symbol 4 has at most 1 inversion.

For the second part, we simple add an edge from element $e_j$ on $P_{i+1}$ to $P_i$ for all $j$ and $i$. Thus $P_i$ can execute $j$ elements only if $P_{i+1}$ has executed $j$ or more elements. The claim then follows by comparing the poset with that corresponding to Young's lattice.

■

The level set at rank $k$ of the permutation lattice consists of all permutations with total number of inversions equal to $k$ and therefore such permutations can be efficiently enumerated [**?**, **?**].

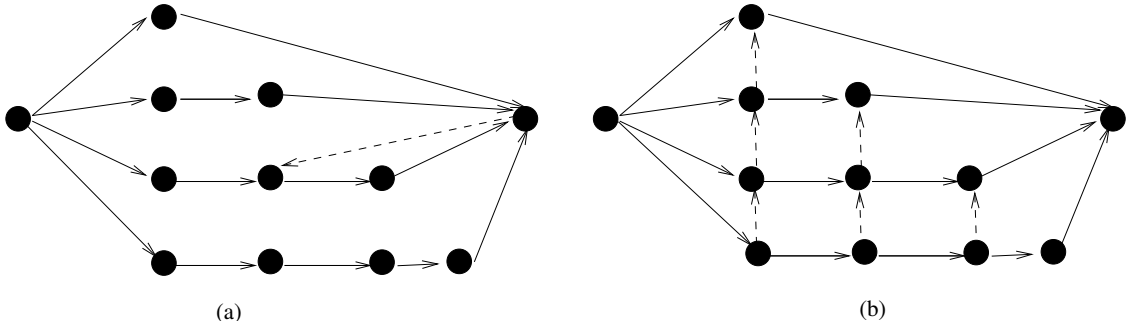



Figure 11.13: Slice for subsets of permutations

As another example, consider the problem of enumerating or counting all permutations such that no symbol has more that $n/2$ inversions and the total number of inversions is less than $c$. The predicate "no symbol has more than $n/2$ inversions" is a regular predicate that can be detected efficiently. The predicate "the total number of inversions is less than $c$," is a relational predicate and can also be detected efficiently. Therefore, by applying results of this paper, we can mechanically construct slices for such permutations.

### 11.5.1 Bipartite Graphs

Consider the set of independent sets in a bipartite graph $(X, Y, E)$. Every independent set $S$ can be written as union of $S_X = S \cap X$ and $S_Y = S \cap Y$. Given two independent sets $S$ and $T$, define

$$S \leq T \equiv S_X \subseteq T_X \wedge S_Y \supseteq T_Y.$$

It is easy to see that the set of all independent sets form a distributive lattice.

Now let us focus on subsets of independent sets. We have the following lemma.

**Lemma 11.10** *The set of independent sets which contain $x$ only if they contain $y$ form a sublattice and the slice can be computed for enumeration.*

11.1. Give an algorithm to enumerate the subsets $\{a_1, \ldots, a_m\}$ of $n$ such that for all $i$ and $j$, $a_i$ does not divide $a_j$.

11.2. Consider the set of all $n$-tuples such that $i^{th}$ coordinate belongs to $[m_i]$. This set has $m_1 m_2 \ldots m_n$ elements and is the ideal lattice of disjoint union of $n$ chains, $C_1, C_2, \ldots, C_n$ such that $|C_i| = m_i - 1$. This set reduces to the boolean lattice when all $m_i$ equal 2 and to the set of all permutations when $m_i = i$. Show that each of the following predicates are regular and therefore the subsets corresponding to them can be enumerated efficiently.

   (a) The set of all tuples $(a_1, a_2, \ldots, a_n)$ such that $\forall i, j : i \leq j \Rightarrow a_i \leq a_j$.

   (b) The set of all tuples $(a_1, a_2, \ldots, a_n)$ such that $a_i \geq c$ (or $a_i = c$, $a_i \neq c$ etc.) for some constant $c$.

   (c) The set of all tuples $(a_1, a_2, \ldots, a_n)$ such that $a_i = a_j$ for fixed $i$ and $j$.

# Chapter 12

# Interval Orders

## 12.1 Introduction

This chapter covers weak orders (ranking), semiorders, and interval orders. All above order are chain-like posets. All are partial orders, with "lots" of order. The relationship between these orders is as follows. The interval order is the most general class. Every weak order is also a semiorder, and every semiorder is also an interval order.

## 12.2 Weak Order

A weak order, also called ranking, is a slight generalization of a total order.

**Definition 12.1** *A poset* $(X, \leq)$ *is a* **weak order** *or a* **ranking** *if there exists a function* $f : X \to \mathbb{N}$ *such that* $\forall x, y \in X : x < y \equiv f(x) < f(y)$

The term "weak order" is somewhat of a misnomer because a weak order has a lot of order. To allow for easy visualization of such orders, we will use the term "ranking" instead of the more accepted term "weak order." The set of elements in a ranking which have the same $f$ value is called a **rank**. For example, a poset $(X, P)$ where $X = \{a, b, c, d\}$ and $P = \{(a, b), (a, c), (a, d), (b, d), (c, d)\}$ is a ranking because we can assign $f(a) = 0, f(b) = f(c) = 1$, and $f(d) = 2$. Here, $b$ and $c$ are in the same rank. The difference between a chain and a ranking is that a chain requires existence of a *one-to-one* mapping such that $x < y$ iff $f(x) < f(y)$. For a ranking, we drop the requirement of the function to be *one-to-one*.

A chain is a ranking in which every rank is of size 1. An antichain is also a ranking with exactly one rank.

Weak orders occur naturally in many applications. Any single scale of measurement provides us a ranking function. For example, in an election if we order all candidates based on the number of

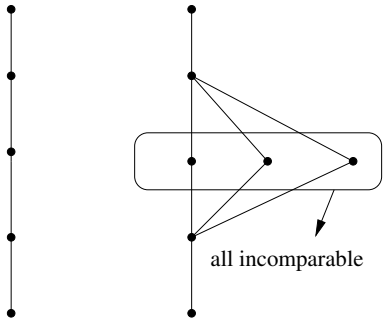



Figure 12.1:

votes they recive, the resulting order is a ranking. In a computer system, jobs are typically assigned a priority level (say a number between 1 and $n$). This assignment imposes a ranking on all jobs.

An alternative characterization of rankings is as follows.

**Theorem 12.2** *$P$ is a ranking iff it does not have $\underline{1} + \underline{2}$ as a sub-poset.*

**Proof:** Left as an exercise.

■

### 12.2.1 Ranking Extension of Partial Orders

Many applications, for example, task scheduling with precedence constraints require that elements in a poset are processed in a order which does not violate precedence constraints. In general, topological sort of a partial order which produces a linear extension of partial order has been useful in such applications. Similar to a linear extension, we can define a ranking extension of a partial order as follows.

**Definition 12.3** *A ranking $s$ is a ranking extension of a partial order $(X, P)$ if $\forall x, y \in X : x <_P y \Rightarrow x <_s y$.*

We call $s$, a *normal* ranking extension of $(X, P)$ if the length of $s$ is equal to the height of $(X, P)$. We have the following result.

**Theorem 12.4** *For every poset $(X, P)$, there exists a normal ranking extension $s$.*

**Proof:** The ranking $s$ can be constructed by the following algorithm (that is implicit in Dilworth's anti-chain covering theorem). Remove all the minimal elements of the partial order and put them in the lowest rank. Get the next set of minimal elements and put them as the next rank. By repeating this procedure till all elements in $(X, P)$ are removed we get the desired ranking. It can be easily verified that the ranking preserve order in $(X, P)$ and has its length equal to the height of the poset.

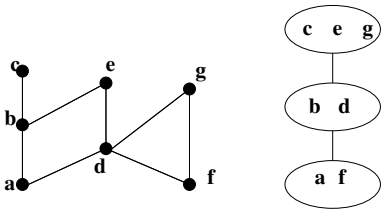



Figure 12.2: A Poset and its Normal Ranking Extension

■

For example, consider the poset in Fig. 12.2. The normal ranking extension produced using the construction in Theorem 12.4 is:

$$\{(a, f), (b, d), (c, e, g)\}$$

It is easily verified that the above ranking preserves the partial order.

If the poset $(X, P)$ denotes tasks, then a normal ranking extension represents a processing schedule (assuming that concurrent tasks can be executed in parallel). The length of the ranking corresponds to a critical path in $(X, P)$.

As another example, consider logical clocks frequently used in distributed systems for timestamping events. A logical clock is a function $C$ from the set of events $E$ to natural numbers such that

$$\forall e, f \in E : e < f \Rightarrow C(e) < C(f)$$

Thus logical clocks are simply ranking extensions of the partial order of events in a distributed system.

## 12.3 Semiorder

A semiorder generalizes a ranking.

**Definition 12.5** $(X, P)$ *is a semiorder if* $\exists f : X \to \mathbb{R}$ *and* $\delta \in \mathbb{R}, \delta \geq 0$ *such that*
$x < y$ *iff* $f(y) - f(x) > \delta$

Note that every ranking is a semiorder, with $\delta = 0$.
Also by scaling, if $\delta \neq 0$ we can always choose $\delta$ to be 1.

**Examples**

1. Load balancing:
   In a distributed system with multiple computers, it is often desirable to move jobs from one computer to the other if one is more heavily loaded than the other. Since moving jobs also incur some cost, it is not desirable to move jobs when the difference in the loads is not much. Therefore, from the perspecive of a load balancing policy, the loads on the computers can be viewed as a semiorder. In this case all processes have a rational number associated with the "load" on each process. If the load numbers fall within a threshold, the processes are considered equally loaded.

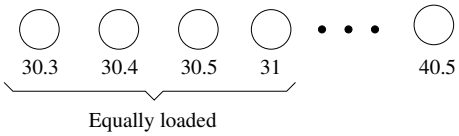



Figure 12.3:

2. Predicate detection (using Physical Time):
   Consider the problem of detecting whether certain "interesting" events can happen simultaneously in a distributed computation Assume that the only information available to us is the sequence of events executed at each process along with the timestamp of the event assigned by the local clock at that process. Since physical clocks can be synchronized only approximately, we may define two events to be concurrent if they are at different processes and their clock values do not differ by some constant $\delta$. If all interesting events within a process are separated by at least $\delta$ amount of time, we can define the order on the set of interesting events as follows:
   $$e < f \equiv T(f) - T(e) > \delta$$
   where $T(e)$, and $T(f)$ are the timestamps of events $e$ and $f$ respectively. It can be verified that the set of interesting events form a semiorder under $<$, and the problem of determining if $k$ interesting events can happen simultaneously is equivalent to determining if there is an antichain of size $k$ in this semiorder.

Semiorder has $\underline{2}+\underline{2}$ and $\underline{1}+\underline{3}$ as forbidden structures. Scott and Suppes' theorem states that if a poset does not have $\underline{2}+\underline{2}$ or $\underline{1}+\underline{3}$ as sub-posets then it is a semiorder. The reader is referred to [Wes04] for a proof of this theorem.

## 12.4 Interval Order

**Definition 12.6** *$(X, <)$ is an interval order if every element $x \in X$ can be assigned a closed interval in real line denoted by $[x.lo, x.hi]$ such that for all $x, y \in X$ $x < y$ iff $x.hi < y.lo$.*

**Lemma 12.7** *Every semiorder is an interval order.*

**Proof:** Consider the mapping: $x \mapsto [f(x) - \delta/2, f(x) + \delta/2]$. Thus every semiorder can be mapped to an interval order.

■

Note that intervals $[0, 1]$ and $[1, 2]$ are concurrent because of the strict $<$ requirement in the definition. Interval orders have been discussed extensively in [**?**].

Interval orders also occur naturally in computer science. In real-time systems, each activity has the beginning and the end time associated with it. Thus, the set of all activities forms an interval order.

We will later see that an interval order can be characterized using the forbidden poset $\underline{2}+\underline{2}$. The following lemma gives a characterization of $\underline{2}+\underline{2}$-free posets in terms of the upper holdings.

**Theorem 12.8** *$\forall x, y \in P : U(x) \subseteq U(y) \vee U(y) \subseteq U(x)$ iff $P$ does not contain $\underline{2}+\underline{2}$.*

**Proof:** LHS
$\equiv$
$\quad \forall x, y \in P : U(x) \subseteq U(y) \vee U(y) \subseteq U(x)$
$\equiv$ { trivially true when $x < y$ or $y < x$ }
$\quad \forall x, y \in P : x \parallel y : U(x) \subseteq U(y) \vee U(y) \subseteq U(x)$

Since the RHS does not contain $U(x)$ or $U(y)$, we eliminate them.
$\forall x, y \in P : x \parallel y : (\forall x' :: x < x' \Rightarrow y < x') \vee (\forall y' :: y < y' \Rightarrow x < y')$.

To prove the equivalence of the above predicate and RHS, we show the inverse.
$\quad \neg[\forall x, y \in P : x \parallel y : (\forall x' :: x < x' \Rightarrow y < x') \vee (\forall y' :: y < y' \Rightarrow x < y')]$.
$\equiv$ { De Morgan's}
$\quad \exists x, y \in P : x \parallel y : (\exists x' :: x < x' \wedge y \not< x') \wedge (\exists y' :: y < y' \wedge x \not< y')$.
$\equiv$ { predicate calculus }
$\quad \exists x, y \in P : x \parallel y : (\exists x', y' :: x < x' \wedge y \not< x' \wedge y < y' \wedge x \not< y')$.
$\equiv$ { $y \not< x'$ replaced by $y \parallel x'$ because of $x' < y$ implies $x < y$ }
$\quad \exists x, y, x', y' \in P : x \parallel y \wedge x < x' \wedge y \parallel x' \wedge y < y' \wedge x \parallel y')$.

The above predicate is equivalent to $P$ does not contain $\underline{2}+\underline{2}$because it is easy to verify that $x, y, x'$ and $y'$ are all distinct in the above predicate (for example, $x \parallel y'$ and $y < y'$ implies that $x \neq y$).

■

From symmetry, we also get that $P$ is $\underline{2}+\underline{2}$-free iff $\{D(x) | x \in P\}$ is a chain.

We now have a characterization theorem for interval posets.

**Theorem 12.9** *[?] $P$ is an interval order iff it does not contain $\underline{2}+\underline{2}$ as a sub-poset.*

**Proof:** [?] We first show that if $P$ contains $\underline{2}+\underline{2}$ then it is not an Interval order. Let $a < b$ and $c < d$ be a subposet that is $\underline{2}+\underline{2}$. Assume if possible, $P$ is an interval poset. By definition of interval order, we get that $a.hi < b.lo$. From $b \not> c$, we get that $b.lo \leq c.hi$. From the last two inequalities, we get that $a.hi < c.hi$. From symmetry of $a$ and $b$ with $c$ and $d$, we also get that $c.hi < a.hi$ which is a contradiction. Thus, if a poset contains $\underline{2}+\underline{2}$ it cannot be an interval order.

Now we show that if $P$ does not contain $\underline{2}+\underline{2}$ then it is an interval order. From Theorem 12.8, upper holdings and lower holdings form a chain. Problem 12.3 asks you to show that the total number of distinct nonempty upper holdings equals the total number of nonempty distinct lower holdings. Let this number be $h$. Now, we provide a function to map an element $x$ of an interval order to a closed interval as:

$$f(x) = [d(x), h - u(x)]$$

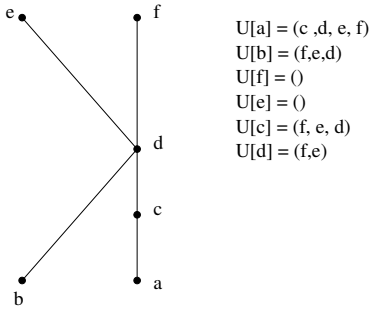
where $d(x)$ is the number of distinct sets of the form $D(y)$ properly contained in $D(x)$, and $u(x)$ is the number of distinct sets of the form $U(y)$ properly contained in $U(x)$. Problem 12.4 requires you to verify that $x < y$ iff $x.hi < y.lo$.

Table 12.1: Forbidden structures for Various Orders

| Order | Forbidden structure |
|---|---|
| Chain | $\underline{1}+\underline{1}$ |
| Weak Order | $\underline{1}+\underline{2}$ |
| Semi Order | $\underline{2}+\underline{2}$ or $\underline{1}+\underline{3}$ |
| Interval Order | $\underline{2}+\underline{2}$ |

■

Table **??** shows the upper and lower holdings for all elements in Figure 12.4. In this example there are three distinct nonempty upper (and lower) holdings.




| | | |
|---|---|---|
| $D(a) =$ | $U(a) = c, d, e, f$ | $a = [0, 0]$ |
| $D(b) =$ | $U(b) = d, e, f$ | $b = [0, 1]$ |
| $D(c) = a$ | $U(c) = d, e, f$ | $c = [1, 1]$ |
| $D(d) = b, c, d$ | $U(d) = f, e$ | $d = [2, 2]$ |
| $D(f) = d, b, c, a$ | $U(f) =$ | $f = [3, 3]$ |
| $D(e) = d, b, c, a$ | $u(e) =$ | $e = [3, 3]$ |

Figure 12.4: Mapping elements of poset to interval order

For example see Table **??** for the example from Figure [12.4].

The proof of Theorem (**??** can readily be converted to an algorithm. Give a poset in adjacency list representation, one can topologically sort it in $O(n + e)$. This allows calculation of $d(x)$ and $u(x)$ for all elements in the poset in $O(n + e)$ time giving us $O(n + e)$ algorithm to obtain interval representation.
Table 12.1 gives the various orders and the corresponding forbidden structures for each order.

12.1. How many nonisomorphic rankings are possible on a set of size $n$? (Hint: Compute the number of rankings of size $k$)

12.2. Give a formula to compute the number of ideals and the number of linear extensions of a ranking.

12.3. Show that the number of distinct nonempty upper holdings is equal to the total number of nonempty distinct lower holdings for any $\underline{2} + \underline{2}$free order.

12.4. Show that the interval assignment in the proof of Theorem **??** satisfies $x < y$ iff $x.hi < y.lo$.

# Chapter 13

# Tractable Posets

## 13.1 Introduction

So far we have seen some efficient algorithms for determining characteristics of posets (e.g., width). However, there are no (known) efficient algorithms for some important properties, such as dimension, number of ideals, number of total orders (linear extensions), and the isomorphism problem. In these cases, we can study special cases and see if there are certain types of posets for which there exist efficient algorithms for some of the difficult problems. In this chapter we cover an important class of posets — 2-dimensional posets and its special cases that include:

1. series parallel posets
2. quasi–series parallel posets
3. decomposable posets

## 13.2 Series Parallel Posets

Consider the following pseudocode, where ';' is the *sequence* (or *series*) operator, and '||' is the *parallel* operator:

```
cobegin
     (s1 ; s2)
  ||
     (s3 ; s4)
coend
```

In this program, $s1$ is always executed before $s2$. However, the two branches formed by the parallel operator are executed independently of one another. The order of execution of statements in this program is a partial order of a special kind called series parallel order.

We now give the definition of a *series parallel poset* (henceforth called *SP poset* or *SP order*):

**Definition 13.1 (Series Parallel Poset)** *1. Any element by itself is an SP poset.*

*2. Given two SP posets $P_1$ and $P_2$,*

*(a) $P_1 * P_2$ is an SP order. The symbol $*$ is the sequencing operator. To form the new poset, take all maximal elements of $P_1$, and make each one less than every minimal element of $P_2$.*

*(b) $P_1 + P_2$ is an SP order. The symbol $+$ is the parallel operator. To form the new poset, take the disjoint union of $P_1$ and $P_2$.*

Figure 13.1 shows example series and parallel compositions for the SP posets $P_1, P_2$, and $P_3$ shown in (a-c).

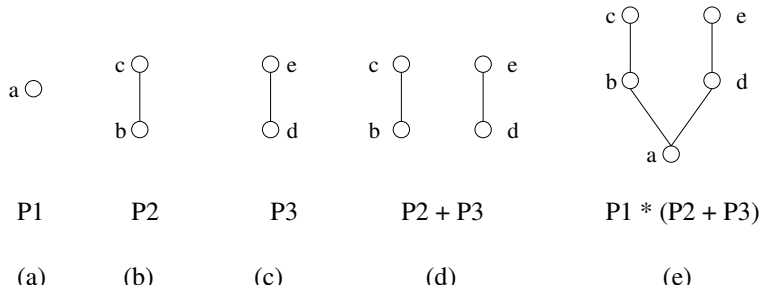



Figure 13.1: Example series and parallel compositions. (a-c) Posets. (d) Result of $P_2 + P_3$. (e) Result of $P_1 * (P_2 + P_3)$.

SP orders can be represented by an *SP tree*. In an SP tree, the leaves are the elements of the poset, and internal nodes are the series and parallel operators. Figure 13.2 (a) shows an SP tree for the poset in Figure 13.1(e). A poset does not necessarily have a unique SP tree representation, since the parallel operator is symmetric.

We can form the *conjugate* of an SP tree by replacing all series operators with parallel, and parallel with series. Note that conjugation is a self-inverse. Figure 13.2(b) shows the conjugate of the SP tree in Figure 13.2(a). Figure 13.2(c) shows the poset for the conjugate.

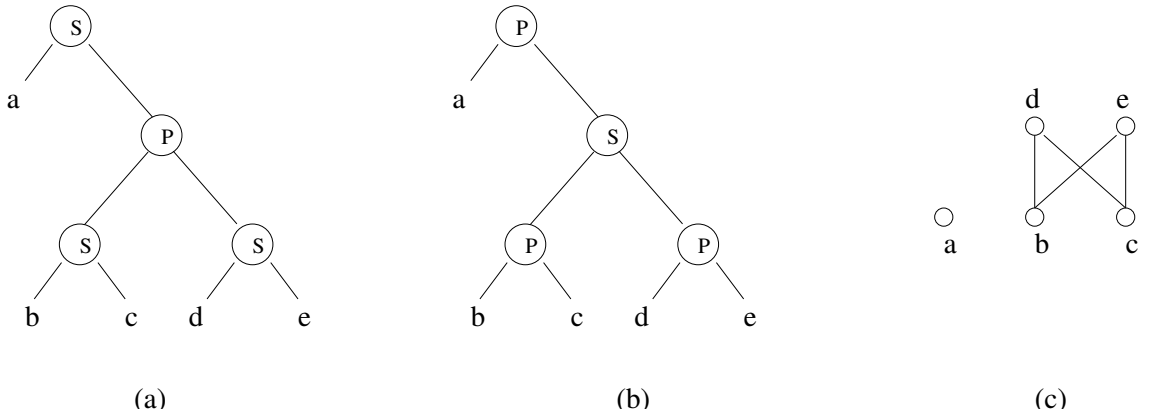



Figure 13.2: (a) An SP tree for the poset in Figure 13.1(e). (b) The conjugate SP tree. (c) The conjugate poset.

We now give a theorem on the dimension of SP orders.

**Theorem 13.2** *dim(SP order)* $\leq 2$.

**Proof:** Recall that a poset is 2-dimensional if there exists a 2-dimensional vector clock that captures the partial ordering. For any SP order, we can perform a depth–first search (DFS) on the poset's

SP tree, labelling the nodes from 1 up in increasing order as we visit them. We then perform a second DFS on the tree, this time visiting the children of P nodes in reverse order. The two sequences give us the two dimensions of a vector clock for each element of the poset.

■

As an example, consider the SP tree from Figure 13.2(a). The first DFS yields the sequence (12345), while the second gives (14523). Thus the time stamp for $b$ = (2,4), $c$ = (3,5), and $d$ = (4,2). The vector clock for $b$ is less than $c$, indicating $b < c$, as indeed is true in the poset. The vector clocks for $b$ and $d$ are incomparable, and we see that in the poset $b \| d$ is also true.

Next we introduce the *comparability graph*:

**Definition 13.3 (Comparability Graph)** *A comparability graph is an undirected graph such that*

$$(u, v) \in E \text{ iff } u \text{ is comparable to } v.$$

The comparability graph can be obtained from either the SP poset or its SP tree:

- $G(P_1 * P_2)$ = add edges from all nodes in $G_1$ to all nodes in $G_2$.
- $G(P_1 + P_2)$ = do nothing.

We will return to comparability graphs later.

Now let us turn to the question of the number of ideals. Let $i(P)$ denote the number of ideals of $P$. For SP orders, the number of ideals can be computed as follows:

- $i(atom) = 2$
- $i(P_1 + P_2) = i(P_1)i(P_2)$
- $i(P_1 * P_2) = i(P_1) + i(P_2) - 1$

## 13.3 Decomposable Partial Orders

We start with a poset $Q$ (Figure 13.3 (a)). Let $a_1, a_2, ..., a_n$ be distinct elements of $Q$. Let $P_1, P_2, ..., P_n$ be disjoint partial orders (see Figure 13.3 (b)). Let the new poset $P = Q(\begin{smallmatrix} P_1 P_2 \ldots P_n \\ a_1 a_2 \ldots a_n \end{smallmatrix})$; that is, replace each element $a_i$ in $Q$ with the corresponding poset $P_i$.

Hence $a < b$ in $P$ iff $a \in V_i$ (the vertex set for $P_i$), $b \in V_j$, and $a_i < a_j$ in $Q$.

The posets $P_i$ thus act like equivalence classes. Members of a set $P_i$ are equivalent to each other with respect to comparisons outside of $P_i$. This may allow us to work with $Q$ instead of $P$, and thus work with a reduced poset.

Consider the full graph of a poset (i.e., $P$ from our example, not the reduced $Q$.). We call the set of vertices of the full graph the *ground set*. We now make the following definition:

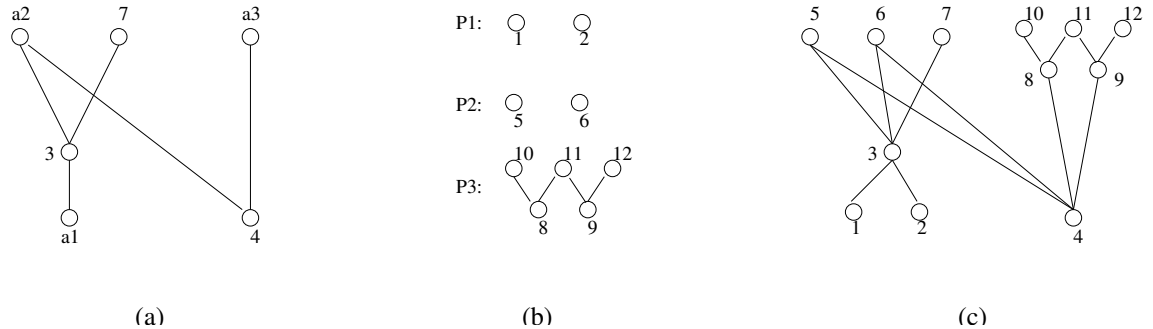



Figure 13.3: (a) A poset $Q$. (b) Three disjoint posets. (c) The new poset $P = Q(\binom{P_1 P_2 P_3}{a_1 a_2 a_3})$.

**Definition 13.4 (Autonomous Set)** *Let $V$ be the ground set of vertices, and let $B \subseteq V$. We say $B$ is autonomous if $\forall a \in V - B, b, c \in B$:*

$$a < b \quad \equiv \quad a < c \qquad \wedge \qquad a > b \quad \equiv \quad a > c$$

We then give the following theorems, without proof, for decomposable posets:

**Theorem 13.5** *A poset is decomposable (also called substitution decomposable) iff it has a non-trivial autonomous set.*

**Theorem 13.6** *Let $l(P)$ be the number of linear extensions of a poset, and $dim(P)$ be the dimension of a poset. Finally, let $P = Q(\binom{P_1 P_2 \ldots P_n}{a_1 a_2 \ldots a_n})$ be a decomposable poset. Then:*

- $l(P) = l(Q) \prod_{i=1}^{n} l(P_i)$
- $dim(P) = max\{dim(Q), dim(P_1), ..., dim(P_n)\}$

These allow us to work with smaller sets and still gain facts about $P$.

## 13.4 2–dimensional Posets

We now define a 2–dimensional poset:

**Definition 13.7** *$dim(P) = 2$ iff there exists a 2-dimensional vector clock that captures the poset relation.*

As an example, consider the poset in Figure 13.4.

Notice that the 2–dimensional vector clock, if it exists, is not necessarily unique. Our example above has many valid vector clocks.

We now give another condition for 2–dim posets.

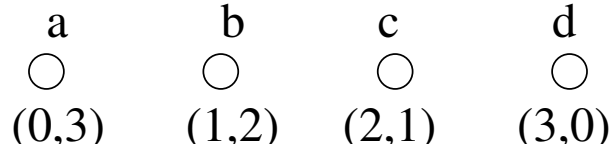



Figure 13.4: A poset and the vector clock values for each element.

**Theorem 13.8** *P is 2–dim iff P has a conjugate poset Q.*

**Definition 13.9 (Conjugate Poset)** *A poset Q is a conjugate poset for P if*

$$a \sim b \text{ in } P \quad \textit{iff} \quad a \parallel b \text{ in } Q$$

Let $G$ be the comaparability graph, $G^C$ its complement. The above states that for 2–dim posets, $G^C$ is guaranteed to also be a comparability graph.

As an example, Figure 13.5(a) shows a poset that is indeed 2–dmiensional. Figure 13.5(b) is the comparability graph $G$, and (c) shows $G^C$. Note that $G^C$ is also a comparability graph. Figure 13.5(d) shows one valid conjugate for the original poset; it is not the only possible one. See the last paragraph below for an example of a poset with no conjugate.

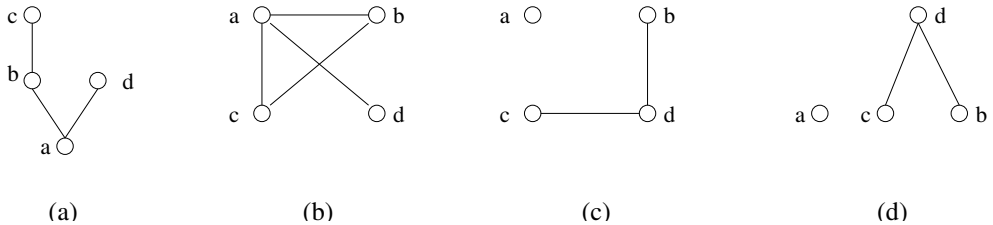



Figure 13.5: (a) A 2–dim poset. (b) Its comparability graph $G$. (c) $G^C$. (d) A valid conjugate for the original poset.

We begin with some definitions of terms to be used further on in our discussion.

**Definition 13.10** *Given a partial order $P = (X, \leq_P)$ , we say $a \sim_P b$ iff*
$(a \leq_P b) \vee (b \leq_P a)$.

**Definition 13.11** *A conjugate of partial order $P = (X, \leq_P)$ is a partial order $Q = (X, \leq_Q)$ such that $(a \sim_P b)$ iff $a \,||_Q\, b$.*

**Definition 13.12** *A linear extension $L = (X, \leq_L)$ of a partial order $P = (X, \leq_P)$ is called a non-separating linear extension (NSLE) if*

$$(u \leq_L w \leq_L v) \wedge (u \leq_P v) \Rightarrow (u \leq_P w) \vee (w \leq_P v)$$

Figure 13.6: (a) is a non-separating linear extension of the partial order (b), when at least one of the dotted edges holds.

Figure 13.6(a) shows a NSLE, and a poset (b) corresponding to it. The dotted lines show the possible relationships between $u$, $v$ and $w$. At least one of the dotted lines must represent a true edge in order for (a) to be a NSLE of (b).

**Theorem 13.13** *The following statements are equivalent:*

1. *P is 2-dimensional.*
2. *The incomparability graph of P is a valid comparability graph of some poset.*
3. *P has a conjugate Q*
4. *P has a non-separating linear extension (NSLE).*
5. *P can be identified with a permutation.*

Figure 13.7: (Poset $P = (X, \leq_P)$.

Figure 13.8: (Linear extensions $L1$ and $L2$ of $P = (X, \leq_P)$. $L1 \cap L2 = P$.

We illustrate Theorem 1 with an example, and use this example to sketch the proof of the theorem. Consider the poset given in Figure 13.7. Figure 13.8 shows two valid linear extensions for this poset, such that their intersection gives back the poset $P$.

$L1$ and $L2$ are called *realizers* of $P$. Note that for every incomparable pair $(a, b)$ in $P$, if $a < b$ in $L1$, then $b < a$ in $L2$. One can "timestamp" each state in $P$ by listing the position of the states in $L1$ and $L2$. Thus, $V1$ has timestamp $(1, 4)$, $V2$ has timestamp $(3, 2)$, and so on.

Figure 13.9: (a) $L2^{-1}$. (b) Conjugate $Q = L1 \cap L2^{-1}$ of the poset $P$.

We can derive the conjugate $Q$ of $P$ from $L1 \cap L2^{-1}$, where $L2^{-1}$ is the total order derived by reversing the order of the elements in $L2$. Figure 13.9(b) shows the poset $Q$ that is the conjugate of $P$ obtained from $L1 \cap L2^{-1}$. We now show that:

$$(x \sim_P y) \Leftrightarrow (x \,||_Q\, y)$$

Since $L1$ and $L2$ are linear extensions of $P$, we have:

$$(x \leq_P y) \Rightarrow (x \leq_{L2} y) \wedge (x \leq_{L1} y)$$

Therefore,

$$y \leq_{L2^{-1}} x$$

Since $Q = L1 \cap L2^{-1}$, we get $x \,||_Q\, y$. Similarly, if $y \leq_P x$, then $y \,||_Q\, x$.

Now, let $P = (X, \leq_P)$ be a 2-dimensional poset with conjugate $Q = (X, \leq_Q)$. Then, $\leq_P \cup \leq_Q$ defines a total order on the elements of $X$.

**Lemma 13.14** $(X, \leq_P \cup \leq_Q)$ *is a non-separating linear extension.*

Figure 13.10: The order imposed by $\leq_P \cup \leq_Q$.

Continuing with our example, Figure 13.10 shows the order imposed by $(X, \leq_P \cup \leq_Q)$. Figure 13.10 uses directed edges to indicate the $\leq$ relation. The non-separating linear extension consistent with the figure is:

$$[V1, V4, V2, V5, V6, V3, V7]$$

Note that every linear extension of $P$ is not an NSLE. For example, the following is a valid linear extension of $P$, but is not an NSLE:

$$[V1, V3, V4, V2, V5, V6, V7]$$

Finally, we show that every 2-dimensional poset $P$ can be identified with a permutation. Let $L1$ and $L2$ be the two linear extensions in a minimal realizer of $P$, where $|P| = n$. We can number the elements of $P$ in such a way that $L1$ is just $[V1, V2, V3, ...., Vn]$. The position of these elements in the other linear extension, $L2$, then defines a permutation on $[n]$.

### 13.4.1 Counting ideals of 2-dimensional poset

Given:

- A 2-dimensional poset $P = (V, \leq_P)$
- $L$: a non-separating linear extension of $P$

We now present a mechanism for computing the number of ideals in the 2-dimensional poset $P$.

Assume, WLOG, that $L = [v_1, v_2, v_3, ..., v_n]$. We assign a label, $L(v_i)$, to each element $v_i$, as follows:

$$L(v_1) = 1$$
$$L(v_i) = 1 + \sum_{j<i,\ v_i||v_j} L(v_j)$$
$$N(P) = \sum_{i=1}^{n} L(v_i)$$

Figure 13.11: Partial order $P$, having a non-separating linear extension $L = [V1, V2, V3, V4, V5, V6, V7]$.

Again, we illustrate the labelling scheme by means of an example. Consider the poset in Figure 13.11. Note that this poset is isomorphic to the one presented in Figure 13.7, but with a different numbering for the nodes. As noted in the figure, the following is a non-separating linear extension for the given poset:

$$[V1, V2, V3, V4, V5, V6, V7]$$

By our labelling scheme, we assign the following labels to the nodes in $P$.

$L(V1) = 1$

$L(V2) = 1 + 0 = 1$ {since $V1 \not\parallel V2$ }

$L(V3) = 1 + L(V1) + L(V2) = 3$ {since $V1 \parallel V3, V2 \parallel V3$}

$L(V4) = 1 + 0 = 1$ {since $V4 \not\parallel \{V1, V2, V3\}$}

$L(V5) = 1 + L(V4) + L(V2) = 3$ {since $V4 \parallel V5, V2 \parallel V5$}

$L(V6) = 1 + L(V1) + L(V2) + L(V3) + L(V4) + L(V5) = 10$ {since $\{V1, V2, V3, V4, V5\} \parallel V6$}

$L(V7) = 1 + L(V1) + L(V2) + L(V4) + L(V5) = 7$ {since $\{V1, V2, V4, V5\} \parallel V7$}

**Lemma 13.15** *$L(v_i)$ is equal to the number of global states containing $v_i$ and all elements $v_j$ with $j < i$.*

13.1. Show that the problem of recognizing any special class of posets that be characterized by a constant number of forbidden posets is in class **NC**. In otherwords, show that there exists an $O(1)$ parallel algorithm to recognize posets in any such class.

13.2. Give a (recursive) formula for $l(P)$, the number of linear extensions of an SP order $P$.

# Chapter 14

# Enumeration Algorithms

## 14.1 Enumeration of ideals for a general poset

The number of ideals of a poset may be exponential in the size of the poset. We have already seen that the ideals of a poset form a distributive lattice under the $\subseteq$ relation. In this section, we explore different ways in which the lattice of ideals may be traversed, in order to enumerate all the ideals of a poset.

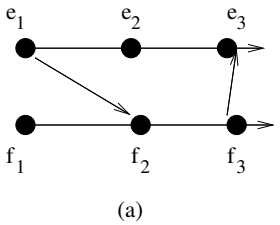
In particular, we explore the following three orders of enumeration:

- Breadth-first search (BFS): This corresponds to the BFS traversal of the lattice of ideals. The algorithm discussed here is from [**?**].
- Depth-first search (DFS): This corresponds to the DFS traversal of the lattice of ideals. Algorithm presented here is due to [**?**].
- Lexicographic order: This corresponds to the "dictionary" order. Algorithm from [**?**].

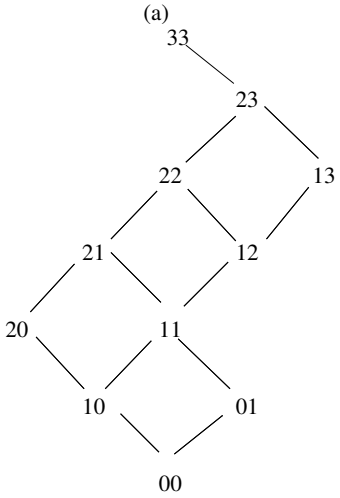
In Chapter 11 we have shown that many families of combinatorial objects can be mapped to the lattice of order ideals of appropriate posets. Thus, algorithms for lex and BFS traversal discussed in the paper can also be used to efficiently enumerate all subsets of $[n]$, all subsets of $[n]$ of size $k$, all permutations, all permutations with a given inversion number, all integer partitions less than a given partition, all integer partitions of a given number, and all $n$-tuples of a product space. Note that [**?**] gives different algorithms for these enumerations. Our algorithms are generic and by instantiating it with different posets all the above combinatorial lex enumeration can be achieved.

### 14.1.1 BFS traversal

We first present a naive algorithm for BFS traversal of the lattice of ideals. This algorithm uses the context of a distributed computation, i.e., the poset is given in chain-decomposed form, and each




(a)




(b)

BFS: 00, 01, 10, 11, 20, 12, 21, 13, 22, 23, 33

DFS: 00, 10, 20, 21, 22, 23, 33, 11, 12, 13, 01

Lexical: 00, 01, 10, 11, 12, 13, 20, 21, 22, 23, 33

(c)

Figure 14.1: (a) A computation (b) Its lattice of consistent global states (c) Traversals of the lattice.

chain corresponds to a process in the distributed computation. The ideals of the poset correspond to consistent global states in the distributed computation.

BFS algorithm 1:

1. $L$: list of the global states at the current level. Initially, $L = \{0, 0, 0, ...., 0\}$.
2. Repeatedly compute $L'$ = list of global states at the next level

The algorithm above has a space complexity that could be exponential in the size of the distributed computation. The algorithm needs to store all the elements at a given level. Each level could have as many elements as the width of the lattice of ideals, which is exponential in the number of processes in the distributed computation. In general, the average number of elements at a certain level in a distributed computation consisting of n processes, each having m events, is given by:

$$\text{Avg number of elements at a level} = O(\frac{m^n}{mn})$$

The other drawback of the above algorithm is that some global states may be visited twice, from different states at the previous level in the lattice. However, this inefficiency is easily corrected by imposing a total order on the set of all events in the computation (*i.e.*, any linear extension of the poset corresponding to the computation). We then traverse the lattice by respecting the order imposed on the events. Thus, during our traversal, an event is executed only if all other eligible events preceding it in the total order have already been executed. This algorithm is presented below.

BFS Algorithm 2:

- $P = (E, \leq)$: partial order.
- $L$: list of global states (ideals). Initially, $L = \{0, 0, 0, ...., 0\}$.
- $\sigma$: total order on the events in $E$ (any linear extension of $P$).
- Repeatedly compute $L'$:

$\forall G \in L$ do
  $\forall e$ enabled in $G$ do
  if $(\forall f \in maximal(G) : f \parallel e \Rightarrow \sigma(e) < \sigma(f))$
    $H := G \cup \{e\}$
  append $H$ to $L'$

## 14.2 Enumeration of ideals for a general poset

### 14.2.1 DFS traversal

Algorithm DFS is a recursive algorithm and is invoked with a global state $S$. Invoking the algorithm with a global state $S$ means that global state $S$ is visited. For each successor of $S'$ of $S$, the

predecessor of $S'$ with the maximum value is computed. The $value(S)$ is the ordered $n$-tuple $(k_1, \ldots, k_n)$, and comparison is done lexicographically. To find the predecessor of $S'$ with maximum value, all the predecessors of $S'$ are computed by decrementing each component of $S'$ and checking whether the resulting state is a global state. Then, if $S$ is the processor of $S'$ with the maximum value, algorithm DFS($S'$) is recursively invoked. The algorithm DFS cannot be used to traverse the lattice of a nonterminating computation.

```
Algorithm DFS(S):
begin
  i = 1;
  Let < k_1, ..., k_n > = S
  while (i ≤ n) do
  begin
    S' = < k_1, ..., k_i + 1, ..., k_n >; //S', if consistent, is a successor of S
    if S' is a global state then
      if value(S) = max(value(pred(S'))) then
        DFS(S');
    i = i + 1;
  end;
end.
```

The above lattice traversal algorithm (algorithm DFS) requires $O(nM)$ time and $O(nE)$ space complexity where $n$ is the number of processes, $M$ is the number of consistent global states and $E$ is the number of events in the computation. The main disadvantage of this algorithm is that it requires recursive calls of depth $O(E)$ with each call requiring $O(n)$ space resulting in $O(nE)$ space requirements besides storing the computation itself.

### 14.2.2 Lexicographic (lex) traversal

Lex traversal is the natural dictionary order used in many applications. It's especially important in distributed computing applications because the user is generally interested in the consistent global state (CGS) that satisfies the predicate to be minimal with respect to some order. The lex order gives a total order on the set of CGS based on priorities given to processes and the CGS detected by this traversal gives the lexicographically smallest CGS.

It is useful to impose on the set of global states the *lex* or the dictionary order. We define the lex order ($<_l$) as follows. $G <_l H$ iff

$$\exists k : (\forall i : 1 \leq i \leq k-1 : G[i] = H[i]) \wedge (G[k] < H[k]).$$

This imposes a total order on all global states by assigning higher priority to small numbered processes. For example, in Figure 14.1, global state (01) $<_l$ (10) because $P_1$ has executed more events in the global state (10) than in (01).

We use $\leq_l$ for the reflexive closure of the $<_l$ relation. Recall that we have earlier used the order $\subseteq$ on the set of global states which is a partial order. The $\subseteq$ order shown in Figure 14.1(b) is equivalent to

$$G \subseteq H \equiv \forall i : G[i] \leq H[i]$$

Note that $01 \not\subseteq 10$ although $01 \leq_l 10$.

Note that we have two orders on the set of global states—the partial order based on containment ($\subseteq$), and the total order based on lex ordering ($\leq_l$). The relationship between the two orders defined is given by the following lemma.

**Lemma 14.1** $\forall G, H : G \subseteq H \Rightarrow G \leq_l H$.

**Proof:** $G \subseteq H$ implies that $\forall i : G[i] \leq H[i]$. The lemma follows from the definition of the lex order.

■

Since there are two orders defined on the set of global states, to avoid confusion we use the term *least* for infimum over $\subseteq$ order, and the term *lexicographically minimum* for infimum over the $\leq_l$ order.

For ease in understanding our approach, we describe two algorithms. The first algorithm has the same space complexity but higher time complexity than the second algorithm. It is based on first enumerating all possible global states in the lex order and then checking for each one of them whether it is consistent. The second algorithm shows how to avoid generating inconsistent global states.

The first algorithm is a generalization of the "add one" algorithm to generate all binary strings of size $n$. To generate all binary strings of size $n$, we start with the number 0 (the string will all 0's) and keep adding 1 till we reach the string with all 1's. Generating all binary strings of size $n$ can be viewed as generating ideals of the poset in which each process has exactly one event. We generalize "add one" algorithm in two ways.

1. Each process $P_i$ has $m_i$ events where $m_i$ is greater than or equal to 1. Therefore, we view the ideal as a string in the mixed-radix notation rather than binary string. For example, in a system with three processes such that $P_1$ has 2 events, $P_2$ has 5 events, and $P_3$ has 4 events, the tuple $(1, 3, 4)$ represents the ideal in which $P_1$ has executed 1 event, $P_2$ has executed 3 events and $P_3$ has executed 4 events. The next ideal in the lex order would be $(1, 4, 0)$.

2. In case of binary strings, we simply added 1 to obtain the next binary string. This is because all binary strings were consistent. In general, a simple addition of 1 may not result in a CGS. For example, in Figure 14.1, we get $(0, 2)$ on adding one to $(0, 1)$ but $(0, 2)$ is not a CGS. Therefore, instead of adding one, we need to add the least number that takes us to the next CGS.

Based on the above considerations, we get the algorithm in Figure 14.2. This algorithm is essentially equivalent to Algorithm M in [**?**] for the purpose of generating all $n$-tuples except for the consistency

```
Input: a distributed computation, a predicate B
Output: the lexicographically minimum CGS that satisfies B
        null if none exists;
var
 G:array[0 ... n] of integer initially ∀i : G[i] = 0;
 m:array[0 ... n] of integer initially m[i] = the number of events at P_i;

 if (B(G)) then return G;// visit the initial CGS

 while (true) do // generate the next global state
     i := n;
     while (G[i] = m[i]) // carry over
         G[i] := 0;
         i := i − 1;
     endwhile;
     if (i = 0) then return null;
     else G[i] := G[i] + 1;

     if (∀j, k : dep(G[j], k) ≤ G[k]) then // if G is consistent
         if (B(G)) then return G; // visit this global state
 endwhile;

 return null;
```

Figure 14.2: An Algorithm for traversal in lex order

check. The array $G$ denotes the vector of the current ideal and the array $m$ is used for the final ideal. To simplify programming, we use $G[0]$ and $m[0]$ as sentinel values with $m[0]$ equal to 1. This ensures that the nested *while* loop always terminates in a valid state.

The consistency check is performed by using *dep* function which encodes the poset as follows. $dep(G[j], k)$ returns the largest index of the event on $P_k$ on which the event $G[j]$ depends. If vector clocks are available, then $dep(G[j], k)$ is simply $G[j].v[k]$ where $v$ is the vector clock at $G[j]$.

The space complexity is $O(nE)$ required for storing the vector clock for each event in the poset for consistency check. The time required is for going through the product space and checking consistency of each of the global state generated. The number of tuples in the product space is $O(k^n)$ assuming that $k$ is the maximum number of events on any one process. Generating each tuple requires $O(n^2)$ amortized time giving us the time complexity of $O(k^n n^2)$.

The above algorithm has the disadvantage that it generates all possible global states and then checks for consistency for each one of them. We now modify the lex algorithm to avoid generating all global states. We assume that vector clocks are available with each event.

Let $nextLex(G)$ denote the CGS that is the successor of $G$ in the lex order. For example, in Figure 14.1, $nextLex(01) = 10$ and $nextLex(13) = 20$. It is sufficient to implement $nextLex$ function efficiently for enumeration of ideals in the lex order. One can set $G$ to the initial CGS $\langle 0, 0, ..., 0\rangle$ and then call the function $nextLex(G)$ repeatedly. We implement the function $nextLex(G)$ using two secondary functions *succ* and *leastConsistent* as described next.

We define $succ(G, k)$ to be the global state obtained by advancing along $P_k$ and resetting components for all processes greater than $P_k$ to 0. Thus, $succ(\langle 7, 5, 8, 4\rangle, 2)$ is $\langle 7, 6, 0, 0\rangle$ and $succ(\langle 7, 5, 8, 4\rangle, 3)$ is $\langle 7, 5, 9, 0\rangle$. Note that $succ(G, k)$ may not exist when there is no event along $P_k$, and even when it exists it may not be consistent.

The second function is $leastConsistent(K)$ which returns the least consistent global state greater than or equal to a given global state $K$ in the $\subseteq$ order. This is well defined as shown by the following lemma.

**Lemma 14.2** *The set of all consistent global states that are greater than or equal to $K$ in the CGS lattice is a sublattice. Therefore, there exists the least CGS $H$ that is greater than or equal to $K$.*

**Proof:** It is easy to verify that if two consistent global states $H_1$ and $H_2$ are both greater than or equal to $K$, then so is their union and intersection.

■

We now show how $nextLex(G)$ can be computed.

**Theorem 14.3** *Assume that $G$ is a CGS such that it is not the greatest CGS. Then,*

$$nextLex(G) = leastConsistent(succ(G, k))$$

*where $k$ is the index of the process with the smallest priority which has an event enabled in $G$.*

**Proof:** We define the following global states for convenience:
$K := succ(G, k)$
$H := leastConsistent(K)$, and
$G' := nextLex(G)$.
Our goal is to prove that $G' = H$.

$G'$ contains at least one event $f$ that is not in $G$; otherwise $G' \subseteq G$ and therefore cannot be lexically bigger. Choose $f \in G' - G$ from the highest priority process possible.

Let $e$ be the event in the smallest priority process enabled in $G$, i.e., $e$ is on process $P_k$. Let $proc(e)$ and $proc(f)$ denote the process indices of $e$ and $f$. We now do a case analysis.

*Case 1: $proc(f) < k$*
In this case, $e$ is from a lower priority process than $f$. We show that this case implies that $H$ is lexically smaller than $G'$. We first claim that $H \subseteq G \cup \{e\}$. This is because $G \cup \{e\}$ is a CGS containing $K$ and $H$ is the smallest CGS containing $K$. Now, since $H \subseteq G \cup \{e\}$ and $G'$ contains an event $f \in G' - G$ from a higher priority process than $e$, it follows that $H$ is lexically smaller than $G'$, a contradiction.

*Case 2: $proc(f) > k$*
Recall that event $e$ is on the process with the smallest priority that had any event enabled at $G$. Therefore, existence of $f$ in CGS $G'$ implies existence of at least another event in $G' - G$ at a process with higher priority than $e$. This contradicts choice of event $f$ because, by definition, $f$ is from the highest prioriy process in $G' - G$.

*Case 3: $proc(f) = k$.*
Then, $K \subseteq G'$ because both $G'$ and $K$ have identical events on process with priority $k$ or higher and $K$ has no events in lower priority processes. Since $G'$ is a CGS and $K \subseteq G'$, we get that $H \subseteq G'$ by definition of $H$. From Lemma 14.1, it follows that $H \leq_l G'$. But $G'$ is the next lecical state after $G$. Therefore, $H = G'$.

■

The only task left in the design of the algorithm is to determine $k$ and implement *leastConsistent* function. The following lemma follows from properties of vector clocks[Fid89, Mat89]. First, we determine that an event $e$ in enabled in a CGS $G$ if all the components of the vector clock for other processes in $e$ are less than or equal to the components of the vector clock in $G$. Secondly, to compute $leastConsistent(K)$ it is sufficient to take the component-wise maximum of the vector clock of all maximal events in $K$. Formally,

**Lemma 14.4**

1. *An event $e$ on $P_k$ is enabled in a CGS $G$ iff $e.v[k] = G[k] + 1$ and*
$$\forall j : j \neq k : e.v[j] \leq G[j]$$
2. *Let $H = leastConsistent(K)$. Then,*
$$\forall j : H[j] = max\{K[i].v[j] \mid 1 \leq i \leq n\}$$

Incorporating these observations, we get the algorithm in Figure 14.3. The outer *while* loop at line (1) iterates till all consistent global states are visited. If the current CGS $G$ satisfies the given predicate $B$, then we are done and $G$ is returned as the lexicographically minimum CGS. Lines (4)-(22) generate $nextLex(G)$. Lines (4)-(14) determine the lowest priority process $k$ which has an event enabled in $G$. The *for* loop on line (4) is exited when an enabled event is found at line (12). We are guaranteed to get an enabled event because $G$ is not the final CGS. Lines (8)-(12) check if the next event on $P_k$ is enabled. This is done using the vector clock. An event $e$ in enabled in a CGS $G$ iff all the events that $e$ depend on have been executed in $G$; or equivalently, all the components of the vector clock for other processes in $e$ are less than or equal to the components of the vector clock in $G$. This test is performed in lines (8)-(11). Lines (15) to (17) compute $succ(G, k)$. Finally, lines (18)-(22) compute $leastConsistent(succ(G, k))$.

Let us now analyze the time and space complexity of the above algorithm. The *while* loop iterates once per CGS of the computation. Each iteration takes $O(n^2)$ time due to nested *for*. Thus the total time taken is $O(n^2M)$. The algorithm uses variables $G, K, H$ and $m$ which requires $O(n)$ space. We also assume that the events are represented using their vector clocks.

Let $nextLex(G)$ denote the CGS that is the successor of G in the lex order. Then, it's sufficient to implement $nextLex$ function efficiently for enumeration of ideals in the lex order. Set $G$ to initial CGS $< 0, 0, \ldots, 0 >$, and call $nextLex(G)$ function repeatedly.
To implement the $nextLex(G)$, two new secondary functions $succ$ and $leastConsistent$, and a scheme to determine $k$ are introduced.

- $succ(G, k)$ : computes the global state by advancing on $P_k$ and resetting components for all processes greater than $P_k$ to 0.
- $leastConsistent(K)$ : computes the least consistent global state greater or equal to a given global state $K$ in the $\subseteq$ order. To compute $leastConsistent(K)$, it is sufficient to take the componentwise maximum of the vector clock of all maximal events in $K$.
- $k$ : $k$ is the least process on which some event $e$ is enabled at $G$. An event $e$ is enabled in $G$ if all the components of the vector clock for other processes in $e$ are less than or equal to components of the vector clock in $G$. Formally an event $e$ is enabled in $G$ iff, $e.v[i] = G.v[i]+1$ and $\forall j : j \neq i : e.v[j] \leq G.v[j]$

```
Algorithm nextLex(G):
begin
  k = least process on which some event is enabled at G;
  K = succ(G, k);
  H = leastConsistent(K);
  return H;
end.
```

The complexity of the above algorithm is $O(n^2)$, since to find such $k$ we need to iterate $n^2$ times at the worst case. Function $succ(G, k)$ has a complexity of $O(n)$, while $leastConsistent(K)$ has

```
lexTraverse(P, B)

Input: a distributed computation P, a predicate B
Output: the smallest CGS in lex order that satisfies B,
        null if none exists;
var
// current CGS
    G:array[1 ... n] of int initially ∀i : G[i] = 0;
// K = succ(G, k)
    K:array[1 ... n] of int;
// H = leastConsistent(K)
    H:array[1 ... n] of int;
// m[i] equals the number of events at P_i
    m:array[1 ... n] of int;

(1)  while (G ≤ m) do
(2)      if (B(G)) then return G;
(3)      if (G = m) then return null;

(4)      for k := n down to 1 do
             // if next event on P_k exists
(5)          if (G[k] ≠ m[k]) then
(6)              e := next event on P_k after G[k]
(7)              boolean enabled := true;
(8)              for j := 1 to n, j ≠ k do
(9)                  if e.v[j] > G[j] then
(10)                     enabled := false;
(11)             end// for;
(12)             if (enabled) break; //goto line (15);
(13)         end// if next event exists;
(14)     end// for;

         // compute K := succ(G, k)
(15)     K := G; //
(16)     K[k] := K[k] + 1; // advance on P_i
(17)     for j := k + 1 to n do K[j] := 0;

         // compute H := leastConsistent(K);
(18)     H := K; // initialize H to K
(19)     for i := 1 to n do
(20)         for j := 1 to n do
(21)             H[j] := max(H[j], K[i].v[j]);

(22)     G := H;
(23)end// while;
```

Figure 14.3: An Algorithm for Traversal in Lex Order

$O(n^2)$.

Let's form the algorithm $lexTraverse(P, B)$ by using $nextLex$ recursively, where $P$ is a distributed computation and $B$ is a predicate. Then the total time taken is $O(n^2M)$, and it requires $O(cn)$ space.

**Open Problem:** Is there a global state enumeration algorithm which is $O(1)$ per global state. Best known algorithm is $O(cM)$, where $c$ is the maximum degree of hasse diagram of the computation/poset.

## 14.3 Algorithms for BFS generation of Ideals

For many applications we may be interested in generating consistent global states in the BFS order, for example, when we want to generate elements in a single level of the CGS lattice. The lex algorithm is not useful for that purpose.

The naive approach that we can take is as follows. We keep two lists of consistent global states: *last* and *current*. To generate the next level of consistent global states, we set *last* to *current* and *current* to the set of global states that can be reached from *last* in one transition. Since a CGS can be reached from multiple global states, an implementation of this algorithm will result in either *current* holding multiple copies of a CGS or added complexity in the algorithm to ensure that a CGS is inserted in *current* only when it is not present. This problem occurs because their algorithm does not exploit the fact that the set of global states form a distributive lattice.

We now show an extension of their algorithm which ensures that a CGS is enumerated exactly once and that there is no overhead of checking that the CGS has already been enumerated (or inserted in the list). Our extension exploits the following observation.

**Lemma 14.5** *If $H$ is reachable from $G$ by executing an event $e$ and there exists an event $f \in G$ such that $f$ is maximal in $G$ and concurrent with $e$, then there exists a CGS $G'$ at the same level as $G$ such that $G' = G - \{f\} + \{e\}$ and $H$ is reachable from $G'$.*

**Proof:** Since $G$ is consistent and $f$ is a maximal event in $G$, it follows that $G - \{f\}$ is a CGS. If $e$ is enabled at $G$ and $f$ is concurrent with $e$, then $e$ is also enabled at $G - \{f\}$. Therefore, $G' = G - \{f\} + \{e\}$ is a CGS. $H$ is reachable from $G'$ on executing $f$.

■

Thus, to avoid enumerating $H$ from both $G$ and $G'$, it is sufficient to have a total order on all events and explore execution of an event $e$ from a global state $G$ iff $e$ is smaller than all maximal events in $G$ which are concurrent with $e$.

$$\{f \mid f \in G, f||e\} \cup \{e\}$$

Let $\sigma$ be a topological sort of all events which maps every event $e$ to $\sigma(e)$ a number from $1..E$. Now the rule to decide which events to explore from a CGS $G$ is simple. Let $e$ be any enabled event in $G$. We explore execution of $e$ on $G$ iff

$$\forall f \in maximal(G) : f||e \Rightarrow \sigma(e) < \sigma(f)$$

With this observation, our algorithm shown in Figure 14.4 keeps a queue $Q$ of the CGS. At every iteration, it removes the head of the queue $G$, checks if the global predicate is true on $G$. If it is, we are done. Otherwise, the algorithm inserts those successors of $G$ that satisfy the above rule.

```
var
 Q:set of CGS at the current level
      initially {(0, 0, ..., 0)};
 σ:a topological sort of the poset P;

 while (Q ≠ ∅) do
      G := remove_first(Q);
      if B(G) then return G;
      // generate CGS at the next level
      for all events e enabled in G do
           if (∀f ∈ maximal(G) : f||e ⇒ σ(e) < σ(f)) then
                H := G ∪ {e};
                append(Q, H);
      end //for;
 end //while;
```

Figure 14.4: An Extension of Cooper Marzullo Algorithm for BFS enumeration of CGS

We now analyze the time complexity of above algorithm. There are $O(M)$ iterations of the *while* loop. In each iteration, there may be at most $n$ events enabled per CGS. Determining whether an event is enabled takes at most $O(n)$ time. Determining whether it is smaller than all the maximal events in $G$ that are concurrent to it also takes $O(n)$ time. Thus the overall time complexity is $O(n^2M)$.

The main disadvantage of above algorithm is that it requires space at least as large as the number of consistent global states in the largest level set. Note that the largest level set is exponential in $n$ and therefore when using this algorithm for a large system we may run out of memory.

We now give two algorithms that use polynomial space to list the global states in the BFS order. The first algorithm is based in integer compositions and consistency checks and the second algorithm is based on using the DFS (or the lex) traversal multiple number of times to enumerate consistent global states in the BFS order.

The first algorithm for BFS traversal uses consistency check to avoid storing the consistent global states. The main idea is to generate all the global states in a level rather than storing them. Assume that we are interested in enumerating level set $r$. Any global state in level set $r$ corresponds to the total number of events executed by $n$ processes to be $r$. A *composition* of $r$ into $n$ parts corresponds to a representation of the form $a_1+a_2+\ldots+a_n = r$ where each $a_i$ is a natural number and the order of the summands is important. In our application, this corresponds to a global state $(a_1, a_2, ..., a_n)$ such that $\sum a_i = r$. There are many algorithms that enumerate all the compositions of an integer $r$ into $n$ parts (for example, the algorithm due to Nijenhuis and Wilf[**?**] (pp. 40-46) runs through the compositions in lexicographic order reading from right to left). For every composition, the

corresponding global state can be checked for consistency.

We now arrive at the algorithm in Figure 14.5

```
var
 G:array[1 ... n] of int
      initially G[1] = levelnum ∧ ∀i ≥ 2 : G[i] = 0;

 while (G ≠ null) do
      if isConsistent(G) then
           if B(G) then return G;
      // generate the next global state
      //use Neijenhuis and Wilf's algorithm;
      G := nextComposition(G)

 endwhile;
```

Figure 14.5: A Space Efficient algorithm to generate the level set of rank *levelnum*

The above algorithm has polynomial space complexity and can therefore generate level sets even for reasonably large systems.

The second algorithm exploits the fact that the DFS and the lex traversal can be done in polynomial space. We perform the DFS traversal for each level number $l$. During the DFS traversal we explore a global state only if its level number is less than or equal to $l$ and visit it (evaluate the predicate or print it depending upon the application) only if its level number is exactly equal to $l$.

The algorithm shown in Figure 14.6 generates one level at a time. In line (2) it reduces the computation to include only those events whose sum of vector clock values is less than or equal to *levelnum*. All other events can never be in a global state at level less than or equal to *levelnum*. In line (3) it performs space efficient lex traversal of the CGS lattice using the algorithm in Figure 14.3. The computation used is the reduced one and the global predicate that is evaluated is modified to include a clause that the CGS should be at level equal to *levelnum*. If no CGS is found, then we try the next level. Since the total number of levels is $O(E)$, we can enumerate the consistent global states in the BFS order in $O(En^2M)$ time and $O(nE)$ space. Note that our algorithm enumerates each level itself in the lex order.

## 14.4 Application to Combinatorial Enumeration

We first give the most straightforward application to enumeration of combinatorial objects. We consider the distributed computation shown in Figure 14.7(a). Each process $P_i$ executes a single event $e_i$. It is easy to verify that there is a bijection between every CGS of the computation and a subset of $X$. For example, when $n = 4$, the CGS $(0, 1, 0, 1)$ corresponds to the set $\{e_2, e_4\}$. The lex enumeration would correspond to enumeration of sets based on binary string representation. Figure 14.7(b) gives another computation which can also be used to enumerate all the subsets of

```
var
    G:array[1 ... n] of integer;

(1) for levelnum := 0 to E do
(2)     Q := {e ∈ P | ∑_i e.v[i] ≤ levelnum}
(3)     G := lexTraverse(Q, B ∧ (lvl = levelnum));
(4)     if (G ≠ null) then
(5)         return G;
(6) endfor;

(7) return null;
```

Figure 14.6: A Space Efficient algorithm for BFS Enumeration

{1..4}. In this example, the enumeration would result in the lex order (reversed). Each global state (or a cut) corresponds to a subset. For example, $G$ corresponds to the subset $\{4, 2, 1\}$. We say that $P_i$ has chosen element $k$ if the cut (global state) goes through the label $k$ for process $P_i$. In this computation $P_1$ chooses the biggest element in the set. By the design of the computation, $P_2$ can only choose an element that is strictly smaller than the element chosen by $P_1$ for the global state to be consistent. For example, the global state $H$ is not consistent because $P_2$ has also chosen 1 which is same as the element chosen by $P_1$. The choice of 0 by any $P_i$ denotes that the process does not choose any element. It can be verified that if $P_i$ chooses 0, then all higher numbered processes can only choose 0.

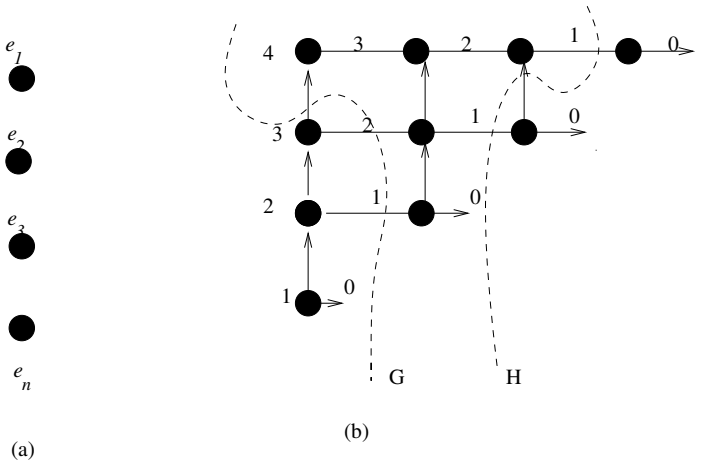



Figure 14.7: Computation for generating subsets of $X$

Figure 14.8 shows a computation such that all the subsets of $X$ of size $k$ are its consistent global states. There are $k$ processes in this computation and each process executes $n - k$ events. By the structure of the computation, if in a CGS $P_i$ has executed $j$ events, then $P_{i+1}$ must have also executed at least $j$ events. The correspondence between subsets of $X$ and global states can be

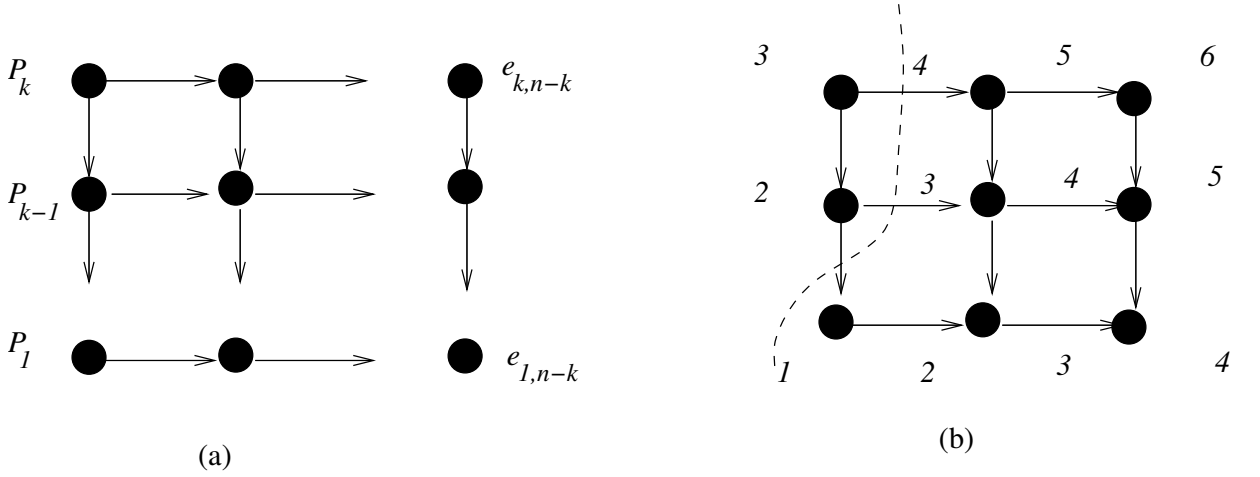



Figure 14.8: Computation for generating subsets of $X$ of size $k$

understood as follows. If process $P_i$ has executed $t$ events in the CGS, then the element $t+i$ is in the set $Y$. Thus process $P_1$ chooses a number from $1 \ldots n-k+1$ (because there are $n-k$ events); process $P_2$ chooses the next larger number and so on. Figure 14.8(b) gives an example of the computation for subsets of size 3 of the set [6]. The CGS, or the ideal, shown corresponds to the subset $\{1, 3, 4\}$.

A $k$-tuple of positive integers $\lambda = (\lambda_1, \ldots, \lambda_k)$ is an integer partition of $N$ if $\lambda_1 + \ldots + \lambda_k = N$ and for all $i$, $\lambda_i \geq \lambda_{i+1}$. The number of *parts* of $\lambda$ is $k$. An example of partition of 10 into 3 parts is $(4, 3, 3)$. An integer partition can be visualized as a *Ferrers diagram* or an array of squares in decreasing order with $\lambda_i$ squares in row $i$. The Ferrers diagram of the partition $(4, 3, 3)$ of 10 is shown in Figure 14.9(a). A partition $\lambda$ is contained in another partition $\delta$ if the number of parts of $\lambda$ is at most that of $\delta$ and $\lambda_i$ is less than or equal to $\delta_i$ for any $i$ between 1 and the number of parts of $\lambda$. For example, $(3, 3, 1)$ is less than $(4, 3, 3)$. Fix any partition $\lambda$. The set of all partitions that are less than or equal to $\lambda$ form the *Young's lattice* denoted by $Y_\lambda$. Figure 14.9(b) shows the computation for generating Young's lattice. By enumerating all consistent global states of the computation in lex order we can enumerate all partitions that are less than or equal to $\lambda$. Now consider the Young Lattice for $\lambda = (N, N, \ldots N \ times)$. The $N^{th}$ level set of this lattice corresponds to all integer partitions of $N$. By enumerating the $N^{th}$ level set we can enumerate all integer partitions of $N$.

Finally, we show that our algorithms can also be used to enumerate permutations. We first show a small computation that generates all permutations of $n$ symbols. The computation consists of $n-1$ processes. Process $P_i$ executes $i-1$ events. We use the inversion table[**?**] to interpret a CGS. The number of inversions of $i$ in a permutation $\pi$ is the number of symbols less than $i$ that appear to the right of $i$ in $\pi$. The way a permutation is generated from a CGS is as follows. We begin the permutation by writing 1. $P_1$ decides where to insert the symbol 2. There are two choices. These choices correspond to number of *inversions* introduced by 2. If we place 2 after 1, then we introduce zero inversions; otherwise we introduce one inversion. Proceeding in this manner we get that there is a bijection between the set of permutations and the consistent global states. The resulting algorithm is similar to Johnson-Trotter method of enumerating all permutations or

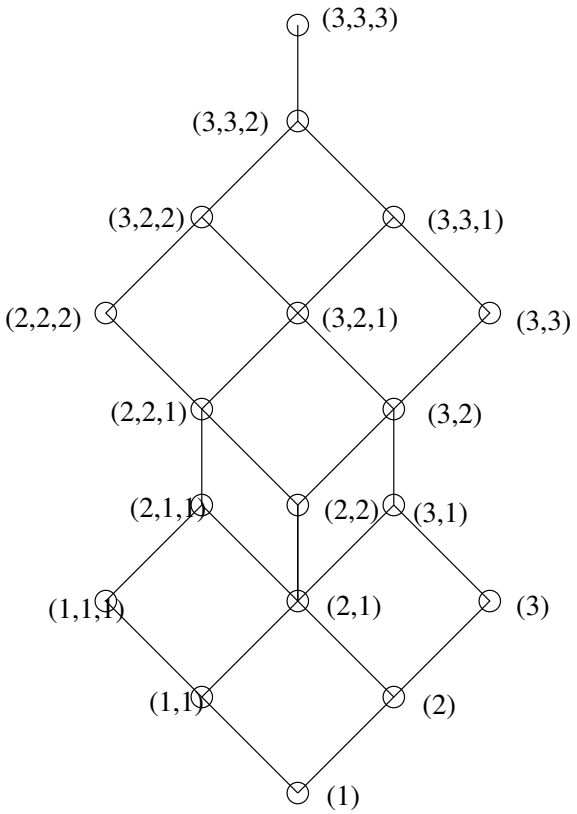



Figure 14.9: (a) A Ferrer diagram (b) A computation for generating Young's lattice

showing bijection between inversion table and permutations[**?**].

In many applications we are interested in subsets of combinatorial families. For example, we may be interested in enumerating all $k$ subsets of $[n]$ which do not have any consecutive numbers. By using methods of slicing one can compute posets such that its ideals correspond to precisely these subsets. Now we can use the algorithm for lex order traversal to enumerate all such subsets.

### 14.4.1 Exercises

1. Show that $H$ is indeed the next lexical state.

# Chapter 15

# Dimension Theory

## 15.1 Introduction

Dimension theory of ordered sets has been an active area of research and the reader will find a good survery of results in the book by Trotter. In this chapter we will cover only the most basic results and their applications.

**Definition 15.1** *An extension of a poset $P$ is a partial order on the elements of $P$ that contains all the relations of $P$. Let $P$ and $Q$ be partial orders on the same set $X$, then $Q$ is an extension of $P$ if $P \subseteq Q$, i.e., $x \leq y$ in $P \implies x \leq y$ in $Q$, for all $x, y \in X$.*

**Definition 15.2** *A linear extension is an extension that is a chain.*

**Definition 15.3** *The intersection of partial orders on a given set is the set of relations that appears in each of them. The intersection of two partial orders is a partial order, and a finite poset is the intersection of its linear extensions.*

To generate a linear extension, we iteratively record (and delete) a minimal element of the subposet that remains. In the algorithmic literature, the resulting list, a linear extension, is called a *topologicalordering*. All linear extensions arise in this way.

**Definition 15.4** *A realizer of $P$ is a set of extensions whose intersection is $P$. A family $R = \{L_1, L_2, \ldots, L_t\}$ of linear orders on $X$ is called a realizer of a partial order $P$ on $X$ if $P = \cap R$.*

**Definition 15.5** *The dimension of $(X, P)$,denoted as $dim(P)$, is the size of the smallest realizer. When defining realizers of posets, repetition of the linear extension in the family is allowed. Therefore, if a poset has dimension $t$, then for every $s \geq t$ it has a realizer $R = \{L_1, \ldots, L_s\}$.*

**Example** For the poset $P = (X, P)$ shown in figure 15.1, there are 14 extensions of $P$ (counting $P$ as one of them). Of these extensions, 5 are linear extensions. Clearly, $dim(X, P) \geq 2$ since it is not a chain. On the other hand, $dim(X, P) \leq 2$ since $R = \{L_4, L_5\}$ is a realizer of P.

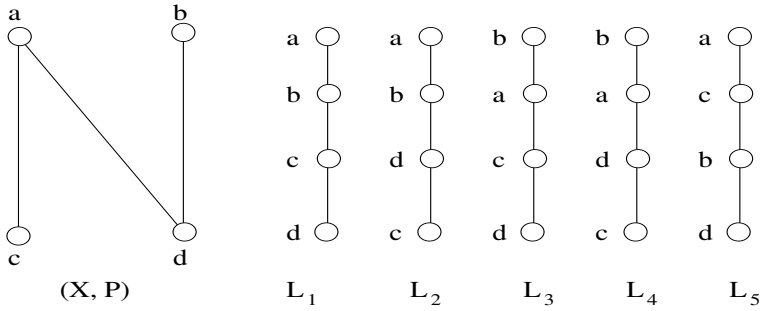



Figure 15.1: Realizer example

### 15.1.1 Standard example:

For $n \geq 3$, let $S_n = (X, P)$ be the height two poset with $X = a_1, a_2, \ldots, a_n \cup b_1, b_2, \ldots, b_n$, and $\forall a_i, b_j \in X$, $a_i < b_j$ in $P$ iff $i \neq j$. Then, the poset $P$ is called *standard example* of an $n$-dimensional poset. An example is shown in figure 15.2 for n=5.

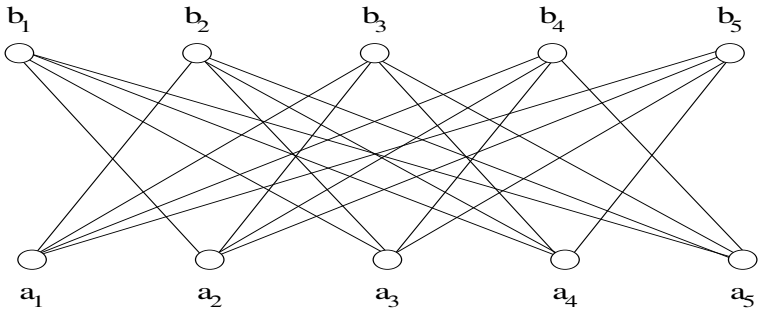



Figure 15.2: Standard example

**Theorem 15.6** *$dim(S_n) = n$.*

**Proof:**

- $dim(S_n) \leq n$:
  For each $i = 1, 2, \ldots, n$, define a linear order $L_i$ on $X$ by
  $L_i = [a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, b_i, a_i, b_1, b_2, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n]$. Then $R = \{L_1, L_2, \ldots, L_n\}$ is a realizer, so $dim(S_n) \leq n$.

- $dim(S_n) \geq n$:
  Let $R = \{L_1, L_2, \ldots, L_t\}$ be any realizer of $S_n$. Define a function $f : [n] \rightarrow [t]$ as follows. Note that $a_i || b_i$ for each $i \in [n]$, so we may choose $f_i$ as some $j \in [t]$ for which $b_i < a_i$ in $L_j$.
  **Claim:** $f$ is injective.
  Suppose contrary that there exists a pair $i, j$ with $1 \leq i < j \leq n$ and $f_i = f_j = s$. Then $b_i < a_i$ in $L_s$, and $b_j < a_j$ in $L_s$. However, $a_j < b_i$ and $a_i < b_j$ in $P$, so $a_j < b_i$ and $a_i < b_j$ in $L_s$. Thus, $a_i < b_j < a_j < b_i < a_i$ in $L_s$ which is clearly false.
  Since $f$ is an injection, $dim(S_n) \geq n$.

**Theorem 15.7** *Interval order can have unbounded dimension.*

**Lemma 15.8** *(Hiraguchi) Let $P = (X, P)$ be a poset and let $C \subseteq X$ be a chain. Then there exist linear extensions $L_1, L_2$ of $P$ so that:*

- *(1) $y < x$ in $L_1$ for every $x, y \in X$ with $x \in C$ and $x||y$ in $P$.*
- *(2) $y > x$ in $L_2$ for every $x, y \in X$ with $x \in C$ and $x||y$ in $P$.*

**Theorem 15.9** *(Dilworth) Let $P = (X, P)$ be a poset. Then $dim(P) \leq width(P)$.*

**Proof:** Let $n = width(P)$. By Dilworth's chain partitioning theorem, there exists a partition $X = C_1 \cup C_2 \cup \cdots \cup C_n$, where $C_i$ is a chain for $i = 1, 2, \ldots, n$. For each $i$, we use Lemma 1 to choose a linear extension $L_i$ of $P$ so that $C/X$ in $L_i$. We claim that $R = \{L_1, L_2, \ldots, L_n\}$ is a realizer of $P$. To show that$P = \cap_{i=1}^{t} L_i$, it suffices to show that for every $(x, y) \in P$, there exits $j \in [t]$ with $y, x$ in $L_j$. The desired value of $j$ is determined by choosing a chain $C_j$ for which $x \in C_j$.

**Theorem 15.10** *(Hiraguchi) Let $P = (X, P)$ be a poset and let $C \subseteq X$ be a chain with $X - C \neq \emptyset$. Then*

$$dim(X, P) \leq 2 + dim(X - C, P).$$

**Theorem 15.11** *(Hiraguchi) Let $P = (X, P)$ be a poset with $|X| \geq 2$, and $x \in X$. Then*

$$dim(X, P) \leq 2 + dim(X - \{x\}, P).$$

**Theorem 15.12** *Let $P = (X, P)$ be a poset with $|X| \geq 4$. Then*

$$dim(X, P) \leq \frac{|X|}{2}.$$

**Conjecture:** $\forall$ posets $P = (X, P)$ st. $|X| \geq 4$, $\exists x, y \in X$ :

$$dim(X, P) \leq 1 + dim(X - \{x, y\}, P).$$

### 15.1.2 Critical pair in the poset

**Definition 15.13** *Let $P = (X, P)$ be a poset and let $(x, y) \in inc(X, P)$. An ordered pair $(x, y)$ is called a critical pair (also nonforced pair) in $P$ if:*

1. *$D(x) \subseteq D(y)$, and // $z < x$ in $P$ implies $z < y$ in $P$*

*2.* $U(y) \subseteq U(x)$. *//* $y < w$ *in* $P$ *implies* $x < W$ *in* $P$

Observe that if $x < y$, above statements hold trivially. However, a critical pair must also be *incomparable*. Let $crit(X, P)$ denote the set of all critical pairs in the poset $P$. Note that $crit(X, P) = \emptyset$ if and only if $P$ is a chain.
**Example** In figure 15.3, while $(a_1, b_1)$ is a critical pair, $(b_1, a_1)$ is not.

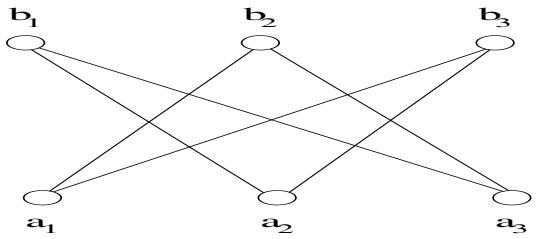



Figure 15.3: Critical pair

**Proposition**(Rabinovitch and Rival)
Let $P = (X, P)$ be a poset and $R$ be a family of linear extensions of $P$. Then the following statements are equivalent.

1. $R$ is a realizer of $P$.
2. For every critical pair (x,y), there is some $L \in R$ for which $y < x$ in $L$.

### 15.1.3 Bounds on the Dimension of a Poset

In this section, we give bounds on the dimension of a poset in terms of widths of its certain subposets. Let $J(P)$ be the subposet formed by the join-irreducibles and $Dis(P)$ be the subposet formed by dissectors of $P$. We now show the following result.

**Theorem 15.14 (Reading)** *For a finite poset* $P$*,*

$$width(Dis(P)) \leq dim(P) \leq width(J(P))$$

**Proof:**

■

### 15.1.4 Encoding Partial Orders Using Rankings

Observe that a ranking can be encoded very efficiently. Given the rank of each element, one can determine the order. Therefore, if all labels of a ranking map to consecutive numbers, then maximum number of bits required are $log_2 r$ where $r$ is the number of ranks in the poset. Note that this encoding drops the distinction between elements which have the same order relationship with all other elements. Thus, two elements $x$ and $y$ have the same code $f(x) = f(y)$ iff for any element

$z$, (1) $x < z$ iff $y < z$, and (2) $z < x$ iff $z < y$. This technique allows more efficient encoding of the partial order than the use of chains as in dimension theory. At an extreme, the range of $f$ may be finite even when the domain of $f$ is infinite. For example, the order in Figure 15.4 where { all even numbers } < { all odd numbers } on natural numbers can be encoded by assigning 0 to all even numbers and 1 to all odd numbers. Such a poset cannot be assigned codes using the classical dimension theory.

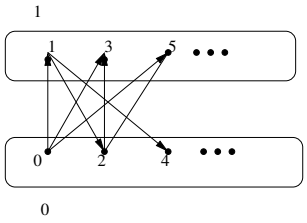



Figure 15.4:

We write $x \leq_s y$ if $x \leq y$ in ranking $s$, and $x <_s y$ if $x < y$ in ranking $s$.

**Definition 15.15 (Ranking Realizer)** *For any poset $(X, P)$, a set of rankings $\mathcal{S}$ is called a ranking realizer iff $\forall x, y \in X : x < y$ in $P$ if and only if*

1. *$\forall s \in \mathcal{S} : x \leq_s y$, and*
2. *$\exists t \in \mathcal{S} : x <_t y$.*

The definition of less-than relation between two elements in the poset based on the rankings is identical to the less-than relation as used in vector clocks. This is one of the motivation for defining ranking realizer in the above manner. A ranking realizer for the poset in Fig. **??** is given by two rankings

$$s_1 = \{(c), (d, a), (b)\} \qquad s_2 = \{(d, b), (c, a)\}$$

There are two important differences between definitions of ranking realizers and chain realizers. First, if $\mathcal{R}$ is a chain realizer of a poset $P$, then $P$ is simply the intersection of linear extensions in $\mathcal{R}$. This is not true for a ranking realizer (see Fig. **??**). Secondly, all the total orders in $\mathcal{R}$ preserve $P$, i.e., $x < y$ in $P$ implies that $x < y$ in all chains in $\mathcal{R}$. This is not true for ranking realizer. For example, $d < a$ in poset $P$ of Fig. **??**, but $(d, a)$ appears as a rank in the ranking $s_1$. We are only guaranteed that $a$ will not appear lower than $d$ in any ranking - they may appear in the same rank.

Now, analogous to the dimension we define

**Definition 15.16 (Ranking Dimension)** *For any poset $(X, P)$, the ranking dimension of $(X, P)$, denoted by $sdim(X, P)$, is the size of the set $\mathcal{S}$ with the least number of rankings such that $\mathcal{S}$ is a ranking realizer for $(X, P)$.*

**Example 15.17** *Consider the standard example $S_m$. The following function $f$ can be used to create a ranking realizer of $S_m$. For all $k, i = 1, 2, \ldots, m$,*

$$f_k(a_i) = \begin{cases} 0 & \text{if } k \neq i \\ 1 & \text{otherwise} \end{cases}$$

$$f_k(b_i) = \begin{cases} 0 & \text{if } k = i \\ 1 & \text{otherwise} \end{cases}$$

*For example,*

$$a_1 = (1,0,0,\ldots,0), \quad b_1 = (0,1,1,\ldots,1)$$
$$a_2 = (0,1,0,\ldots,0), \quad b_2 = (1,0,1,\ldots,1)$$

*In this example, the length of each ranking is 2 and thus each element requires only $m$ bits for encoding. If we use classical dimension based on total orders, each element would require $m * \log m$ bits.*

**Example 15.18** *Consider the poset $(X, P)$ as follows.*
$X = \{\emptyset, \{a\}, \{b\}, \{a,b\}, \{a,c\}, \{a,b,c\}\}$
$P = \{(A,B) \in X \times X : A \subseteq B\}$.

*A ranking realizer for the poset can be obtained as follows. For each set $A \in X$, we use a bit vector representation of the set A. Thus, $\{a,c\}$ is represented by $(1,0,1)$ and the set $\{a,b\}$ is represented by $(1,1,0)$. This representation gives us a ranking realizer with three rankings such that every ranking has exactly two ranks.*

The concept of ranking realizer has the advantage over chain realizer that it generally requires less number of bits to encode a partial order by using rankings than by using chains. Formally, consider the following problem. Given a partial order $(X, P)$, define a coding function $code : X \longrightarrow \{0,1\}^k$ and a binary relation $<$ on codes such that $\forall x, y \in X : x < y$ in $P \iff code(x) < code(y)$. Note that the order relation may be any arbitrary order (not necessarily vector order). The only requirement is that it can only use the bits in $code(x)$ and $code(y)$ to determine the order.

Using dimension theory, if a partial order has dimension $k$, then it can be encoded using $k * log(n)$ bits. Let $cbits(P)$ denote the minimum number of bits required to code an element when chain realizer is used. Clearly,

$$cbits(P) = dim(P) * \lceil \log n \rceil.$$

Ranking realizers result in a lower number of bits for encoding. Let $sbits(P)$ denote the minimum number of bits required to code an element using a ranking realizer. Formally,

$$sbits(P) = \min_{\mathcal{S}} \sum_{s \in \mathcal{S}} \lceil \log(height(s)) \rceil$$

where $\mathcal{S}$ ranges over all ranking realizers of $P$. In other words, we consider all ranking realizers and use the one which results in the least number of bits.

The number of bits used by a ranking realizer is always less than or equal to that used by a chain realizer, i.e.,

**Theorem 15.19** *For all posets $(X, P)$,*

$$sbits(P) \leq cbits(P).$$

**Proof:** From Theorem **??**, $sdim(P) \leq dim(P)$ for all $P$. To encode a ranking of length $l$, we require $\lceil \log l \rceil$ bits. The length of each ranking is clearly less than or equal to $n = |X|$. The result follows from definition of $cbits(P)$.

■

Note that for any non-ranking poset, given a ranking realizer with $k$ coordinates, there is also a chain realizer with $k$ coordinates. The difference lies in the number of bits required to code a single coordinate. Given a ranking realizer $R$, if $R$ has $k$ rankings each of length less than or equal to $l$, then $(X, P)$ can be coded using $k \log l$ bits. Depending upon the structure of the poset, $l$ may be much smaller than $n$ as seen for the case of the standard example. We have shown that ranking encoding always outperform chain encoding. Although chain encoding is concise when the poset has small dimension, it is inefficient for posets with large dimensions. For the standard example chain encoding takes $n/2 * \log n$ bits per element.

Another popular method to encode a partial order is to use "adjacency matrix" type representation. Any partial order can be coded using $log(n) + n$ bits per element using "adjacency matrix" type representation. For every element, we store a binary array of size $n$. Further, each element is assigned a unique index into the array. Let $index(x)$ be the index of $x$ in $1..n$ and $x.v$ be the $n$ bit array for element $x$. We set $x.v[index(y)]$ to 1 if $x < y$ and to 0 otherwise. It is easy to determine the relationship between $x$ and $y$ given $index(x)$, $x.v$, $index(y)$, and $y.v$. We now show that ranking based encoding is always better than adjacency matrix based coding.

**Theorem 15.20** *For all posets* $(X, P)$

$$sbits(P) \leq n.$$

**Proof:** We show a ranking realizer of $n$ rankings with each ranking of size 2. Let $X = \{x_1, x_2, ..., x_n\}$. We construct a ranking $s_i$ for each $x_i$, $1 \leq i \leq n$. Every ranking $s_i$ has two ranks. The upper rank consists of $x_i$ and all elements greater than $x_i$. The lower rank consists of all elements smaller than $x_i$ and all elements that are incomparable to $x_i$.

We now show that this is a proper ranking realizer of $P$. Consider any two elements $x_i$ and $x_j$. By symmetry, there are only two cases:

*Case 1*: $x_i < x_j$
In this case, $x_i$ is less than $x_j$ in ranking $s_j$. In all other rankings $x_i$ appears either in the same rank or in the lower rank as $x_j$. Note that if $x_i$ appears in the upper rank for ranking $s_k$ for any $k$, then $x_j$ also appears in the upper rank.

*Case 2*: $x_i || x_j$
In this case $x_i$ is less than $x_j$ in ranking $s_j$, but $x_j$ is less than $x_i$ in ranking $s_i$.

■

Note that we could have used a "dual" ranking realizer in the proof of Theorem 15.20 in which each ranking $s_i$ has two ranks as follows. The upper rank consists of all elements bigger than $x_i$ and all

elements that are incomparable to $x_i$. The lower rank consists of $x_i$ and all elements smaller than $x_i$. It can be easily verified that this leads to a proper ranking realizer.

We can improve the bound for $sbits(P)$ for bipartite posets. We call a poset $\mathbf{P} = (X, P)$ *bipartite* if there is no chain in the poset involving more than two elements. In that case, the ground set $X$ can be partitioned into two disjoint subsets $L$ and $U$ so that $P \subseteq L \times U$ and we write $\mathbf{P} = (X, P) = (L, U, P)$. Note that the partition of $X$ into $L$ and $U$ may not be unique.

**Theorem 15.21** *For all bipartite posets* $(X, P)$

$$sbits(P) \leq n/2 + \log n.$$

**Proof:** Assume without loss of generality that $|L| \leq n/2$. We can use dual constructions when $|U| \leq n/2$.

For each $x \in L$, we construct a ranking $s_x$ as in the proof for Theorem 15.20. These rankings give information about any pair of elements such that both are in $L$. We create one more ranking $t$ which will allow us to infer relationship for other types of pairs. The lowest rank in $t$ consists of all elements in $L$. The upper part of $t$ is constructed as follows. We first derive the poset $Q$ on the set $U$ which is given by the ranking realizer formed by rankings $\{s_x | x \in L\}$. If a set of elements in $U$ are indistinguishable in all rankings (i.e., they appear in the smae rank in all rankings), we combine all those elements into a single element called a *group* element. Now we consider any chain consistent with dual of the poset $Q$. (The *dual* of a partial order $Q$ on $Y$ is defined by $\{(y, x) \mid (x, y) \in Q\}$). From this chain, we construct a ranking by replacing every group element by the set of elements in that group. This ranking is the upper part of the ranking $t$ and the lowest rank of $t$ is all elements in $L$. The length of the ranking $t$ is at most $\log n$.

We show this construction gives a ranking realizer for $(X, P)$. As before, there are two cases.

*Case 1*: $x_i < x_j$
In this case $x_i \in L$ and $x_j \in U$. By our construction, $x_i$ is less than $x_j$ in ranking $t$. $x_i$ appears in the upper rank in exactly one ranking $s_i$, but in that ranking $x_j$ is also in the upper rank.

*Case 2*: $x_i || x_j$
If both $x_i$ and $x_j$ are in $L$, then we know that $x_i$ is less than $x_j$ in ranking $s_j$, and $x_j$ is less than $x_i$ in ranking $s_i$.

If both of them are in $U$, we have the following cases. If $x_i$ is less than $x_j$ in first $|L|$ rankings, then by our construction of $t$ it is greater than $x_j$ in ranking $t$. If both of them are in the same rank in all rankings, then we ensure that they are in the same rank in $t$. Otherwise, we know that in one of the rankings among $L$ rankings $x_i$ is less than $x_j$ and in some other ranking $x_j$ is less than $x_i$.

The last case is when $x_i$ is in $L$ and $x_j$ is in $U$ (or vice-versa). In this case $x_j$ is less than $x_i$ is ranking $s_i$ and $x_i$ is less than $x_j$ in ranking $t$.

■

Our constructions of ranking realizers so far were dependent on the number of elements. We now discuss a method that may result in a more efficient construction when the width of the poset is small.

**Theorem 15.22** *Every partial order $(X, P)$ on $n \geq 2$ elements can be encoded using a ranking realizer in at most $\log(height(P) + 1) * width(P)$ bits.*

**Proof:** For convenience, let $w = width(P)$. We use Dilworth's chain covering theorem which states that $(X, P)$ can be partitioned into $w$ chains $C_1, C_2, ..., C_w$. We then use the transitively reduced diagram of $(X, P)$ with $w$ processes as given by the chain decomposition. Further, we use Fidge and Mattern's algorithm to assign vector timestamp for each event when the poset diagram is viewed as a computation. These vector timestamps determine a ranking realizer with $w$ coordinates such that no coordinate is greater than $height(P) + 1$.

■

We now define the notion of ranking length to derive a lower bound on dimension of any poset. The length of a realizer $S$ for the poset $P$, denoted by $slength(P, \mathcal{S})$, is defined as the length of the longest ranking in the ranking realizer $\mathcal{S}$ of $P$. Let $slength(P)$ denote the length of the longest ranking in the ranking realizer with minimum number of rankings. The following definition is useful in determining the lower bound on the dimension.

**Definition 15.23** *Let $(X, P)$ be any poset. For $x, y \in X$, we say that $x$ is order-equivalent to $y$ (denoted by $x \sim y$ ) iff $x$ is incomparable to $y$ and for all $z \in X : x < z \equiv y < z$ and for all $z \in X : z < x \equiv z < y$*

Let $numeq(P)$ denote the number of equivalence classes of the relation $\sim$. The following lemma shows the relationship among $dim(P), slength(P)$ and $numeq(P)$.

**Lemma 15.24** $dim(P) \geq \log(numeq(P)) / \log(slength(P))$.

**Proof:** The proof follows from the fact that the total number of codes is equal to $slength(P)^{dim(P)}$. Further, two elements in different equivalence classes cannot have the identical code. This implies that $slength(P)^{dim(P)} \geq numeq(P)$.

■

## 15.2 Rectangular Dimension

In our approach, we use posets with dimension at most two—called *two-dimensional* posets—as "building blocks" for realizing a given poset. For convenience, we refer to two-dimensional posets as *rectangles*. An element $x$ is less than another element $y$ in the given partial order if and only if $x$ is less than $y$ in at least one of the rectangular orders and $y$ is not less than $x$ is any of the rectangular orders. The set of rectangular orders that realizes a given partial order constitutes its *rectangular realizer*. Also, the *rectangular dimension* of a poset is the least number of rectangular orders needed to realize the corresponding partial order. Clearly, by definition, the rectangular dimension of a poset is one if and only if its dimension is at most two. Trivially, the rectangular dimension of a poset is upper bounded by its dimension.

It turns out that there are posets with arbitrarily high dimension but only constant rectangular dimension. As an illustration, consider the family of bipartite posets called *standard examples*. The standard example $\mathbf{S}_n$ for $n \geq 3$ is the poset induced by the 1-element and $(n-1)$-element subsets of $n$ distinct elements when ordered by set containment. The graph representation of $\mathbf{S}_5$, for example, is shown in Figure **??**. In the figure, all edges are directed upwards. It can be proved that a standard example has "large" dimension [**?**, Chapter 1]. Specifically, the dimension of $\mathbf{S}_n$ is given by $n$ for each $n \geq 3$. This implies that $O(n \log n)$ bits per element are required to encode $\mathbf{S}_n$ using the dimension theory. On the other hand, we prove in this paper that the rectangular dimension of $\mathbf{S}_n$ is two for each $n \geq 3$. Each rectangle can be encoded using $O(\log n)$ bits. Therefore using rectangles leads to a much more efficient representation of $\mathbf{S}_n$. We further prove that the rectangular dimension of the generalized crown $\mathbf{S}_n^k$ for $n \geq 3$ and $k \geq 0$ [**?**, Chapter 2], which is a generalization of the standard example, is also two. Its dimension, however, is given by $\lceil 2(n+k)/(k+2) \rceil$. Note that encoding $\mathbf{S}_n$ and $\mathbf{S}_n^k$ requires a large number of bits per element (specifically, $O(n \log n)$ and $O(n \log(n+k))$, respectively) using adjacency list representation as well.

we formally define the notions of rectangle, rectangular realizer and rectangular dimension of a poset.

**[rectangle] A *rectangle* is a two-dimensional poset.**

When a poset $\mathbf{P} = (X, P)$ is a rectangle, we call $P$ as a *rectangular order*. For a rectangular order $R$, we use $R.1$ and $R.2$ to refer to two total orders that realize $R$. In case the dimension of $R$ is one, both $R.1$ and $R.2$ refer to the same total order. Clearly, $R = R.1 \cap R.2$. A chain as well as an antichain is a rectangle.

**[rectangular realizer] Let $\mathbf{P} = (X, P)$ be a poset. A family $\mathcal{R} = \{R_1, R_2, \ldots, R_t\}$ of rectangular orders on $X$ is called a *realizer* of $P$ on $X$ (also, $\mathcal{R}$ *realizes* $\mathbf{P}$) if for every $x, y \in X$, $x < y$ in $P$ if and only if $y \not< x$ in $R_i$ for each $i \in [1, t]$ and $x < y$ in $R_j$ for some $j \in [1, t]$.**

If $x \| y$ in $P$, then in a rectangular realizer $\mathcal{R}$ of $P$ two cases are possible. Either $x \| y$ in all rectangular orders in $\mathcal{R}$, or $x < y$ in some rectangular order in $\mathcal{R}$ and $y < x$ in some other rectangular order in $\mathcal{R}$. On the other hand, if $x < y$ in $P$, then $x < y$ in some rectangular order in $\mathcal{R}$ and $x < y$ or $x \parallel y$ in all other rectangular orders in $\mathcal{R}$. The notion of rectangular dimension can now be defined as follows:

**[rectangular dimension] The *rectangular dimension* of a poset $\mathbf{P} = (X, P)$, denoted by $\mathrm{rdim}(X, P)$ (or $\mathrm{rdim}(\mathbf{P})$), is the least positive integer $t$ for which there exists a family $\mathcal{R} = \{R_1, R_2, \ldots, R_t\}$ of $t$ rectangular orders on $X$ so that $\mathcal{R}$ realizes $\mathbf{P}$.**

As an example, the rectangular dimension of the poset depicted in Figure **??** is two. The two rectangular orders realizing it are given by $\{(a_1 < a_2 < a_3 < b_1 < b_2 < b_3), (a_3 < a_2 < a_1 < b_3 < b_2 < b_1)\}$ and $\{(b_1 < a_2 < b_2 < a_3 < b_3 < a_1), (b_3 < a_1 < b_2 < a_3 < b_1 < a_2)\}$.

The rectangular dimension of a poset and its dual are identical because the dual of a two-dimensional poset is again a two-dimensional poset. The notions of rectangular realizer and rectangular dimension defined for a poset can be generalized to any (acyclic) relation on a ground set. For a collection of rectangular orders $\mathcal{R} = \{R_1, R_2, \ldots, R_t\}$, let $\mathrm{rel}(\mathcal{R})$ denote the relation realized by $\mathcal{R}$. For example, consider the rectangular orders $\{(a < b < c), (c < a < b)\}$ and $\{(a < b < c), (b < c < a)\}$.

The relation realized by the two orders collectively is given by $\{(a,b),(b,c)\}$. Note that the relation is not transitive because it does not contain the ordered pair $(a,c)$. Next, we define two concepts that we use when deriving rectangular realizers for posets using the point and order decomposition methods. Let $X = Y \cup Z$ be a partition of $X$, and let $Q$ and $R$ be rectangular orders on $Y$ and $Z$, respectively. Then $P = Q \cup R$ is also a rectangular order on $X$. Evidently, two chains realizing $P$ are given by $P.1 = Q.1 < R.1$ and $P.2 = R.2 < Q.2$. Moreover, $P$ satisfies the following properties:

1. $P(Y) = Q$ and $P(Z) = R$, and
2. $y \parallel z$ in $P$ for all $y \in Y$ and $z \in Z$.

**[disjoint composition] Let $X = Y \cup Z$ be a partition of $X$, and let $Q$ and $R$ be rectangular orders on $Y$ and $Z$, respectively. Then we say that the rectangular order $P = Q \cup R$ on $X$ is obtained by *disjoint composition* of the rectangular orders $Q$ on $Y$ and $R$ on $Z$.**

**[non-interference] A rectangular order $R$ is said to be *non-interfering* with a partial order $P$ if $R \subseteq P$.**

Also, a rectangular realizer is *non-interfering* with a partial order if every rectangular order in the realizer is non-interfering with the partial order. In other words, if two elements are incomparable in the partial order, they are also incomparable in all rectangular orders in the realizer. In that case, the partial order is given by the union of all rectangular orders in the realizer. Trivially, every partial order (even a relation) has a non-interfering rectangular realizer.

## 15.3 Point Decomposition Method and its Applications

### 15.3.1 The Main Idea

Given a poset $\mathbf{P} = (X, P)$, we partition the ground set $X$ into two subsets: $Y$ and $X \setminus Y$. We first compute rectangular realizers, say $\mathcal{Q}$ and $\mathcal{R}$, of the two induced subposets $(Y, P(Y))$ and $(X \setminus Y, P(X \setminus Y))$, respectively. A rectangular realizer $\mathcal{S}$ of $P(Y) \cup P(X \setminus Y)$ on $X$ can then be obtained by disjoint composition of $\mathcal{Q}$ and $\mathcal{R}$; each rectangular order of $\mathcal{S}$ is obtained by disjoint composition of corresponding rectangular orders of $\mathcal{Q}$ and $\mathcal{R}$ (padding can be done if necessary). Finally, we compute a rectangular realizer, say $\mathcal{T}$, of the relation $P \setminus (P(Y) \cup P(X \setminus Y))$ on $X$. To guarantee that $\mathcal{S} \cup \mathcal{T}$ constitutes a rectangular realizer of $P$ on $X$, it suffices to ensure that $\mathcal{T}$ is non-interfering with $P$. Note that it is not necessary that the relation realized by $\mathcal{T}$, given by $\mathrm{rel}(\mathcal{T})$, be exactly $P \setminus (P(Y) \cup P(X \setminus Y))$. It is sufficient that $P \setminus (P(Y) \cup P(X \setminus Y)) \subseteq \mathrm{rel}(\mathcal{T}) \subseteq P$. We often choose $Y$ such that $\mathcal{T}$ consists of a single rectangular order.

Our results in this section are based on the notion of *indistinguishable elements*. Let $\mathbf{P} = (X, P)$ be a poset and consider an element $x \in X$ and a subset $Y \subseteq X$. We denote the subset of elements in $Y$ that are less than $x$ in $P$, that is, $\{y \in Y \mid y < x \text{ in } P\}$ by $D(x, Y)$ (called the *down set* of $x$ in $Y$). Similarly, the subset of elements in $Y$ that are greater than $x$ in $P$, that is, $\{y \in Y \mid x < y \text{ in } P\}$ is denoted by $U(x, Y)$ (called the *up set* of $x$ in $Y$). For convenience, we abbreviate $D(x, X)$ by $D(x)$ and $U(x, X)$ by $U(x)$.

**[indistinguishable elements] Two elements $x, y \in X$ are said to be *indistinguishable* with respect to $Y$ in $P$ if $D(x, Y) = D(y, Y)$ and $U(x, Y) = U(y, Y)$.**

The elements of $X \setminus Y$ can be partitioned into equivalence classes such that elements in the same class are mutually indistinguishable with respect to $Y$ in $P$. These equivalence classes are referred to as *Y–indistinguishable classes* of $X \setminus Y$ in $P$. The class for which $D(x, Y) = U(x, Y) = \emptyset$ for each element $x$ in the class is called *Y–disconnected class*. (No element in the Y–disconnected-class is comparable to any element in $Y$.) When deriving the rectangular realizer $\mathcal{T}$ to represent $P \setminus (P(Y) \cup P(X \setminus Y))$ described earlier, it suffices to consider at most one representative element from every Y–indistinguishable class.

For example, for the poset shown in Figure **??**, $D(b_1, \{a_3, b_3\}) = \{a_3\}$. Furthermore, there are three $\{a_3, b_3\}$–indistinguishable classes of $\{a_1, a_2, b_1, b_2\}$, namely $\{b_1\}$, $\{a_1, b_2\}$ and $\{a_2\}$. The class $\{a_1, b_2\}$ is the $\{a_3, b_3\}$–disconnected class.

### 15.3.2 Applications

In this section, we present some removal theorems and later use them to provide bounds on rectangular dimension of various posets.

#### Removal Theorems

Out first theorem is based on the notion of *critical subposet*. A pair $(x, y)$ from $X$ with $x \parallel y$ in $P$ forms a *critical pair* in the subposet $(Y, P(Y))$ if $D(x, Y) \subseteq D(y, Y)$ and $U(x, Y) \supseteq U(y, Y)$. A subposet $(Y, P(Y))$ is called a *critical subposet* of the poset $(X, P)$ if for every incomparable pair $(x, y)$ from $Y$ either $(x, y)$ or $(y, x)$ forms a critical pair in the subposet $(X \setminus Y, P(X \setminus Y))$.

**Theorem 15.25 (critical subposet removal theorem)** *Let $(Y, P(Y))$ be a critical subposet of the poset $\mathbf{P} = (X, P)$ where $Y \subsetneq X$. Then,*

$$\text{rdim}(X, P) \leq 1 + \max\{\text{rdim}(X \setminus Y, P(X \setminus Y)),\ \dim(Y, P(Y))\}$$

**Proof:** First, we compute a rectangular order $R$ that contains all ordered pairs in the relation $P \setminus (P(Y) \cup P(X \setminus Y))$. However, $R$ may interfere with $P$. But, $R$ is such that the ordered pairs in $R$ that do not belong to $P$ only involve the elements of $Y$. Such ordered pairs are reversed later. Now, to compute $R$, we claim that the elements of $Y$ can be "viewed" as a chain with respect to the elements of $X \setminus Y$. More precisely, it is possible to linearize the partial order $P(Y)$ to obtain a total order $T$ on $Y$ that satisfies the following property: for all elements $x, y \in Y$ if $x < y$ in $T$ then $D(x, X \setminus Y) \subseteq D(y, X \setminus Y)$ and $U(x, X \setminus Y) \supseteq U(y, X \setminus Y)$. Let $Q_D$ and $Q_U$ be the relations as defined:

$$Q_D = \{(x, y) \mid x, y \in Y,\ x \parallel y \text{ in } P(Y) \text{ and } D(x, X \setminus Y) \subsetneq D(y, X \setminus Y)\}$$

$$Q_U = \{(x, y) \mid x, y \in Y,\ x \parallel y \text{ in } P(Y) \text{ and } U(x, X \setminus Y) \supsetneq U(y, X \setminus Y)\}$$

We establish that the relation $P(Y) \cup Q_D$ is acyclic. The main idea is that any cycle in $P(Y) \cup Q_D$, if it exists, must involve a pair from $Q_D$ because $P(Y)$ is acyclic. Note that for every pair $(x, y) \in P(Y)$, $D(x, X \setminus Y) \subseteq D(y, X \setminus Y)$, and for every pair $(x, y) \in Q_D$, $D(x, X \setminus Y) \subsetneq D(y, X \setminus Y)$. Hence

Figure 15.5: The $Y$–indistinguishable classes of $X \setminus Y$ in $P$ where $(Y, P(Y))$ is a critical subposet of the poset $(X, P)$ (note that in the Hasse diagram, the line segments between elements are directed from left to right instead of the usual bottom to top).

if an element $p$ is involved in a cycle then $D(p, X \setminus Y) \subsetneq D(p, X \setminus Y)$—a contradiction. Similarly, it can be proved that the relation $P(Y) \cup Q_D \cup Q_U$ is acyclic as well. All incomparable pairs that remain after taking the transitive closure of $P(Y) \cup Q_D \cup Q_U$ are $(X \setminus Y)$–indistinguishable in $P$ and therefore can be ordered either way in $T$. The required total order $T$ on $Y$ is given by any linearization of the relation $P(Y) \cup Q_D \cup Q_U$.

We use $y_i$ to refer to the $i^{th}$ element in the total order $T$ for $i = 1, 2, \ldots, n$ where $n = |Y|$. We now compute the $Y$–indistinguishable classes of $X \setminus Y$ in $P$. Figure 15.5 shows various $Y$–indistinguishable classes of $X \setminus Y$ in $P$ and how they relate to $Y$. In the figure we depict each equivalence class by a single representative element, namely $a$ for $A$, and $d_i$, $u_i$ and $b_i$ for $D_i$, $U_i$ and $B_i$, respectively, for each $i$. The classes other than the $Y$–disconnected class $A$ can be partitioned into three categories. The first family of classes, denoted by $\mathcal{D}$, consists of classes $D_i$ for $i = 1, 2, \ldots, n$ so that an element $x \in X \setminus Y$ belongs to $D_i$ if $x < y_i$ in $P$ but $y_{i-1} \parallel x$ in $P$. Note that when $i = 1$ only the first condition is applicable, that is, $x \in D_1$ if $x < y_1$ in $P$. The second family of classes, denoted by $\mathcal{U}$, consists of classes $U_i$ for $i = 1, 2, \ldots, n$ so that an element $x \in X \setminus Y$ belongs to $U_i$ if $y_i < x$ in $P$ but $x \parallel y_{i+1}$ in $P$. Again, note that when $i = n$ only the first condition applies, that is, $x \in U_n$ if $y_n < x$ in $P$. The third family of classes, denoted by $\mathcal{B}$, contains classes $B_i$ for $i = 1, 2, \ldots, n-1$ such that an element $x \in X \setminus Y$ is contained in $B_i$ if $y_i < x < y_{i+1}$. Clearly, all $Y$–indistinguishable classes of $X \setminus Y$ in $P$ are covered by $\{A\} \cup \mathcal{D} \cup \mathcal{U} \cup \mathcal{B}$. The required rectangular order $R$ is given by:

$$\begin{aligned} R.1 &= a < d_n < d_{n-1} < \cdots < d_1 < y_1 < b_1 < y_2 < b_2 < \cdots < y_n < \\ &\quad u_n < u_{n-1} < \cdots < u_1 \\ R.2 &= d_1 < y_1 < u_1 < b_1 < d_2 < y_2 < u_2 < b_2 < \cdots < b_{n-1} < d_n < \\ &\quad y_n < u_n < a \end{aligned}$$

Now, we independently compute representations for the subposets $(Y, P(Y))$ and $(X \setminus Y, P(X \setminus Y))$. To represent the former, we use a chain realizer, and, to represent the latter, we use a rectangular realizer. Set $t = \max\{\dim(Y, P(Y)), \operatorname{rdim}(X \setminus Y, P(X \setminus Y))\}$. Let $\mathcal{C} = \{C_1, C_2, \ldots, C_t\}$ be a chain realizer of $P(Y)$ on $Y$, and let $\mathcal{S} = \{S_1, S_2, \ldots, S_t\}$ be a rectangular realizer of $P(X \setminus Y)$ on $X \setminus Y$. We construct a family of $t$ rectangular orders $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$ where the rectangular order $T_i$ on $X$ is given by disjoint composition of the order $C_i$ on $Y$ and the order $S_i$ on $X \setminus Y$ for $i = 1, 2, \ldots, t$. We choose a chain realizer and not a rectangular realizer for representing $(Y, P(Y))$ because when we linearize $P(Y)$ we may introduce ordered pairs from $Y \times Y$ that are not present in $P$. These pairs are reversed by the chain realizer. Finally, $\{R\} \cup \mathcal{T}$ constitutes a rectangular realizer of $P$ on $X$.

■

A chain trivially constitutes a critical subposet of any poset because it does not contain any incomparable pair. Also, the dimension of a chain is one. Thus, from critical subposet removal theorem, it follows that:

Figure 15.6: The $Y$-indistinguishable classes of $X \setminus Y$ in $P$ where $Y = \{x, y\} \subseteq X$ with $x \parallel y$ in $P$.

**Theorem 15.26 (chain removal theorem)** *Let $C$ be a chain in the poset $\mathbf{P} = (X, P)$. Then,*

$$\mathrm{rdim}(X, P) \leq 1 + \mathrm{rdim}(X \setminus C, P(X \setminus C))$$

Does a similar theorem exist for an antichain? The answer is in general no. But in case the antichain consists of only two elements, a removal theorem can indeed be provided.

**Theorem 15.27 (incomparable pair removal theorem)** *Let $\mathbf{P} = (X, P)$ be a poset and $(x, y)$ be an incomparable pair $P$. Then,*

$$\mathrm{rdim}(X, P) \leq 1 + \mathrm{rdim}(X \setminus \{x, y\}, P)$$

**Proof:** For convenience, set $Y = \{x, y\}$. We give a rectangular order $R$ containing the relation $P \setminus P(X \setminus Y)$ (in this case, $P(Y) = \emptyset$) that is non-interfering with $P$. Figure 15.6 depicts various $Y$-indistinguishable classes and how they relate to $Y$. As shown, there are seven such classes represented by $a$, $b$, $c$, $d$, $e$, $f$ and $g$. The required rectangular order $R$ is given by:

$$\begin{aligned} R.1 &= a < b < c < x < e < d < y < f < g \\ R.2 &= d < c < y < g < b < x < f < e < a \end{aligned}$$

It can be verified that $R$ contains $P \setminus P(X \setminus Y)$ and does not interfere with $P$. Let $\mathcal{S}$ be a rectangular realizer of $P(X \setminus Y)$ on $X \setminus Y$. We can obtain a rectangular realizer $\mathcal{T}$ of $P(X \setminus Y)$ on $X$ by disjointly composing each rectangular order of $\mathcal{S}$ with the empty order on $Y$. Then $\{R\} \cup \mathcal{T}$ constitutes a rectangular realizer of $P$ on $X$.

■

### Establishing Upper Bounds on Rectangular Dimension

Since $\mathrm{rdim}(X, P) \leq \dim(X, P)$ and $\dim(X, P) \leq |X|/2$ when $|X| \geq 4$, trivially, $\mathrm{rdim}(X, P) \leq |X|/2$ when $|X| \geq 4$. The bound is not tight for rectangular dimension as shown in this section. Before we give a bound on the rectangular dimension of a general poset, we provide a bound on the rectangular dimension of a bipartite poset. Recall that a poset $\mathbf{P} = (X, P)$ is *bipartite* if it does not contain any chain involving more than two elements. In that case, the ground set $X$ can be partitioned into two disjoint subsets $L$ and $U$ so that $P \subseteq L \times U$ and we write $\mathbf{P} = (X, P) = (L, U, P)$.

**Theorem 15.28** *Let $\mathbf{P} = (X, P) = (L, U, P)$ be a bipartite poset with nonempty $L$ and $U$. Then,*

$$\mathrm{rdim}((X, P)) \leq \min\{\lceil |L|/2 \rceil, \lceil |U|/2 \rceil\}$$

**Proof:** Without loss of generality, assume that $\lceil |L|/2 \rceil \leq \lceil |U|/2 \rceil$ and further that $|L|$ is even. In case $|L|$ is odd, we add an element to $L$ that is not connected to any other element. Set $t = \lceil |L|/2 \rceil$. Using incomparable pair removal theorem repeatedly, we successively remove two elements from $L$ which, by definition of $L$, are incomparable in $P$ until we have exhausted all elements in $L$. Clearly, $t$ pairs of elements are removed. Thus,

$$\text{rdim}(X, P) \leq t + \text{rdim}(U, P(U)) = t + \text{rdim}(U, \emptyset) = t + 1$$

Note, however, that we do not need a separate rectangle to represent the poset $(U, \emptyset)$. This is because the rectangular order constructed in the proof of the incomparable pair removal theorem is non-interfering with $P$. This implies that for all $x, y \in U$, $x \parallel y$ in each of the other $t$ rectangular orders. Therefore the fact that the elements of $U$ form an antichain is already captured in the other $t$ rectangular orders. As a result, $\text{rdim}(X, P) \leq t = \lceil |L|/2 \rceil$.

■

Clearly, either $|L| \leq |X|/2$ or $|U| \leq |X|/2$. Thus, from Theorem 15.28, it follows that:

**Corollary 15.29** *Let* $\mathbf{P} = (X, P)$ *be a bipartite poset. Then,*

$$\text{rdim}(X, P) \leq \lceil |X|/4 \rceil$$

For a general poset, a slightly weaker upper bound can be given.

**Theorem 15.30** *Let* $\mathbf{P} = (X, P)$ *be a poset with* $|X| \geq 3$. *Then,*

$$\text{rdim}(X, P) \leq |X|/3$$

**Proof:**[ for Theorem 15.30] The proof is by induction on the number of elements in $X$.

**Base Case ($|\mathbf{X}| \leq 5$):** It can be verified by doing case analysis that whenever $|X| \leq 5$, $\dim(X, P) \leq 2$ implying that $\text{rdim}(X, P) \leq 1$ [**?**, Page 23].

**Induction Step:** Suppose that $\text{rdim}(X, P) \leq |X|/3$ whenever $|X| \leq k$ where $k \geq 5$. Now consider a poset $(X, P)$ with $k+1$ elements. In case the poset $(X, P)$ contains a chain $C$ involving three elements, from Theorem 15.26, $\text{rdim}(X, P) \leq 1 + \text{rdim}(X \setminus C, P(X \setminus C)) \leq 1 + |X \setminus C|/3$ $= 1 + (|X| - 3)/3 = |X|/3$. Thus assume that the poset $(X, P)$ does not contain any chain involving more than two elements, or, in other words, it is a bipartite poset. From Corollary 15.29, $\text{rdim}(X, P) \leq \lceil |X|/4 \rceil \leq |X|/3$.

■

## 15.4 Order Decomposition Method and its Applications

### 15.4.1 The Main Idea

Given a poset $\mathbf{P} = (X, P)$, we first decompose the partial order $P$ into $t$ suborders $P_i$ for $i = 1, 2, \ldots, t$. It is not necessary for the suborders to be disjoint. We next compute a rectangular realizer $\mathcal{R}_i$ for each subposet $(P, X_i)$ such that $\mathcal{R}_i$ is non-interfering with $P$. Then the collection of rectangular orders $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \cdots \cup \mathcal{R}_t$ constitutes a rectangular realizer of $P$ on $X$. When we specify a decomposition of a partial order into suborders, we do not enumerate the reflexive pairs which can always be added later.

### 15.4.2 Applications

In this section, we provide upper bounds on rectangular dimension of posets based on two measures: "degree of connectivity" and "degree of adjacency".

#### Bounding Rectangular Dimension of Posets based on Degree of Connectivity

Suppose the Hasse diagram (or covering graph) of a poset $\mathbf{P} = (X, P)$ is such that every element in the graph has at most one outgoing edge. In this case the covering graph resembles a forest of trees. In particular, for every element $x \in X$, $U(x)$ forms a chain in $P$. Such a poset belongs to the class of *series-parallel* posets [**?**, **?**, **?**]. The dimension of a series-parallel poset is at most two and hence its rectangular dimension is at most one [**?**, **?**, **?**]. Similarly, a poset whose Hasse diagram is such that every element has at most one incoming edge also has rectangular dimension of one.

A natural question to ask is: what other posets have "small" rectangular dimension? In this section, we show that posets with "low" indegree have "small" rectangular dimension. Furthermore, we show that posets in which the indegree of every element is either "low" or "high" (but not "medium") also have "small" rectangular dimension.

For a poset $\mathbf{P} = (X, P)$ and an element $x \in X$, the *indegree* of $x$ in $P$, denoted $\deg_{\mathrm{D}}(x)$, is defined as the number of elements less than $x$ in $P$ [**?**, Page 165]. Let $\Delta_{\mathrm{D}}(X, P)$ denote $\max\{\deg_{\mathrm{D}}(x) \,|\, x \in X\}$. The *outdegree* of an element can be dually defined.

**Theorem 15.31** *Let $\mathbf{P} = (X, P)$ be a poset with $\Delta_{\mathrm{D}}(X, P) \leq k$ where $k \geq 1$. Then,*

$$\mathrm{rdim}(X, P) \leq k$$

**Proof:** The central idea is to decompose the partial order into at most $k$ suborders such that the subposet induced by each suborder is a rectangle.

For each element $x \in X$, number all elements in $D(x)$ from 1 to $|D(x)|$. The $i^{th}$ suborder $P_i$ for $i = 1, 2, \ldots, k$ is given by the reflexive transitive closure of the set $\{(x, y) | x \in X$ and $y$ is the $i^{th}$ element in $D(x)$, if it exists$\}$. Clearly, each element has at most one incoming edge in the Hasse diagram of $(X, P_i)$. Hence $(X, P_i)$ is a series-parallel poset, which implies that $P_i$ is a rectangular order. Since $P_i \subseteq P$, $P_i$ is non-interfering with $P$. Therefore it follows that the set $\{P_1, P_2, \ldots, P_k\}$ constitutes a rectangular realizer of $P$ on $X$.

■

We now show that if the indegree of every element in a poset is either at most $k$ or at least $|X|-k$, then the rectangular dimension of the poset is at most $\lceil 3k/2 \rceil + 1$.

**Theorem 15.32** *Let* $\mathbf{P} = (X, P)$ *be a poset such that for every element* $x \in X$, *either* $\deg_{\mathrm{D}}(x) \leq k$ *or* $\deg_{\mathrm{D}}(x) \geq |X| - k$ *where* $k \geq 1$. *Then,*

$$\mathrm{rdim}(X, P) \leq \lceil 3k/2 \rceil + 1$$

**Proof:** It suffices to prove that $\mathrm{rdim}(X, P) \leq \lceil 3k/2 \rceil + 1$ when $k \leq \lfloor |X|/2 \rfloor$. Our approach is to partition the ground set $X$ into two disjoint subsets $L$ and $U$ such that (1) for every element $x \in L$, $|D(x)| \leq k$, and (2) for every element $x \in U$, $|D(x)| \geq |X| - k$. Clearly, there is no element in $U$ that is less than some element in $L$; otherwise $D(x) \supset D(y)$ with $x \in L$, $y \in U$, and $y < x$ in $P$ implying that $|D(x)| > |D(y)| \geq |X| - k \geq \lceil |X|/2 \rceil$—a contradiction. We now bound the size of $U$ in case it is non-empty. Consider a minimal element $x$ of $(U, P(U))$. By definition, $D(x) \subseteq L$. Therefore $|L| \geq |D(x)| \geq |X| - k$ which in turn implies that $|U| \leq k$.

Set $t = \lceil 3k/2 \rceil + 1$. Let $T$ be some total order on $L \cup U$. We construct a rectangular order $R$ on $X$ as follows:

$$\begin{aligned} R.1 &= T(L) < T(U) \\ R.2 &= T^d(L) < T^d(U) \end{aligned}$$

Note that $R(L) = \emptyset$, $R(U) = \emptyset$ and every element of $L$ is less than every element of $U$ in $R$. Therefore some ordered pairs in $R$ need to be reversed. However, since every element in $U$ has high indegree, the number of such ordered pairs is small. Let $Q_1 = P(L) \cup P(U)$ and $Q_2 = \{(y, x) \mid x \in L,\ y \in U \text{ and } x \parallel y \text{ in } P\}$. Informally, $Q_1$ captures those ordered pairs in $P$ that do not belong to $R$, and $Q_2$ reverses those ordered pairs in $R$ that are not present in $P$.

To represent $(X, Q_1)$, we observe that the indegree of every element in $(X, Q_1)$ is at most $k$; for elements in $L$, it follows from the definition of $L$, and for elements in $U$, it follows from the fact that $|U| \leq k$. As a result, we can use the construction in the proof of Theorem 15.31 to compute a non-interfering rectangular realizer of $Q_1$ on $X$ consisting of at most $k$ rectangular orders, say $\mathcal{S}$.

To represent $(X, Q_2)$, we observe that $(X, Q_2)$ is actually a bipartite poset, say $(L', U', Q_2)$, where $L' = U$ and $U' = L$. Further, $|L'| = |U| \leq k$. As a result, we can use the construction in the proof of Theorem 15.28 to compute a non-interfering rectangular realizer of $Q_2$ on $X$ consisting of at most $\lceil k/2 \rceil$ rectangular orders, say $\mathcal{T}$.

Finally, $\{R\} \cup \mathcal{S} \cup \mathcal{T}$ constitutes a rectangular realizer of $P$ on $X$ consisting of at most $1 + k + \lceil k/2 \rceil = \lceil 3k/2 \rceil + 1$ rectangular orders.

■

### 15.4.3 Exercises

1. Prove theorems 2,4, 5, and 6.

# Chapter 16

# Online Problems

## 16.1 Introduction

We assume that the poset is presented to us one element at a time. Our job is to decompose the poset into antichains or chains.

If the elements are presented in arbitrary order, the following results hold.

**Theorem 16.1** *[?] There exists an online algorithm that partitions a poset of height $k$ into $\binom{k+1}{2}$ antichains.*

**Proof:** Whenever a new element $x$ arrives, we assign it a label $(l, u)$ where $l$ is the length of the longest chain below $x$ and $u$ is the length of the longest chain above $x$. As the poset has height $k$, we get that

$$0 \leq l + u \leq k - 1$$

Any element $y$ that is already in the poset and comparable to $x$ cannot have this label. Hence all elements with the same label are incomparable. There are at most $\binom{k+1}{2}$ number of labels.

■

The algorithm is optimal because of the following result.

**Theorem 16.2** *[?] There is an adversary that can force any online algorithm to use $\binom{k+1}{2}$ antichains to partition a poset of height $k$ and dimension 2.*

If the poset is presented online but in a linear order that is compatible with the poset, then we have

**Theorem 16.3** *There exists an online algorithm that partitions a poset of height $k$ into $k$ antichains if the poset is presented in a (topologically) sorted order.*

**Proof:** Whenever a new element $x$ arrives it is assigned the label $h(x)$. If $x < y$, it is clear that $h(x) < h(y)$. Therefore, $h(x) = h(y)$ implies that $x||y$. This is same as the logical clock algorithm.

■

Let us now turn our attention to decomposing the poset into chains. If the poset is presented in arbitrary order, then the best algorithm known requires exponential number of chains.

**Theorem 16.4** *[?] There exists an online algorithm that partitions a poset of width $k$ into $(5^k-1)/4$ chains.*

The lower bound is still quadratic.

**Theorem 16.5** *[?] There is an adversary that can force any online algorithm to use $\binom{k+1}{2}$ chains to partition a poset of width $k$ and dimension 2.*

We now focus on the online algorithm when the poset is presented in a sorted order. This implies that if $x$ is alread in the poset and $y$ is presented, then either $x < y$ or $x||y$.

We first show that any sorted poset of width two can be partitoned into three chains in an online manner.

**Theorem 16.6** *There exists an online algorithm that partitions a poset of width 2 into 3 chains.*

**Proof:** The main idea is to maintain the invariant that if there are two noneemty chains then at least two of the chains have their maximal elements incomparable. Initially, the algorithm keeps all incoming elements in one chain as long as possible. Since there is only one nonempty chain, the invariant is trivially true. As soon as the first time we get an element that is incomparable, we put it in another chain. Our invariant is still true. Now assume that the new element is $z$. Assume that the first two chains have their maximum elements $x$ and $y$ incomparable. The element $z$ cannot be incomparable to both of them otherwise we have an antichain of size 3. If $z$ is comparable to $x$ but not to $y$, we add $z$ after $x$ and our invariant stays true. Finally, if $z$ is comparable to both $x$ and $y$, we look at the third chain. If the third chain is empty, or the maximum element $w$ of the third chain is less than $z$, we can insert $z$ in the third chain and our invariant holds. If $w$ is incomparable to $z$, then we can insert $z$ after $x$ and now $z$ and $w$ form the incomparable pair.

■

We now give the result for width k.

**Theorem 16.7** *There exists an online algorithm that partitions a poset of width $k$ into $k(k+1)/2$ chains.*

**Proof:** We will maintain the invariant that a set of k chains have their max incomparable, a set of k-1 chains have their max incomparable and so on.

Let the new element be $z$. We find the smallest $B_i$ such that $z$ can be inserted into one of the queues in $B_i$. Note that an element can always be added to an empty queue. Since $B_k$ has all heads incomparable we are guaranteed to find one such queue; otherwise, we have an antichain of size $k+1$.

If $z$ can be inserted into $B_1$, we can insert into $B_1$ and all invariants are maintained. Since $i$ was the least index in which $z$ can be inserted in one of the queues, we can conclude that all chains in $B_j$ for $j < i$ are nonempty. Further $z$ is incomparable to all heads in $B_{i-1}$. We insert $z$ into the queue in $B_i$ and move that queue to the set $B_{i-1}$ which has now $i$ queues and $B_i$ has $i-1$ queues. By interchanging $B_i$ and $B_{i-1}$ our invariant continues to hold.

If $z$ is incomparable to all the heads in the first set, we have an antichain of size $k+1$. If it is comparable to $k-1$ of them, we can insert it in the remaining chain maintaining the invariant. If it is incomparable to $k-2$ of them

■

## 16.2 Lower Bound on Online Chain Decomposition

We define the chain decomposition problem for a computation poset with process information, $\mathcal{T}(k, N)$, as a game between players Bob and Alice in the following way:

Bob presents elements of an up-growing partial order of width $k$ to Alice. Information about the decomposition of the poset into $N$ chains is given to Alice in the form of a chain label assigned to every element that is presented. Alice needs to decompose the poset into as few chains as possible.

**Theorem 16.8** *For the problem $\mathcal{T}(k, N)$ with $k \leq \eta$, Bob can force Alice to use $\binom{k+1}{2}$ chains.*

**Proof:** Let the chains of the poset produced by decomposition be $C_i$ and $top(i)$ be the maximal element of the chain $C_i$. If $x$ is the maximal element of the poset, then $private(x)$ is the set of chains $C_i$ such that $top(i) \leq x$ and $top(i) \not\leq y$ for all maximal elements $y \neq x$. The process to which an element $e$ belongs is denoted by $p(e)$ and similarly for a set of elements $A$, $p(A)$ denotes the set of processes to which the elements in $A$ belong. In particular, when we say $p(private(x))$, it implies $\bigcup p(top(i))$ where $C_i \in private(x)$. Similarly $top(private(x))$ is the set $\bigcup top(i)$, $C_i \in private(x)$.

**Induction Hypothesis** : For every positive integer $k$ with $\binom{k+1}{2} \leq N$, there is a strategy $S(k)$ for Bob so that the poset $P$ presented so far is of width $k$, has exactly $k$ maximal elements and uses elements from only $\binom{k+1}{2}$ processes. Moreover, the maximal elements can be numbered $x_1, \ldots, x_k$ such that for all $i$, $|private(x_i)| \geq i$
and $|p(private(x_i)| = i$.

**Base Case** : For $k = 1$, we use one element from process 1. The hypothesis holds for this case.

**Induction Step** : Suppose we have strategy $S(k)$. We construct $S(k+1)$ using $S(k)$ as follows:

1. Run strategy $S(k)$. This phase ends with an order $Q_1$ with maximal elements $x_1, \ldots, x_k$, $|private(x_i)| \geq i$ and number of processes used $\binom{k+1}{2}$.

2. Run strategy $S(k)$ again. This time every new element is made greater than each of $x_1, \ldots, x_{k-1}$ and their predecessors but incomparable to rest of the elements in $Q_1$. In particular, the new elements are incomparable to elements in $top(i)$ for $C_i \in private(x_k)$. For constructing this new $S(k)$, we reuse the processes $p(Q_1) \setminus p(private(x_k))$ and add a set of $k$ new processes. The important observation here is that for $C_i \in private(x_k)$, there does not exist any element $f \in Q_1 \setminus top(private(x_k))$ such that $top(i) < f$. This effectively implies that process $p(top(i))$ cannot be used for the new $S(k)$.

   This phase ends with an order $Q_2$ with $k+1$ maximal elements $y_1, \ldots, y_k, x_k$. At this point, there are at least $i$ chains in $private(y_i)$ and an additional $k$ chains in $private(x_k)$. Similarly, at these point we have made use $\binom{k+2}{2} - 1$ processes.

3. Add a new element $z$ so that $z$ is greater than all elements of $Q_2$ and $p(z) = p(y_1)$. For the chain $C_i$ to which $z$ is assigned, it holds that $i \notin private(x_k)$ or $i \notin private(y_k)$. Wlog assume that $i \notin private(x_k)$. Now $private(z)$ has chains from $x_k$ and the chain to which $z$ belongs. So $|private(z)| \geq k+1$ and $|p(private(z))| = k+1|$. We refer to $z$ as $z_{k+1}$ from now on.

4. In this final phase, run strategy $S(k)$ again with all new elements greater than $y_1, \ldots, y_k$. For this phase, we use the processes
   $p(Q_2) \setminus (p(private(x_k)) \cup p(y_1))$ and add a new process to the system. This way we again have $\binom{k+1}{2}$ processes available for this phase without violating process semantics. This phase ends with maximal elements $z_1 \ldots, z_{k+1}$ so that $|private(z_i)| \geq i$ and
   $|p(private(z_i))| = i$.

During all the four phases we added $k+1$ new processes to the system and hence the total number of processes used till now is $\binom{k+1}{2} + (k+1) = \binom{k+2}{2}$.

■

# Chapter 17

# Fixed Point Theory

In this chapter we introduce the notion of lattices and obtain techniques for obtaining extremal solutions of inequations involving operations over lattices. Our motivation for studying these concepts and techniques stems from our interest in solving a system of inequations involving operations over the lattice of languages. Results presented in this chapter have a "primal" and a "dual" version. We only prove the primal version, as the dual version can be proved analogously.

**Remark 17.1** It is sometimes useful to consider a weaker requirement than that of a complete lattice. A poset $(X, \leq)$ is called a *complete partial order (cpo)* if

- $\inf X$ exists, and
- the least upper bound exists for any increasing chain of $X$.

Note that a complete lattice is always a cpo. The set of natural numbers with the natural ordering is an example of a lattice which is not a cpo.

■

## 17.1 Extremal Fixed Points

Given a set $X$ and a function $f : X \to X$, $x \in X$ is called a *fixed* point of $f(\cdot)$ if $f(x) = x$. Here we present sufficient conditions under which such a function possesses fixed points, and obtain techniques for computing some of them. Fixed point calculations are useful in *analysis* of DESs, as often it is required that the behavior of a DES satisfy a certain *invariance* property. Fixed point calculations can be used to compute the portion of the behavior which satisfies the required invariance property. Fixed point techniques are also used for *specifying* the behavior of a DES—instead of explicitly specifying the behavior, it is implicitly specified as a fixed point of a function defined over languages.

We begin by identifying several useful properties of functions defined over lattices. Given a poset $(X, \leq)$, a function $f : X \to X$ is said to be *idempotent* if

$$\forall x \in X : f(x) = f(f(x));$$

it is said to be *monotone* if

$$\forall x, y \in X : [x \leq y] \Rightarrow [f(x) \leq f(y)].$$

Given a complete lattice $(X, \leq)$, a function $f : X \to X$ is said to be *disjunctive* if

$$\forall Y \subseteq X : f(\sqcup_{y \in Y} Y) = \sqcup_{y \in Y} f(y);$$

it is said to be *conjunctive* if

$$\forall Y \subseteq X : f(\sqcap_{y \in Y} Y) = \sqcap_{y \in Y} f(y);$$

it is said to be *sup-continuous* if

$$\forall \text{ increasing chain } \{y_i, i \geq 0\} \subseteq X : f(\sqcup_{i \geq 0} y_i) = \sqcup_{i \geq 0} f(y_i);$$

it is said to be *inf-continuous* if

$$\forall \text{ decreasing chain } \{y_i, i \geq 0\} \subseteq X : f(\sqcap_{i \geq 0} y_i) = \sqcap_{i \geq 0} f(y_i).$$

It is readily verified that disjunctive, conjunctive, sup-continuous, and inf-continuous functions are also monotone. Note that since $\sup \emptyset = \inf X$ and $\inf \emptyset = \sup X$, by setting $Y = \emptyset$ in the last two definitions we obtain for a disjunctive function that $f(\inf X) = \inf X$, and for a conjunctive function that $f(\sup X) = \sup X$.

**Example 17.2** Consider the power set lattice of languages defined over the event set $\Sigma$, and the prefix and extension closure operations defined over it. Then it is easily verified that for a language $K \subseteq \Sigma^*$, $pr(K) = K/\Sigma^*$ and $ext(K) = K\Sigma^*$. Thus prefix closure operation is a special case of the quotient operation and extension closure operation is a special case of of concatenation operation. It can be checked that both concatenation and quotient operations are disjunctive and thus monotone; however, none of them are conjunctive. The prefix and extension closure operations are also both idempotent.

Now consider the operation $f(K) \stackrel{\text{def}}{=} K \cup L$, where $L$ is any fixed non-empty language. $f$ is conjunctive since $f(\cap_i K_i) = (\cap_i K_i) \cup L = \cap_i (K_i \cup L) = \cap_i f(K_i)$. It is not disjunctive because $f(\inf X) = f(\emptyset) = L$ which is different from $\inf X$.

The function defined as $f(K) \stackrel{\text{def}}{=} L - K$ is not even monotone.

■

The following fixed point theorem is due to Knaster and Tarski:

**Theorem 17.3** Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a monotone function. Let $Y \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$ be the set of fixed points of $f$. Then

1. $\inf Y \in Y$, and $\inf Y = \inf\{x \in X \mid f(x) \leq x\}$.
2. $\sup Y \in Y$, and $\sup Y = \sup\{x \in X \mid x \leq f(x)\}$.

**Proof:** For notational simplicity, define $Z \stackrel{\text{def}}{=} \{x \in X \mid f(x) \leq x\}$. Then it suffices to show that $\inf Y = \inf Z$ and $\inf Z \in Y$. Since $X$ is a complete lattice, $\sup X \in X$. By the definition of $\sup X$, $f(\sup X) \leq \sup X$, i.e., $\sup X \in Z$, which implies that $Z \neq \emptyset$. Also, since $X$ is a complete lattice, it follows that $\inf Z \in X$. By definition, we have $Y \subseteq Z$. Hence $\inf Z \leq \inf Y$. It remains to show that $\inf Y \leq \inf Z$. Since $\inf Y$ is the infimal fixed point of $f$, it suffices to show that $\inf Z$ is a fixed point of $f$, i.e., $f(\inf Z) = Z$.

We first show that $f(\inf Z) \leq \inf Z$. By definition, we have $\inf Z \leq z$ for each $z \in Z$. Monotonicity of $f$ implies that $f(\inf Z) \leq f(z)$ for each $z \in Z$. Since for each $z \in Z$, $f(z) \leq z$, this implies that $f(\inf Z) \leq z$ for each $z \in Z$. Hence $f(\inf Z) \leq \inf Z$. Next we show that $\inf Z \leq f(\inf Z)$. From above we have $f(\inf Z) \leq \inf Z$. Monotonicity of $f$ implies that $f(f(\inf Z)) \leq f(\inf Z)$. Hence it follows from the definition of $Z$ that $f(\inf Z) \in Z$, which implies that $\inf Z \leq f(\inf Z)$.

■

It follows from Theorem 17.3 that a monotone function defined over a complete lattice always has an infimal and a supremal fixed point. The following theorem provides a technique for computing such fixed points under stronger conditions. We first define the notion of disjunctive and conjunctive closure. Given a complete lattice $(X, \leq)$, and a function $f : X \to X$, the *disjunctive closure* of $f$, denoted $f^*$, is the map $f^* : X \to X$ defined as:

$$\forall x \in X : f^*(x) \stackrel{\text{def}}{=} \sqcup_{i \geq 0} f^i(x);$$

and the *conjunctive closure* of $f$, denoted $f_*$, is the map $f_* : X \to X$ defined as:

$$\forall x \in X : f_*(x) \stackrel{\text{def}}{=} \sqcap_{i \geq 0} f^i(x),$$

where $f^0$ is defined to be the identity function, and for each $i \geq 0$, $f^{i+1} \stackrel{\text{def}}{=} f f^i$. It is easy to see that the disjunctive as well conjunctive closures of $f$ are idempotent.

**Theorem 17.4** Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a function. Let $Y \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$ be the set of fixed points of $f$.

1. If $f$ is sup-continuous, then $\inf Y = f^*(\inf X)$.
2. If $f$ is inf-continuous, then $\sup Y = f_*(\sup X)$.

**Proof:** Since $f$ is sup-continuous, it is also monotone. Hence it follows from Theorem 17.3 that $\inf Y \in Y$. We first show that $\{f^i(\inf X), i \geq 0\}$ is an increasing chain. By definition we have

$$f^0(\inf X) = \inf X \leq f(\inf X) = f^1(\inf X).$$

Hence monotonicity of $f$ implies that for each $i \geq 0$, $f^i(\inf X) \leq f^{i+1}(\inf X)$.

Next we prove that $f^*(\inf X)$ is a fixed point. This follows from equalities:

$$\begin{aligned} f(f^*(\inf X)) &= f[\sqcup_{i \geq 0} f^i(\inf X)] \\ &= \sqcup_{i \geq 0} f^{i+1}(\inf X) \\ &= \sqcup_{i \geq 1} f^i(\inf X) \\ &= \sqcup_{i \geq 0} f^i(\inf X) \\ &= f^*(\inf X), \end{aligned}$$

where the second equality follows from sup-continuity of $f$, and the fourth equality follows from the fact that $f^0(\inf X) = \inf X \leq f^i(\inf X)$ for all $i \geq 1$.

It remains to show that $f^*(\inf X) \leq y$ for any $y \in Y$. Fix $y \in Y$. By definition of $\inf X$ we have $\inf X \leq y$. Since $f$ is sup-continuous, it is also monotone. Hence $\inf X \leq y$ implies

$$\forall i \geq 0 : f^i(\inf X) \leq f^i(y) = y,$$

where the last equality follows from the fact that $y$ is a fixed point. Thus we conclude that $f^*(\inf X) = \sqcup_{i \geq 0} f^i(\inf X) \leq y$, as desired.

■

**Remark 17.5** In the first part of Theorem 17.4, since $\{f^i(\inf X), i \geq 0\}$ is an increasing chain, $f^*(\inf X)$ exists under the weaker condition that $(X, \leq)$ is a cpo. Thus it is possible to compute the infimal fixed point of a sup-continuous function whenever it is defined over a cpo.

■

## 17.2 Dual, Co-Dual, Inverse, and Converse Operations

In this section we develop the notion of dual, co-dual, inverse, and converse operations and study some of their properties. These concepts are used in the next section for obtaining extremal solutions of a system of inequations. We begin by providing conditions for the existence of extremal solutions of simple inequations.

**Lemma 17.6** Consider a complete lattice $(X, \leq)$ and functions $f, g : X \to X$.

1. If $f$ is disjunctive, then the supremal solution of the inequation $f(x) \leq y$, in the variable $x$, exists for each $y \in X$.
2. If $g$ is conjunctive, then the infimal solution of the inequation $y \leq g(x)$, in the variable $x$, exists for each $y \in X$.

**Proof:** Since $(X, \leq)$ is complete, $\inf X \in X$, and by definition $\inf X \leq y$. Using the disjunctivity of $f$ we obtain $f(\inf X) = \inf X \leq y$. Thus the set of solutions of the inequation $f(x) \leq y$ is nonempty. Let $I$ be an indexing set such that for each $i \in I$, $x_i \in X$ is a solution of the inequation $f(x) \leq y$. Then it suffices to show that $\sqcup_{i \in I} x_i$ is also a solution of the inequation. Since $(X, \leq)$ is complete, it follows that $\sqcup_{i \in I} x_i \in X$. Also, $f(\sqcup_{i \in I} x_i) = \sqcup_{i \in I} f(x_i) \leq y$, where the equality follows from the fact that $f$ is disjunctive and the inequality follows from the fact that $f(x_i) \leq y$ for each $i \in I$.

■

Lemma 17.6 can be used to define the notion of dual of a disjunctive function and co-dual of a conjunctive function.

**Definition 17.7** Consider a complete lattice $(X, \leq)$ and functions $f, g : X \to X$. If $f$ is disjunctive, then its *dual*, denoted $f^{\perp}(\cdot)$, is defined to be the supremal solution of the inequation $f(x) \leq (\cdot)$. If $g$ is conjunctive, then its *co-dual*, denoted $g^{\top}(\cdot)$, is defined to be the infimal solution of the inequation $(\cdot) \leq g(x)$. ($x \in X$ is the variable of the inequation.)

**Example 17.8** Consider the power set lattice of languages defined over the event set $\Sigma$ and the prefix and extension closure operations. Since these operations are disjunctive, their duals exist. It follows from the definition of duality that for $K \subseteq \Sigma^*$, $pr^{\perp}(K)$ is the supremal language whose prefix closure is contained in $K$. Thus $pr^{\perp}(K)$ is the *supremal prefix closed sublanguage* of $K$, which we denote as $\sup P(K)$. Similarly, $ext^{\perp}(K)$ is the *supremal extension closed* sublanguage of $K$, which we denote as $\sup E(K)$.

As another example of duality, consider $f(K) \stackrel{\text{def}}{=} K \cap L$, where $L$ is any nonempty fixed language. Since intersection with a constant set is a disjunctive operation, its dual exists. It can be verified that $f^{\perp}(K) = K \cup L^c$.

■

The following proposition provides an alternative definition of duality as well as of co-duality.

**Theorem 17.9** Consider a complete lattice $(X, \leq)$, a disjunctive function $f : X \to X$, and a conjunctive function $g : X \to X$. Then the following are equivalent.

1. $f^{\perp} = g$.
2. $\forall x, y \in X : [f(x) \leq y] \Leftrightarrow [x \leq g(y)]$.
3. $g^{\top} = f$.

**Proof:** We only prove the equivalence of the first and the second assertion; the equivalence of the second and the third assertion can be proved analogously. Since $f$ is disjunctive, $f^{\perp}$ is defined. Suppose the first assertion is true. In order to see the forward implication of the second assertion, suppose $f(x) \leq y$, which implies $x$ is a solution of the inequation. Since $f^{\perp}(y) = g(y)$ is the supremal solution of the inequation, it follows that $x \leq g(y)$. Next in order to see the backward implication, suppose $x \leq g(y)$. So from monotonicity of $f$ we obtain that $f(x) \leq f(g(y))$. Since $g(y) = f^{\perp}(y)$ is a solution of the inequation, we have $f(g(y)) \leq y$. So $f(x) \leq f(g(y)) \leq y$, as desired.

Next suppose the second assertion holds. By setting $x = g(y)$ in the second assertion, we obtain that for all $y \in X$, $f(g(y)) \leq y$. This shows that $g(y)$ is solution of the inequation. Finally using the forward implication of the second assertion we conclude that if $x$ is a solution of the inequation, then $x \leq g(y)$. This shows that $g(y)$ is the supremal solution of the inequation. So $g(y) = f^{\perp}(y)$ for all $y \in X$.

■

Note that the equivalence of the first two assertions in Theorem 17.9 does not require $g$ to be conjunctive. Hence if we replace $g$ by $f^{\perp}$, then the first assertion is identically true; consequently, the second assertion is also identically true. Similarly it can be argued that the second assertion is identically true with $f$ replaced by $g^{\top}$. This is stated in the following corollary.

**Corollary 17.10** Consider a complete lattice $(X, \leq)$, and functions $f, g : X \to X$.

1. If $f$ is disjunctive, then $\forall x, y \in X : [f(x) \leq y] \Leftrightarrow [x \leq f^{\perp}(y)]$.
2. If $g$ is conjunctive, then $\forall x, y \in X : [g^{\top}(x) \leq y] \Leftrightarrow [x \leq g(y)]$.

Corollary 17.10 can be used to obtain several interesting properties of the dual and co-dual operations. We first show that dual of a disjunctive function is conjunctive, and co-dual of a conjunctive function is disjunctive.

**Lemma 17.11** Consider a complete lattice $(X, \leq)$ and functions $f, g : X \to X$.

1. If $f$ is disjunctive, then $f^{\perp}$ is conjunctive.
2. If $g$ is conjunctive, then $g^{\top}$ is disjunctive.

**Proof:** Pick $Y \subseteq X$. We need to show that $f^{\perp}(\sqcap_{y \in Y} y) = \sqcap_{y \in Y} f^{\perp}(y)$. The forward inequality can be shown as follows:

$$\begin{aligned}
[f^{\perp}(\sqcap_{y \in Y} y) \leq f^{\perp}(\sqcap_{y \in Y} y)] &\Leftrightarrow [f(f^{\perp}(\sqcap_{y \in Y} y)) \leq \sqcap_{y \in Y} y] \\
&\Leftrightarrow [\forall y \in Y : f(f^{\perp}(\sqcap_{y \in Y} y)) \leq y] \\
&\Leftrightarrow [\forall y \in Y : f^{\perp}(\sqcap_{y \in Y} y) \leq f^{\perp}(y)] \\
&\Leftrightarrow [f^{\perp}(\sqcap_{y \in Y} y) \leq \sqcap_{y \in Y} f^{\perp}(y)],
\end{aligned}$$

where the first and the third equivalence follow from Corollary 17.10.

Next the reverse inequality can be obtained as follows:

$$\begin{aligned}
[\forall y \in Y : \sqcap_{y \in Y} f^{\perp}(y) \leq f^{\perp}(y)] &\Leftrightarrow [\forall y \in Y : f(\sqcap_{y \in Y} f^{\perp}(y)) \leq y] \\
&\Leftrightarrow [f(\sqcap_{y \in Y} f^{\perp}(y) \leq \sqcap_{y \in Y} y] \\
&\Leftrightarrow [\sqcap_{y \in Y} f^{\perp}(y) \leq f^{\perp}(\sqcap_{y \in Y} y)],
\end{aligned}$$

where the first and the final equivalence follow from Corollary 17.10.

■

It follows from Lemma 17.11 that it is possible to define co-dual of the dual of a disjunctive function and dual of the co-dual of a conjunctive function. The following proposition describes some other properties of dual and co-dual operations.

**Theorem 17.12** Consider a complete lattice $(X, \leq)$, functions $f, f_1, f_2 : X \to X$ that are disjunctive, and functions $g, g_1, g_2 : X \to X$ that are conjunctive.

1. ($^\perp$, $^\top$ inverses) $(f^\perp)^\top = f$ $\quad$ $(g^\top)^\perp = g$
2. (composition) $(f_1 f_2)^\perp = f_2^\perp (f_1^\perp)$ $\quad$ $(g_1 g_2)^\top = g_2^\top (g_1^\top)$
3. (idempotence) $[ff = f] \Leftrightarrow [f^\perp f^\perp = f^\perp]$ $\quad$ $[gg = g] \Leftrightarrow [g^\top g^\top = g^\top]$

**Proof:** 1. Since $f$ is disjunctive, it follows from Lemma 17.11 that $f^\perp$ is conjunctive, so $(f^\perp)^\top$ is defined. By replacing $g$ with $f^\perp$ in Theorem 17.9 we obtain from its third assertion that $(f^\perp)^\top = f$, as desired.

2. Since disjunctivity is preserved under composition of functions, $f_1 f_2$ is disjunctive, so that its dual is defined. Fix $x, y \in X$. Then the repeated application of Corollary 17.10 yields the following series of equivalences:

$$\begin{aligned} [f_1 f_2(x) \leq y] &\Leftrightarrow [f_2(x) \leq f_1^\perp(y)] \\ &\Leftrightarrow [x \leq f_2^\perp f_1^\perp(y)]. \end{aligned}$$

Since $f_2^\perp f_1^\perp$ is conjunctive (follows from Lemma 17.11, and the fact that conjunctivity is preserved under composition of functions), if we replace $f$ by $f_1 f_2$ and $g$ by $f_2^\perp f_1^\perp$ in Theorem 17.9, we obtain $(f_1 f_2)^\perp = f_2^\perp f_1^\perp$, as desired.

3. The forward implication can be shown as follows: $f^\perp = (ff)^\perp = f^\perp f^\perp$, where the first equality follows from the hypothesis, and the second from part 2. The backward implication can be obtained as follows: $f = (f^\perp)^\top = (f^\perp f^\perp)^\top = ff$, where the first equality follows from part 1, the second from the hypothesis, and the final from parts 2 and 1.

■

**Example 17.13** Consider the power set lattice of languages defined over the event set $\Sigma$. We showed in Example 17.8 that $pr^\perp = \sup P$ and $ext^\perp = \sup E$. Then it follows from Lemma 17.11 that $\sup P$ as well as $\sup E$ are conjunctive. Moreover, Theorem 17.12 implies that $(\sup P)^\top = (pr^\perp)^\top = pr$ and $(\sup E)^\top = (ext^\top)^\perp = ext$. Finally, since $pr$ and $ext$ are idempotent, it follows from Theorem 17.12 that $pr^\perp = \sup P$ and $ext^\top = \sup E$ are idempotent.

■

## 17.3 Extremal Solutions of Inequations

Given a complete lattice $(X, \leq)$ and a finite family of functions $\{f_i, g_i : X \to X\}_{i \leq n}$, where $n \in \mathcal{N}$, we next consider computation of extremal solutions of the system of inequations:

$$[\forall i \leq n : f_i(x) \leq g_i(x)],$$

where $x \in X$ is the variable of the system of inequations. Note that this also allows us to obtain extremal solutions of a system of *equations*, as each equation can equivalently be written as a pair of *inequations*. We show that the computation of extremal solutions of the above system of inequations can be reduced to extremal fixed point computations of certain induced functions. We need the result of the following lemma:

**Lemma 17.14** Consider the system of inequations $\{f_i(x) \leq g_i(x)\}_{i \leq n}$ over a complete lattice $(X, \leq)$. Define functions $h_1, h_2 : X \to X$ as:

$$\forall y \in X : h_1(y) \stackrel{\text{def}}{=} \sqcap_{i \leq n} f_i^{\perp}(g_i(y)); \quad \forall y \in X : h_2 \stackrel{\text{def}}{=} \sqcup_{i \leq n} g_i^{\top}(f_i(y)). \tag{17.1}$$

1. If $f_i$ is disjunctive and $g_i$ is monotone for each $i \leq n$, then $h_1$ is monotone, and $\forall y, z \in X : [y \leq h_1(z)] \Leftrightarrow [\forall i \leq n : f_i(y) \leq g_i(z)]$.
2. If $f_i$ is monotone and $g_i$ is conjunctive for each $i \leq n$, then $h_2$ is monotone, and $\forall y, z \in X : [h_2(y) \leq z] \Leftrightarrow [\forall i \leq n : f_i(y) \leq g_i(z)]$.

**Proof:** In order to show the monotonicity of $h_1$, it suffices to show that for each $i \leq n$, $f_i^{\perp} g_i$ is monotone. This follows from the facts that $g_i$ is given to be monotone, $f_i^{\perp}$ is conjunctive (refer to Lemma 17.11), so that it is also monotone, and monotonicity is preserved under composition of functions.

In order to see the second claim, fix $y, z \in X$. Then we have the following series of equivalences:

$$\begin{aligned} [y \leq h_1(z)] &\Leftrightarrow [y \leq \sqcap_{i \leq n} f_i^{\perp}(g_i(z))] \\ &\Leftrightarrow [\forall i \leq n : y \leq f_i^{\perp}(g_i(z))] \\ &\Leftrightarrow [\forall i \leq n : f_i(y) \leq g_i(z)], \end{aligned}$$

where the final equivalence follows from the first part of Corollary 17.10.

■

**Theorem 17.15** Consider the system of inequations $\{f_i(x) \leq g_i(x)\}_{i \leq n}$ over a complete lattice $(X, \leq)$. Let

$$Y \stackrel{\text{def}}{=} \{y \in X \mid \forall i \leq n : f_i(y) \leq g_i(y)\} \tag{17.2}$$

be the set of all solutions of the system of inequations; and

$$Y_1 \stackrel{\text{def}}{=} \{y \in X \mid h_1(y) = y\}, \quad Y_2 \stackrel{\text{def}}{=} \{y \in X \mid h_2(y) = y\} \tag{17.3}$$

be the sets of all fixed points of $h_1$ and $h_2$, respectively, where $h_1$ and $h_2$ are defined by (17.1).

1. If $f_i$ is disjunctive and $g_i$ is monotone, then $\sup Y \in Y$, $\sup Y_1 \in Y_1$ and $\sup Y = \sup Y_1$.
2. If $f_i$ is monotone and $g_i$ is conjunctive, then $\inf Y \in Y$, $\inf Y_2 \in Y_2$ and $\inf Y = \inf Y_2$.

**Proof:** It follows from the first part of Lemma 17.14 that $h_1$ is monotone. Hence it follows from the second part of Theorem 17.3 that $\sup Y_1 \in Y_1$, and

$$\sup Y_1 = \sup Y_1', \text{ where } Y_1' \stackrel{\text{def}}{=} \{y \in X \mid y \leq h_1(y)\}. \tag{17.4}$$

It remains to show that $\sup Y \in Y$ and $\sup Y = \sup Y_1$. In view of (17.4), it suffices to show that $Y = Y_1'$, i.e, $y \in X$ is a solution of the system of inequations if and only if $y \leq h_1(y)$. This follows from the first part of Lemma 17.14 by setting $z = y$.

■

Theorem 17.15 provides a condition for the existence of extremal solutions of a system of inequations. The following theorem provides techniques for the computation of such extremal solutions.

**Theorem 17.16** Consider the system of inequations $\{f_i(x) \leq g_i(x)\}_{i \leq n}$ over a complete lattice $(X, \leq)$; and the set $Y$ of all solutions of the system of inequations as defined by (17.2).

1. Let $f_i$ be disjunctive and $g_i$ be monotone. Consider the following iterative computation:

   - $y_0 \stackrel{\text{def}}{=} \sup X$,
   - $\forall k \geq 0 : y_{k+1} \stackrel{\text{def}}{=} h_1(y_k)$,

   where $h_1$ is defined by (17.1). Suppose $m \in \mathcal{N}$ is such that $y_{m+1} = y_m$; then $y_m = \sup Y$.

2. Let $f_i$ be monotone and $g_i$ be conjunctive. Consider the following iterative computation:

   - $y_0 \stackrel{\text{def}}{=} \inf X$,
   - $\forall k \geq 0 : y_{k+1} \stackrel{\text{def}}{=} h_2(y_k)$,

   where $h_2$ is defined by (17.1). Suppose $m \in \mathcal{N}$ is such that $y_{m+1} = y_m$; then $y_m = \inf Y$.

**Proof:** It follows from Theorem 17.15 that the supremal solution of the system of inequations, $\sup Y$, exists. We first show that $y_m \in Y$, i.e., it is a solution of the system of inequations. Since $y_{m+1} = y_m$, we have $h_1(y_m) = y_m$, which implies that $y_m \leq h_1(y_m)$. Thus by setting $y = z = y_m$ in the equivalence of the first part of Lemma 17.14, we obtain that $y_m \in Y$.

Next we show that if $z \in Y$ is another solution of the system of inequations, then $z \leq y_m$. We use induction to show that for each $k \geq 0$, $z \leq y_k$. If $k = 0$, then $y_k = \sup X$, so that $z \leq y_k = \sup X$. Thus the base step trivially holds. Suppose for the induction hypothesis that $z \leq y_k$ for some $k \geq 0$. From the first part of Lemma 17.14 we have that $h_1$ is monotone. This together with the induction hypothesis implies that $h_1(z) \leq h_1(y_k) = y_{k+1}$. Thus it suffices to show that $z \leq h_1(z)$. Since $z \in Y$, $f_i(z) \leq g_i(z)$ for each $i \leq n$. Thus by setting $y = z$ in the first part of Lemma 17.14, we obtain that $z \leq h_1(z)$.

■

**Example 17.17** Suppose that we are interested in the supremal solution of the inequation $X \cap A \subseteq X \cap B$ defined over the power set lattice of languages. Then, the iteration of Theorem 17.16 yields the following:

$$
\begin{aligned}
y_0 &= \Sigma^* \\
y_1 &= (\Sigma^* \cap B) \cup A^c = B \cup A^c \\
y_2 &= ((B \cup A^c) \cap B) \cup A^c = B \cup A^c.
\end{aligned}
$$

Since $y_2 = y_1$, it follows from Theorem 17.16 that $y_1$ is the supremal solution of the inequation.

■

## 17.4 Exercises

1. Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a function. Prove that disjunctivity, conjunctivity, sup-continuity, and inf-continuity of $f$ each imply monotonicity of $f$.

2. Consider the complete lattice $(X, \leq)$, where

$$X = \{a, b, c, d\}; \quad \leq \; = \{(a, b), (a, c), (b, d), (c, d), (a, d)\}.$$

   Let $f : X \to X$ be defined as: $f(a) = f(b) = f(c) = d$ and $f(d) = d$.

   (a) Draw the Hasse diagram for the poset $(X, \leq)$.

   (b) Determine whether $f$ is disjunctive or conjunctive.

3. Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a function. $f$ is said to be *increasing* if $x \leq f(x), \forall x \in X$; it is said to be *decreasing* if $f(x) \leq x, \forall x \in X$.

   (a) Prove that $f^*$ is increasing and idempotent; and $f_*$ is decreasing, idempotent.

   (b) If $f$ is monotone (respectively, disjunctive), then show that $f^*$ is monotone (respectively, disjunctive).

   (c) If $f$ is monotone (respectively, conjunctive), then show that $f_*$ is monotone (respectively, conjunctive).

4. Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a monotone function. Show that the set of fixed points of $f$ is also a complete lattice.

5. Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be a monotone function. Prove that $\sup\{x \in X \mid f(x) = x\} = \sup\{x \in X \mid x \leq f(x)\}$.

6. Let $(X, \leq)$ be a complete lattice and $f : X \to X$ be an inf-continuous function. Prove that $f_*(\sup X)$ is the supremal fixed point of $f$.

7. Consider the power set lattice $(2^{\Sigma^*}, \subseteq)$ of all languages defined over the event set $\Sigma$. Let $K_1, K_2 \subseteq \Sigma^*$ be two fixed languages. Define a function $f : 2^{\Sigma^*} \to 2^{\Sigma^*}$ as follows:

$$\forall H \subseteq \Sigma^* : f(H) \stackrel{\text{def}}{=} K_1 H + K_2.$$

   Show that $f$ commutes with the supremal operation taken over a countable set. Using Theorem 17.4 compute the infimal fixed point of $f$.

8. Prove the second part of Lemma 17.6.

9. Prove the equivalence of the second and third assertions in Theorem 17.9.

10. Prove the second part of Lemma 17.11.

11. Prove Lemma **??**.

12. Prove the second parts of Lemma 17.14 and Theorem 17.15.

13. Prove the second parts of Theorems 17.16 and **??**.

14. Prove the second part of Theorem **??**.

15. Consider the power set lattice of a set $X$, and a function $f : 2^X \to 2^X$ such that it is increasing, monotone, and idempotent. Consider the inequation $f(Y) \subseteq Y$, where $Y \subseteq X$ is the variable of the inequation.

    (a) Give an example of $X$ and $f$ to illustrate that a supremal solution of the inequation need not exist.

    (b) Prove that the infimal solution of the inequation exists.

    (c) Obtain a formula for the infimal solution larger than a given set $Z \subseteq X$.

## 17.5 Bibliographic Remarks

One of the early results on existence of fixed points of a monotone function is due to Knaster-Tarski [**?**]. Lassez-Nguyen-Sonenberg [**?**] provide a nice historical account of this and other fixed point theorems. Our treatment of inequations is influenced by Dijkstra-Scholten [**?**]; in particular, the term *converse* of a function have been borrowed from there. The existence and computation of extremal solutions of a system of inequations over lattices is reported in Kumar-Garg [**?**].

# Chapter 18

# Decomposed Posets

## 18.1 Introduction

Let $(S, <)$ be any partially ordered finite set. We are given a decomposition of $S$ into $n$ sets $P_0, ...P_{n-1}$ such that for any $i$, $P_i$ is a chain of size at most $m$. We call such a structure a *decomposed* poset. These structures arise in distributed systems, because any execution of a distributed program can be viewed as a decomposed poset with local states of the processes as elements of the poset, and ordering between these states as that imposed by *happened-before* relation defined by Lamport[1].

Our first problem is to find an element $s_i$ in each $P_i$ such that $< \forall i, j : i \neq j : s_i \not< s_j >$. In other words, we have to find an anti-chain of size $n$ if one exists. Clearly, this anti-chain is the largest as there cannot be any anti-chain of size greater than $n$. Our second problem is to find any two elements $s_i$ and $s_j$ such that $(s_i \not< s_j) \wedge (s_j \not< s_i)$. If no two such elements exist, then the given poset is actually a totally ordered set.

These problems have many applications in distributed processing. In distributed debugging[2,5], one may need to detect a global snapshot so that the local condition $C_i$ is true in process $i$. Each process may then detect $C_i$ locally and send the states in which it became true to a coordinator process. The task of the coordinator is to find one state in each process $i$ in which $C_i$ is true and states are causally unrelated to each other. Intuitively speaking, these states can be considered to have occurred simultaneously. We call such a set of states a consistent cut. For example, consider an implementation of the dining philosophers problem. Let $holding_i$ denote the condition that $philosopher_i$ is holding a fork. We may want to detect the condition $< \forall i : holding_i >$ if it ever becomes true for any consistent cut. Assume that each $philosopher_i$ sends the state in which $holding_i$ became true to a coordinator process. The task of the coordinator is to find causally unrelated sets of states in which $holding_i$ is true. If it succeeds then we can claim that for some execution speeds of the philosophers the algorithm may lead to a deadlock.

As another example, consider the problem of recovery in a distributed system[3]. While a distributed computation proceeds, it is desirable to keep a global consistent cut so that in case of any fault the computation may start from the consistent cut instead of the beginning. Assuming that local

checkpointing and messages-logging is done asynchronously, each process outputs states from which it can restart its local computation. The task again is to find a consistent cut.

The second problem may be useful in debugging a mutual-exclusion algorithm. Each process may record its state when it accessed the critical region. If there is any pair of concurrent states in which critical region is accessed, i.e., $(s_i \not< s_j) \wedge (s_j \not< s_i)$, then mutual exclusion can be violated with appropriate processor speeds.

We describe an algorithm that requires $O(n^2m)$ comparisons for finding the largest anti-chain and an algorithm that requires $O(mn \log n)$ comparisons to find any anti-chain of size two.

## 2. Consistent Cut Problem

We use $s||t$ to denote that $s$ and $t$ are concurrent, i.e., $(s_i \not< s_j) \wedge (s_j \not> s_i)$. The consistent cut problem is: Given $(S, <)$ and its decomposition $(P_0, P_1, ..., P_{n-1})$, does there exists $\{s_i \in P_i\}$ such that $(\forall i, j : i \neq j : s_i||s_j)$ ? We first present an off-line algorithm which assumes that the entire poset is available at the beginning. Later, we adapt this algorithm for on-line detection of consistent cut in a distributed environment.

### 2.1 Algorithm

We will assume that any two elements of the set can be compared in O(1) time. This is true in distributed processing if states are time-stamped with vector clocks[4]. Our algorithm uses one queue $q_i$ per chain to store elements of $p_i$. We assume that elements within a queue are sorted in increasing order. In the following discussion, we use the following operations on queues:

insert(q,elem); insert elem in the queue q
deletehead(q); remove the head of the queue
empty(q); true if q is empty
head(q); first element if ¬ empty(q), returns *max* (the biggest element) otherwise

The algorithm is given below:

**function** antichain($q_0, ..., q_{n-1}$:queues):boolean;
*(* returns true if there is an anti-chain of size n, false otherwise. If true, the anti-chain is given by the head of all the queues *)*
**const** all = {0,1,...,n-1};
**var** low,newlow: subsets of all;
i,j: 0..n-1;
**begin**
  low := all;
  **while** low $\neq \phi$ **do**
  **begin**
    newlow := {};
    **for** i **in** low **do**
      **for** j **in** all **do**
        **if** $head(q_i) < head(q_j)$ **then** newlow:=newlow ∪ {i};
        **if** $head(q_j) < head(q_i)$ **then** newlow:=newlow ∪ {j};
    low := newlow;
    **for** i **in** low **do**
      deletehead($q_i$)
  **end**

return($\forall i : \neg$ empty($q_i$))
**end**

The algorithm works as follows. It compares only the heads of queues. Moreover, it compares only those heads of the queues which have not been compared earlier. For this purpose, it uses the variable *low* which is the set of indices for which the head of the queues have been updated. The invariant maintained by the *while* is
$(I) \qquad (\forall i, j \notin low : \neg empty(q_i) \wedge \neg empty(q_j) \Rightarrow head(q_i) || head(q_j))$
$I$ is true initially because *low* contains the entire set. The while loop maintains the invariant by finding all those elements which are lower than some other elements and including them in *low*. This means that there can not be two comparable elements in $all - low$. The loop terminates when *low* is empty. At that point, if all queues are non-empty, then by the invariant $I$, we can deduce that all the heads are concurrent. The procedure clearly terminates because the number of elements in the queues decreases on every execution of the while loop unless *low* is $\phi$ in which case the procedure terminates.

The above procedure can be made to terminate earlier when any of the queue becomes empty. For that purpose, it is enough to introduce a variable *noempty* which is made false whenever the last element in any queue is deleted. This flag is also a part of the while condition. We have not done so for clarity of the discussion.

We now discuss the complexity of above algorithm. Our complexity analysis will be done based on the number of comparisons used by the algorithm. The following theorem shows that the number of comparisons are quadratic in $n$ and linear in $m$.

**Theorem 18.1** *The above algorithm requires at most $O(n^2m)$ comparisons.*

*Proof:* Let $comp(k)$ denote the number of comparisons required in the $k^{th}$ iteration of the while loop. Let $t$ denote the total number of iterations of the while loop. Then, total number of comparisons $= \sum_{k=1}^{k=t} comp(k)$. Let $low(k)$ represent the value of *low* at the $k^{th}$ iteration. It is *all* in the first iteration. We note that $|low(k)|$ for $k \geq 2$ represents the number of elements deleted in the $k-1$ iteration of the while loop. Therefore, $\sum_{k=2}^{t} |low(k)|$ = total elements deleted $<$ total elements in system $\leq mn$.

From the structure of the for-loops we get that $comp(k) = O(n * |low(k)|)$. Therefore, the total number of comparisons required are
$\sum_{k=1}^{t} comp(k) = n * \sum_{k=1}^{t} |low(k)|$
$= n * n + n * \sum_{k=2}^{t} |low(k)|$
$\leq n * n + n * mn = O(n^2m)$

■

The above algorithm assumes that the entire queues were available as the input at the beginning. For many applications, the queues may be available only one element at a time. The problem of on-line computation of consistent-cut is to detect the consistent cut as soon as all elements forming the cut are available. This cut corresponds to the infimum element of the lattice of all consistent cuts in $S$[4]. The following algorithm assumes that a centralized process receives all the elements

in the set S one at a time. This algorithm is a minor variant of the previous algorithm. The main difference is that an empty queue signifies that no consistent cut has been found so far. The centralized process may receive more elements in future and succeed in finding one. It computes only on receiving a message from some process. It maintains the assertion that heads of all non-empty queues are incomparable. The computation is shown below:

```
Upon recv(elem) from P_i do
begin
    if ¬ empty(q_i) then insert(q_i, elem)
    else begin
        insert(q_i, elem)
        low := { i }
        while low ≠ φ do
        begin
            newlow := {};
            for k in low do
                for j in all do
                    if head(q_k) < head(q_j) then newlow:=newlow ∪ {k};
                    if head(q_j) < head(q_k) then newlow:=newlow ∪ {j};
            low := newlow;
            for k in low do
                deletehead(q_k)
        end;(* while *)
    if ∀k : ¬empty(q_k) then found:=true
end
```

### 2.2 Adversary Arguments

In this section we show that the complexity of the above problem is at least $\Omega(n^2m)$, thus showing that our algorithm is optimal. We first show an intermediate lemma which handles the case when the size of each queue is exactly one, i.e. $m = 1$.

**Lemma 18.2** *Let there be $n$ elements in a set $S$. Any algorithm which determines if all elements are incomparable must make at least $n(n-1)/2$ comparisons.*

*Proof:* The adversary will give to the algorithm a set in which either zero or exactly one pair of elements are incomparable. The adversary also chooses to answer "incomparable" to first $n(n-1)/2 - 1$ questions. Thus, the algorithm cannot determine if the set has a comparable pair unless it asks about all the pairs.

■

We use the above Lemma to show the desired result.

**Theorem 18.3** *Let $(S, <)$ be any partially ordered finite set of size $mn$. We are given a decomposition of $S$ into $n$ sets $P_0, ...P_{n-1}$ such that $P_i$ is a chain of size $m$. Any algorithm which determines if there exists an anti-chain of size $n$ must make at least $mn(n-1)/2$ comparisons.*

*Proof:* Let $P_i[k]$ denote the $k^{th}$ element in $P_i^{th}$ chain. The adversary will give the algorithm $S$ and $P_i$'s with the following characteristic:
$\forall i, j, k : P_i[k] < P_j[k+1]$
Thus, the above problem reduces to $m$ instances of the problem which checks if any of the $n$ elements is incomparable. The algorithm for the adversary can be stated as follows:
**var** num[k]:integer initially 0;
{ *number of questions asked about level k*}
On being asked to compare $P_i[k]$ and $P_j[l]$
**if** $(k < l)$ **then** return $P_i[k] < P_j[l]$
**if** $(l < k)$ **then** return $P_j[l] < P_i[k]$
**if** $(l = k)$ **then begin**
num[k]++;
**if** (num[k] = n*(n-1)/2) **then** return $P_i[k] < P_j[l]$
**else** return $P_j[l]||P_i[k]$
**end**
If the algorithm does not completely solve one instance then the adversary chooses that instance to show a poset consistent with all its answers but different in the final outcome.

■

## 3. Total Order Problem

Let $(S, <)$ be any decomposed partially ordered set. Our problem is to find if there exists elements $s_i$ and $s_j$ such that $s_i||s_j$. In other words, we have to find if the given decomposed poset is a total order.

### 3.1. An Algorithm

We first provide a sequential algorithm for the above problem. It can be seen that if there is a total order then the above poset can be sorted. Thus any sorting algorithm will answer the query in $O(mn \log mn)$ time. However, we have not exploited the fact that all events within a chain are comparable. To do so, we employ the following algorithm:

$L = \{P_0, P_1, P_2, ..., P_{n-1}\}$
**while** $|L| \neq 1$ **do**
**begin**
$X_i := removemin(L)$;
$X_j := removemin(L)$;
$Y := merge(X_i, X_j)$;
$insert(L, Y)$
**end**

$L$ in the above algorithm is a set of all chains known so far. It may be a heap in an actual implementation. *removemin* removes the chain from $L$ with the smallest size. *merge* inputs two chains and outputs a merged chain if all elements are comparable. The complexity of merging two lists $X_i$ and $X_j$ is $O(|X_i| + |X_j|)$. Each chain is involved in at most $O(\log n)$ merges. In each merge it makes a contribution of $m$. Since there are $n$ chains, the total number of comparisons is $O(mn \log n)$.

The above algorithm assume that all elements of the poset are available as input. An on-line computation can be done as follows. All the elements seen so far are kept in a sorted order. On receiving any new element, its position in the sorted list can be determined in at most $O(\log mn)$ comparisons. Thus, $O(mn \log mn)$ comparisons would be required to detect the first anti-chain. This algorithm is similar to insertion sort used for sorting an array of integers. Note that if $n < m$, then $O(mn \log mn) = O(mn(\log m + \log n)) = O(mn \log m)$.

### 3.2. An Adversary Argument

The lower bound for the problem is $\Omega(mn \log n)$. We use our previous technique of dividing the poset in $m$ levels. Given two elements $s$ and $t$ at levels $i$ and $j$ with $i < j$, the adversary always returns $s < t$. Thus, the task of the algorithm is reduced to finding if each level is a total order. For each level, we show that at least $\Omega(n \log n)$ comparisons are required. We just need to show that for any poset of size $n$, at least $\Omega(n \log n)$ comparisons are required to determine whether it is a total order. To prove this we claim that the algorithm must ask enough questions so that it can determine the precise order just to answer if the set is totally ordered or not. In other words, the algorithm must ask enough questions to be able to "sort" the poset. If not, there is a pair of elements $e$ and $f$ for which all the answers are consistent for both $e < f$ and $f < e$. In this case, the adversary can always produce a poset that is inconsistent with the answer given by the algorithm. As the number of comparisons required to sort is $\Omega(n \log n)$, so is the complexity for determining if any level is a total order.

# Bibliography


[Bir40] G. Birkhoff. *Lattice Theory*. Providence, R.I., 1940. first edition.

[Bir48] G. Birkhoff. *Lattice Theory*. Providence, R.I., 1948. second edition.

[Bir67] G. Birkhoff. *Lattice Theory*. Providence, R.I., 1967. third edition.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–166, 1950.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[Ege31] E. Egervary. On combinatorial properties of matrices. *Mat. Lapok*, 38:16–28, 1931.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proc. of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, (ACM SIGPLAN Notices)*, 24(1):183–194, January 1989.

[Gar96] V. K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publishers, Boston, MA, 1996.

[GM01] V. K. Garg and N. Mittal. On slicing a distributed computation. In *21st Intnatl. Conf. on Distributed Computing Systems (ICDCS' 01)*, pages 322–329, Washington - Brussels - Tokyo, April 2001. IEEE.

[Grä71] George Grätzer. *Lattice theory*. W.H. Freeman and Company, San Francisco, 1971.

[Grä03] George Grätzer. *General Lattice Theory*. Birkhäuser, Basel, 2003.

[Kon31] D. Konig. Graphen und matrizen. *Mat. es Fiz. Lapok*, 38:116–119, 1931.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the Intnatl. Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[MG01] N. Mittal and V. K. Garg. Slicing a distributed computation: Techniques and theory. In *5th Intnatl. Symp. on DIStributed Computing (DISC'01)*, October 2001.

[Szp37] E. Szpilrajn. La dimension et la mesure. *Fundamenta Mathematicae*, 28:81–89, 1937. (E. Szpilrajn = E. Marczewski).

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal for Parallel and Distributed Computing*, 1997. a preliminary version appeared in Proc. of the ACM Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993, pp.21 – 31.

[Wes04] D. B. West. *Order and Optimization*. 2004. pre-release version.