Symmetric chain decompositions of partially ordered sets

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ondrej Zjevik

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

John Greene, Dalibor Froncek

July, 2014
Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school. First, I have to express my deep appreciation to prof. John Greene who has been providing me with outstanding guidelines and recommendations ever since he introduced me to this topic. I am especially grateful for his persistence in reading various copies of this paper.

I am indebted to prof. Dalibor Froncek for encouraging me to apply to UMD, his support and for serving on my committee.

I would also like to thank prof. Dougles Dunham for serving on my committee.
Abstract

A partially ordered set, or poset, is a set of elements and a binary relation which determines an order within elements. Various combinatorial properties of finite and ordered posets have been extensively studied during the last 4 decades. The Sperner property states that the size of the largest subset of pairwise incomparable elements does not exceed the size of the largest level set in an ordered poset. Since a symmetric chain decomposition is a sufficient condition for the Sperner property, we may prove the Sperner property by finding a symmetric chain decomposition for a poset.

In this paper we focus on three types of posets: the Boolean algebra, inversion poset and the Young's lattice. An explicit construction for a symmetric chain decomposition is known only for Boolean algebras. No explicit construction has been found for inversion posets and Young's lattices, a symmetric chain decomposition was found only for a small subset of these posets. Using a maximal flow, we introduce an algorithm for finding this decomposition. We present our results and discuss two implementations of this algorithm.
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Formal definitions and fundamental properties of posets</td>
<td>6</td>
</tr>
<tr>
<td>3 Additional properties of posets</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Division and Boolean posets</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Inversion posets and Young’s lattices</td>
<td>20</td>
</tr>
<tr>
<td>4 The algorithm</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Finding a flow in a graph</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Algorithm description</td>
<td>31</td>
</tr>
<tr>
<td>5 Results</td>
<td>38</td>
</tr>
<tr>
<td>5.1 Future work</td>
<td>41</td>
</tr>
<tr>
<td>References</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of posets with the maximal size for which a symmetric saturated</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>chain decomposition was found by our program.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Number of different longest chains in inversion posets $I_n$, where $4 \leq n \leq 10$.</td>
<td>41</td>
</tr>
<tr>
<td>B.1</td>
<td>Performance of each program on selected posets.</td>
<td>50</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Examples of posets.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Different saturated chain decompositions of a poset depicted in figure 1.1(d)</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of a direct product of two ranked posets.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Examples of two different lattices.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Diagrams of an inversion poset and a Young’s lattice.</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagrams of inversion posets for multisets.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of theorem 2.2.</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>An isomorphism between $L(3,2)$ and $S(5,3)$.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>An illustration of an isomorphism between $I_{{1,2,3}}$ and $I_{{1,2,3,}}$.</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Two different flows in a graph.</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of two different maximal flows between middle layers and</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>how they can affect the poset decomposition.</td>
<td></td>
</tr>
<tr>
<td>B.1</td>
<td>Performance of programs on inversion posets.</td>
<td>51</td>
</tr>
<tr>
<td>B.2</td>
<td>Performance of programs on Young’s lattices.</td>
<td>52</td>
</tr>
<tr>
<td>B.3</td>
<td>Inversion poset, $I_4$.</td>
<td>53</td>
</tr>
<tr>
<td>B.4</td>
<td>$I_5$: 1,2,3,4,5 FULL_NO_LABELS.png</td>
<td>54</td>
</tr>
<tr>
<td>B.5</td>
<td>$I_5$: 1,2,3,4,5 SPARSE_NO_LABELS.png</td>
<td>55</td>
</tr>
<tr>
<td>B.6</td>
<td>$I_6$: 1,2,3,4,5,6 FULL_NO_LABELS.png</td>
<td>56</td>
</tr>
<tr>
<td>B.7</td>
<td>$I_6$: 1,2,3,4,5,6 SPARSE_NO_LABELS.png</td>
<td>57</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A partially ordered set (poset) is an ordered pair of a set with an order relation, e.g., $(\mathcal{P}, \mathcal{R})$. This relation $\mathcal{R}$ is

- reflexive: each element is related to itself, i.e., $\forall a \in \mathcal{P}, (a, a) \in \mathcal{R}$,
- transitive: if $(a, b) \in \mathcal{R}$ and $(b, c) \in \mathcal{R}$, then $(a, c) \in \mathcal{R}$,
- antisymmetric: if $(a, b) \in \mathcal{R}$ and $a \neq b$, then $(b, a) \notin \mathcal{R}$.

If two elements are related, $(a, b) \in \mathcal{R}$, we use $a \leq b$ to denote this relation. If we would like to emphasis a specific poset $\mathcal{P}$ we use $a \leq_{\mathcal{P}} b$ instead. The set $\mathcal{P}$ with the relation $\mathcal{R}$ can be considered as a directed graph where $\mathcal{P}$ is the vertex set and $\mathcal{R}$ is the edge set. The vertex set $\mathcal{P}$ can be finite or infinite. A simple example is when $\mathcal{P} = \mathbb{Z}^+$ and the relation is defined as $a \leq b$ if $b - a \in \mathbb{N}$. This poset is infinite and since we can compare each pair of positive integers, one integer is always less or equal then another, we say that this order is total. A partial order on the same set $\mathcal{P} = \mathbb{Z}^+$ is given by divisibility. That is, $a \leq b$ when $b$ is divisible by $a$. For this order relation, not every pair of positive numbers is comparable. For example if we consider 3 and 5, neither 3 divides 5 nor 3 is divisible by 5. We say 3 and 5 are incomparable. If we restrict $\mathcal{P}$ to all divisors of 36 we get a finite poset. These types of posets are called division posets, since the relation is
defined by division. Finite posets will be our main focus throughout this paper. More
types of posets and precise definitions will be introduced in Chapter 2.

The directed graph of the finite division poset of divisors of 36 is shown in Figure 1.1(a). This is the only time we show the entire directed graph of a poset. We will use Hasse diagrams for graphical representations of posets because they contain the same amount of information but have fewer edges than directed graphs. An edge \((a, b)\) is in a Hasse diagram if and only if there is no \(c\) such that oriented paths from \(a\) to \(c\) and from \(c\) to \(b\) would exist. Additionally, Hasse diagrams do not have oriented edges but we assume that every edge is oriented in upward direction. We can see the difference between a directed graph and a Hasse diagram of a division poset in Figures 1.1(a) and

Figure 1.1: Examples of posets.
Posets have a variety of possible properties. A poset is \textit{ranked} if elements of the Hasse diagram can be partitioned into horizontal level sets such that edges are only between the closest sets. Since there are no edges within each layer nor between layers which are separated by at least one other layer, every oriented path between two elements must have the same length. Each Hasse diagram in Figure 1.1 is ranked except (c). This poset is not ranked because there are two paths from 2 to 8 with different lengths. The path $2 \rightarrow 5 \rightarrow 8$ has length 2 but the path $2 \rightarrow 4 \rightarrow 7 \rightarrow 8$ has length 3.

An \textit{upper bound} of a subset $A$ of $\mathcal{P}$ is an element $p$ such that $a \leq p$ for all $a \in A$. Let $U$ be a set of all upper bounds of $A$. If there exists an upper bound $q$ such that $q \leq p$ for all $p \in U$, we call $q$ the \textit{least upper bound} of $A$. \textit{Lower bounds} and the \textit{greatest lower bound} are defined similarly. Notice that the definition of the least upper bound and the greatest lower bound imply uniqueness, since the relation is antisymmetric.

A poset is called a \textit{lattice} if the least upper bound and the greatest lower bound exist for every pair of elements. As we could suppose the division poset and the poset in Figure 1.1(d) are lattices, but the poset in Figure 1.1(c) is not a lattice. If we look at the subset $\{3, 4, 6, 7\}$ in Figure 1.1(c), we can find that $\{3, 4\}$ does not have the least upper bound and $\{6, 7\}$ does not have the greatest lower bound.

A \textit{decomposition} of a poset is a partition of $\mathcal{P}$ into disjoint subsets $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k$ such that $\bigcup_{i=0}^k \mathcal{P}_i = \mathcal{P}$. There are many types of decomposition but we will focus on symmetric saturated chain decompositions (SSCD). An ordered $n$-tuple $(c_1, c_2, \ldots, c_n)$ is called a \textit{chain}, with length $n-1$, of the poset $\mathcal{P}$ if $c_i \neq c_{i+1}$ and $c_i \leq c_{i+1}$ for $1 \leq i < n$. A chain $(c_1, c_2, \ldots, c_n)$ is \textit{saturated} if it cannot be internally extended, i.e.; $(c_i, c_{i+1})$ is an edge in the Hasse diagram of the poset for $1 \leq i < n$. A saturated chain is \textit{symmetric} if it starts and ends at levels whose distance from the middle level(s) of the poset is the same. Figure 1.1(b) contains a diagram with two highlighted symmetric saturated chains. A poset has a SSCD if $\mathcal{P}$ can be decomposed into symmetric saturated chains.
Figure 1.2: Different saturated chain decompositions of a poset depicted in figure 1.1(d).

The division poset (lattice) in Figure 1.1(b) is decomposable into three symmetric saturated chains. The chains, as highlighted in the Figure, are (1, 3, 9, 18, 36), (2, 4, 12) and (6). The lattice in Figure 1.1(d) cannot be decomposed into symmetric saturated chains. If this lattice were decomposable, there would have to be a longest chain from 1 to 12. We have exactly 4 choices for this chain, the choice depends on which element from the middle layer is included. Each choice will leave two non symmetric chains as we can see in Figure 1.2.

Before we introduce an algorithm for finding a symmetric saturated chain decomposition for a poset; fundamental terminology, a few properties of posets and essential theorems will be introduced in chapter 2. We also explain a connection between Sperner’s property and the existence of a SSCD in this chapter.

The rest of the proofs are placed in chapter 3. In this chapter we describe isomorphisms between division and boolean posets and within inversion posets.
Chapter 4 introduces an algorithm for finding a SSCD for posets. This algorithm uses a maximum flow in a graph to construct a SSCD. Two different algorithms for finding a maximum flow in a graph are discussed.

Chapter 5 contains a brief description of two implementations of the algorithm from chapter 4. We provide a table with all posets for which our implementation finds a SSCD and discuss suggestions for a future work.

Both of our implementations have similar interface and a description of these interfaces is given in appendix A. A performance comparison between the two implementations, examples of a SSCD for selected posets and source codes of our implementations are placed in appendixes, also.
Chapter 2

Formal definitions and fundamental properties of posets

The theory of partially ordered sets investigates a poset as a graph or as a set of elements. This set can be infinite but we will focus on finite sets only. At the beginning of this chapter the most common notation and definitions from Graph Theory are introduced. Concrete examples of posets with known attributes will be introduced after definitions. Most of these definitions can be found in [1] or in [10].

Let $P$ be a poset, $R$ its relation and $p, p', q, q'$ elements of $P$. We say that $p$ and $q$ are comparable if $(p, q) \in R$ or $(q, p) \in R$, and we typically write this as $p \leq q$ or $q \leq p$. In the case when $p$ and $q$ are comparable and $p \neq q$ either $p < q$ or $q < p$ is valid. An element $p$ covers $q$, denoted as $q < p$, if $q < p$ and if $q < p' \leq p$ implies $p = p'$. An element $p$ is a minimal element of $P$ if $q \leq p$ implies $q = p$; a maximal element is defined similarly.

A ranked poset is equipped with a rank function $\rho$. This is a function from $P$ to $\mathbb{N}$ satisfying $\rho(p) = \rho(q) + 1$ if $p$ covers $q$. The rank of a poset is defined as the largest rank over all elements from the poset. Since every finite lattice has only one minimal element, the rank of this minimal element is zero.
A chain $C = (c_1, c_2, \ldots, c_k)$ is saturated if $c_1 \prec c_2 \prec \cdots \prec c_k$. This chain is maximal if there is no $c_0$ such that $(c_0, c_1, c_2, \ldots, c_k)$ or $(c_1, c_2, \ldots, c_k, c_0)$ would be a valid chain in $\mathcal{P}$. Each of the highlighted chains in Figure 1.2 is saturated.

A saturated chain $(c_1, c_2, \ldots, c_n)$ is symmetric if $\rho(c_1) + \rho(c_n) = \rho(\mathcal{P})$. Note that $\rho(c_1) + \rho(c_n) = \rho(\mathcal{P})$ implies that $\rho(c_{1+i}) + \rho(c_{n-i}) = \rho(\mathcal{P})$ for $0 \leq i < n/2$.

A set $Q \subset \mathcal{P}$ is an antichain if every pair of elements in $Q$ is incomparable, i.e., there are no $p, q \in Q$ such that $p \neq q$ and $p \leq q$ or $q \leq p$. Sets $\{2, 7, 8\}, \{2, 3, 4\}, \{6, 9, 11\}, \{5, 6, 7, 8\}$ are examples of antichains in Figure 1.2. The direct product of two posets $(\mathcal{P}, \mathcal{R})$ and $(\mathcal{Q}, \mathcal{S})$, $(\mathcal{P}, \mathcal{R}) \times (\mathcal{Q}, \mathcal{S})$, is a poset with the set defined as a Cartesian product of $\mathcal{P}$ and $\mathcal{Q}$, the order relation of $\mathcal{P} \times \mathcal{Q}$ is given by $(p, q) \leq (p', q')$ if and only if $p \leq p'$ and $q \leq q'$ in $\mathcal{R}$ and $\mathcal{S}$, respectively. Consider a direct product of divisors of 36 ordered by divisibility and $\mathbb{N}$ ordered by a total order $p \leq q$ if $q - p \in \mathbb{N}$ shown in Figure 2.1. An example of a saturated chain is $\left((1, 0), (2, 0), (2, 1), (6, 1), (6, 2)\right)$. Since 2 and 3 are incomparable in the division poset, the set $\{(2, 0), (3, 1)\}$ is an antichain in $\mathcal{P} \times \mathcal{Q}$. If $\mathcal{P}$ and $\mathcal{Q}$ are ranked, then so is $\mathcal{P} \times \mathcal{Q}$ by $\rho((p, q)) = \rho_{\mathcal{P}}(p) + \rho_{\mathcal{Q}}(q)$.

If $\mathcal{P}$ is a ranked poset, its elements can be partitioned into layers, the $i$th layer of a ranked poset $\mathcal{P}$ is denoted as $N_i(\mathcal{P}) := \{p \in \mathcal{P} | \rho(p) = i\}$. The size of $N_i(\mathcal{P})$ is called the $i$th Whitney number, denoted by $W_i = |N_i(\mathcal{P})|$. Note that $W_j = 0$ if $j < 0$ or $j > \rho(\mathcal{P})$.
from the definition of $N_j(\mathcal{P})$, since each layer $N_j(\mathcal{P})$ is empty. A generating function of a ranked poset $\mathcal{P}$ is given by

$$GF(\mathcal{P}) = \sum_{w \in \mathcal{P}} q^{\rho(w)} = \sum_{i=0}^{\rho(\mathcal{P})} W_i q^i.$$  

We stated that posets in Figures 1.1(b) and 1.1(d) are ranked. Hence, there is a generating function for each poset. The generating function for the poset in Figure 1.1(d) is $1 + 3q + 4q^2 + 3q^3 + q^4$, $GF(D_{36}) = 1 + 2q + 3q^2 + 2q^3 + q^4$.

A poset is rank unimodal if there exist $j$ such that $W_0 \leq W_1 \leq \cdots \leq W_{j-1} \leq W_j \geq W_{j+1} \geq \cdots \geq W_{\rho(\mathcal{P})}$. A rank symmetric poset is a ranked poset whose Whitney numbers are symmetric with respect to the middle layer(s), i.e.; $W_i = W_{\rho(\mathcal{P})-i}$ for $0 \leq i < \rho(\mathcal{P})/2$. Again, if we look at Figures 1.1(b) and 1.1(d), sequences of Whitney numbers are $(1, 2, 3, 2, 1)$ and $(1, 3, 4, 3, 1)$, respectively. Both sequences are symmetric and unimodal but only the division poset has a SSCD.

Posets $\mathcal{P}$ and $\mathcal{Q}$ are isomorphic, $\mathcal{P} \cong \mathcal{Q}$, if there is a bijective mapping $\varphi$ from $\mathcal{P}$ onto $\mathcal{Q}$ such that $p \leq \mathcal{P} q$ if and only if $\varphi(p) \leq \mathcal{Q} \varphi(q)$.

**Theorem 2.1** (Sperner’s Theorem). Let $n$ be a positive integer and $\mathcal{F}$ be a family of subsets of $[n] := \{1, 2, \ldots, n\}$ such that no member of $\mathcal{F}$ is included in another member of $\mathcal{F}$, that is, for all $X, Y \in \mathcal{F}$ we have $X \nsubseteq Y$. Then

$$|\mathcal{F}| \leq \begin{cases} \binom{n/2}{\rho(\mathcal{P})} & \text{if } n \text{ is even,} \\ \binom{n}{(n-1)/2} & \text{if } n \text{ is odd.} \end{cases}$$

Sperner’s Theorem was at the beginning of the Poset Theory and many important results were obtained by using this theorem. Three different proofs of this theorem can be found in [4], as well as other important results. We will prove this theorem in the next chapter using a symmetric chain decomposition of the family of subsets of $\{1, 2, \ldots, n\}$.

Since each level set of a poset contains only incomparable elements, a lower bound
for the size of the largest antichain is given by the largest Whitney number. Sperner's
Theorem was stated for the family of subsets of a set ordered by inclusion and it says
that this lower bound is tight, the size of the biggest antichain is equal to the largest
Whitney number. A generalization of this theorem is available for other types of posets;
a ranked poset $P$ has the Sperner property if the size of any antichain is less than or
equal to the biggest Whitney number.

The two lattices in Figure 1.1 both have the Sperner property. However, the lattice
in Figure 2.2(b) is rank symmetric but it does not have the Sperner property nor a
SSCD, the antichain \{3, 4, 5, 6, 7\} is larger than the size of the middle layer.

Unfortunately, it is not true that every poset with the Sperner property has a sym-
metric saturated chain decomposition. An example of such a poset is in Figure 1.1(d).
This poset satisfies the Sperner property, the size of each antichain is not bigger than 4
which is the size of the biggest antichain formed by the middle layer. However, there is
no SSCD of this poset as we discussed in Chapter 1.

We will find a use of $q$-functions in examining rank functions. A $q$-analog of $n$ is
defined as $\{n\}_q = \frac{1 - q^n}{1 - q}$ and we often want to evaluate the $q$-analog of $n$ for $q = 1$. This

Figure 2.2: Examples of two different lattices.
value does not exist, but we can find a limit as $q$ approaches 1,

$$\lim_{q \to 1} \frac{1 - q^n}{1 - q} = \lim_{q \to 1} \frac{(1 - q)(1 + q + q^2 + \cdots + q^{n-1})}{1 - q} = \lim_{q \to 1} 1 + q + q^2 + \cdots + q^{n-1} = n.$$ 

Additionally, we will use a continuous extension of $q$-analog of $n$,

$$\frac{1 - q^n}{1 - q} \equiv 1 + q + q^2 + \cdots + q^{n-1},$$

because $\frac{1 - q^n}{1 - q} = 1 + q + q^2 + \cdots + q^{n-1}$ almost everywhere.

A $q$-factorial of $n$, $\{n\}_q!$, is defined as a product of $q$-analogs of $i$, where $0 < i \leq n$. Hence,

$$\{n\}_q! = \{n\}_q \cdot \{n-1\}_q \cdot \{1\}_q$$

$$= \frac{1 - q^n}{1 - q} \cdot \frac{1 - q^{n-1}}{1 - q} \cdots \frac{1 - q}{1 - q}$$

$$\equiv (1 + q + q^2 + \cdots + q^{n-1})(1 + q + q^2 + \cdots + q^{n-2}) \cdots (1 + q)(1).$$

Similar as the binomial coefficient, the $q$-binomial coefficient is given by

$$\binom{n}{m}_q = \frac{\{n\}_q!}{\{m\}_q! \{n-m\}_q!}.$$ 

The Boolean algebra $B_n$ is the poset of all subsets of a set with $n$ elements, ordered by an inclusion. The size of $B_n$ is $2^n$ and its Whitney numbers correspond to a row in the Pascal’s triangle. The rank of an element is its cardinality.

The division poset $D_n$ is a poset of all divisors of $n$, ordered by divisibility. Let $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ be the prime factorization of $n$. Each element of the division poset can be written as $p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}$, where $0 \leq b_i \leq a_i$ for each $i$. The rank of $p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}$ is $\sum_{i=1}^k b_i$.

The Symmetric group $S_n$ is the group of all permutations of $(1, 2, \ldots, n)$. The sum of all Whitney numbers of any partial ordering on $S_n$ has to be $n!$ since there are $n!$ possible arrangements of $n$ elements in a sequence. Suppose $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ are elements of $S_n$. A pair $(i, j)$ is called an inversion of $a$ if $i < j$
Figure 2.3: Diagrams of an inversion poset and a Young’s lattice.

and $a_i > a_j$. The inversion sequence of $a$ is a sequence $(i_1, i_2, \ldots, i_n)$ where each $i_k$ represents how many elements in $a$ on the left of $a_k$ are greater than $a_k$. For example, consider a sequence $(7, 5, 8, 3, 2, 6, 1, 4)$. A pair $(2, 7)$ is an inversion since $5 > 1$ and the inversion sequence is $(0, 1, 0, 3, 4, 2, 6, 4)$. The inversion number of a permutation is the sum of its inversion sequence, i.e., $\text{inv}(a) = \sum_{j=1}^{n} i_j$. The set $S_n$ with a relation given by $a < b$, if $b$ is created from $a$ by interchanging $a_i$ with $a_{i+1}$ when $a_i < a_{i+1}$, is called the inversion poset, $I_n$. This binary relation is called the (weak) Bruhat order. The generating function is given by

$$GF(I_n) = \{n\} q! = \prod_{i=0}^{n-1} (1 + q + q^2 + \cdots + q^i)$$
and the rank of an element is its inversion number.

Similar to inversion posets we define an inversion poset for multisets ordered by inversions. Inversion posets for \( \{1, 1, 2, 3\} \) and \( \{1, 2, 2, 3\} \), \( I_{\{1,1,2,3\}} \) and \( I_{\{1,2,2,3\}} \), respectively, are shown in Figure 2.4. The generating function for \( I_N \), where \( N = \{n_1, n_1, n_2, n_2, \ldots, n_k, n_k\} \), is given by a q– multinomial coefficient,

\[
GF(I_N) = \left( \sum_{i=1}^{l_1} l_i \right)_{q} \cdot \left( \sum_{i=1}^{l_2} l_i \right)_{q} \cdots \left( \sum_{i=1}^{l_k} l_i \right)_{q} = \frac{\left\{ \sum_{i=1}^{l_i} l_i \right\} !}{\left\{ l_1 \right\} q! \cdot \left\{ l_2 \right\} q! \cdots \left\{ l_k \right\} q!}.
\]

We will prove this result in Chapter 3. For example, for \( I_{\{1,1,2,3\}} \) we have

\[
GF(I_{\{1,1,2,3\}}) = \frac{\{2 + 1 + 1\} q!}{\{2\} q! \cdot \{1\} q!} = \frac{(1 + q + q^2 + q^3)(1 + q + q^2)(1 + q)(1)}{(1 + q)(1)(1)}
\]

\[
= (1 + q + q^2 + q^3)(1 + q + q^2)
\]

\[
= 1 + 2q + 3q^2 + 3q^3 + 2q^4 + q^5.
\]
Note that the generating function for $I_{\{1,2,2,3\}}$ is the same as the generating function for $I_{\{1,1,2,3\}}$ even though these posets are not isomorphic.

The Young’s lattice, $L(m,n)$, is a poset of $n$-tuples $(a_1, a_2, \ldots, a_n)$, where $0 \leq a_1 \leq a_2 \leq \cdots \leq a_n \leq m$ with order relation $(a_1, a_2, \ldots, a_n) \leq (b_1, b_2, \ldots, b_n)$ if $a_i \leq b_i$ for all $i$. The rank function is given by a sum of all elements in an $n$-tuple, hence $\rho((a_1, a_2, \ldots, a_n)) = a_1 + a_2 + \cdots + a_n$. We can find in [11, p. 72] that the generating function for a Young’s lattice is

$$GF(L(m,n)) = \binom{m+n}{m} q = \frac{(1-q^{n+1})(1-q^{n+2}) \cdots (1-q^{n+m})}{(1-q)(1-q^2) \cdots (1-q^m)}.$$

**Theorem 2.2 ([2]).** If $P$ and $Q$ are posets with a symmetric saturated chain decomposition, then $P \times Q$ has a symmetric saturated chain decomposition.

**Proof.** Let $P_0, P_1, \ldots, P_m$ and $Q_0, Q_1, \ldots, Q_n$ be SSCD’s for $P$ and $Q$, respectively.

Consider a pair of chains $(P_i, Q_j)$, say

$$P_i = p_0 < p_1 < \cdots < p_k \text{ and } Q_j = q_0 < q_1 < \cdots < q_h.$$

We can create a new chain $E_l$, where

$$E_l = \left( (p_0, q_l), (p_1, q_l), \ldots, (p_{k-l}, q_l), (p_{k-l+1}, q_l), \ldots, (p_{k-1}, q_l), \ldots, (p_k, q_l) \right),$$

for $0 \leq l \leq \min\{k, h\}$. Each $E_l$ is evidently a saturated chain in $P \times Q$, since the change is only in the first or the second term and both $P_i$ and $Q_j$ are saturated. $P_i$ and $Q_j$ are
symmetric, so $\rho(P) = \rho(p_0) + \rho(p_k)$ and $\rho(Q) = \rho(q_0) + \rho(q_h)$. Additionally, since

$$\rho((p_0, q_l)) + \rho((p_k-l, q_h)) = \rho(p_0) + \rho(q_l) + \rho(p_k-l) + \rho(q_h)$$

$$= \rho(p_0) + \rho(q_k) - (k - l) + \rho(p_0) + (k - l) + \rho(q_h)$$

$$= \rho(p_0) + \rho(q_k) + \rho(p_0) + \rho(q_h)$$

$$= \rho(P) + \rho(Q)$$

$$= \rho(P \times Q),$$

$E_l$ is also symmetric. If $k \leq h$, the number of elements in all $E_l$’s for a pair $(P_i, Q_j)$ is

$$\sum_{l=0}^{k} |E_l| = \sum_{l=0}^{k} (k + h + 1 - 2l) = (k + 1)(k + h + 1) - 2 \sum_{l=0}^{k} l$$

$$= k^2 + kh + 2k + h + 1 - 2k \frac{k+1}{2}$$

$$= k^2 + kh + 2k + h + 1 - k(k + 1) = kh + k + h + 1$$

$$= (k + 1)(h + 1) = |P_i| \cdot |Q_j|.$$  

If $h < k$, the same result can be obtained by the same procedure. Since we can do this procedure for each pair of chains from the decomposition, the total number of elements in all $E_l$’s is

$$\sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{l=0}^{\min\{k,h\}} |E_l| = \sum_{i=0}^{m} \sum_{j=0}^{n} |P_i| \cdot |Q_j| = \sum_{i=0}^{m} |P_i| \cdot \sum_{j=0}^{n} |Q_j| = |P| \cdot |Q|.$$  

Hence we will obtain new symmetric chain decomposition of $P \times Q$ by this procedure.  

For example, consider a direct product of $I_3$ and a chain of three elements, $C = (c_1, c_2, c_3)$. A diagram of this direct product is shown in Figure 2.5. To follow the proof of Theorem 2.2 we pick a symmetric saturated chain in each poset. These chains are highlighted by thick solid line. These chains will yield, by Theorem 2.2, three symmetric saturated chains in the direct product, since the shortest chain contains three elements. If we follow the same notation, we have chains $E_0, E_1$ and $E_2$. The chain $E_0$ is the
longest chain and $E_2$ is the shortest. Additionally, these chains cover all elements in the Cartesian product of the original highlighted chain in $I_3$ and $C$. We can describe this construction easier if a diagram of the direct product is available. Given two saturated symmetric chains $P$ and $Q$, each from a different poset, a SSCD of the direct product $P \times Q$ can be found by following an algorithm:

1. Let $S = P \times Q$.
2. Pick an element $p$ with the smallest rank in $S$.
3. Construct a new symmetric saturated chain $E_i$ which starts at $p$, follows an upward path given by $P$ as far as it can and then continues to the right following a path given by $Q$.
4. If we have not used all elements in $P \times Q$ we have to shrink $S$ and find a next chain. In order to do that we have to remove $E_i$ from $S$ and go back to 2.

We will investigate a few types of posets which are created as a direct product of two or more ranked posets. To analyze their properties we prove the following theorems.

**Theorem 2.3 (H).** *The generating function for the direct product of two ranked posets*
\( \mathcal{P} \) and \( \mathcal{Q} \) is given by

\[
GF(\mathcal{P} \times \mathcal{Q}) = GF(\mathcal{P}) \cdot GF(\mathcal{Q}).
\]

**Proof.** Since the generating function of a poset \( \mathcal{P} \) is given by \( GF(\mathcal{P}) = \sum_{p \in \mathcal{P}} q^{\rho(p)} \), we can use this formula on \( \mathcal{P} \times \mathcal{Q} \)

\[
GF(\mathcal{P} \times \mathcal{Q}) = \sum_{(a,b) \in \mathcal{P} \times \mathcal{Q}} q^{\rho((a,b))} = \sum_{(a,b) \in \mathcal{P} \times \mathcal{Q}} q^{\rho(a)+\rho(b)} = \sum_{a \in \mathcal{P}} \sum_{b \in \mathcal{Q}} q^{\rho(a)+\rho(b)}
\]

\[
= \sum_{a \in \mathcal{P}} q^{\rho(a)} \sum_{b \in \mathcal{Q}} q^{\rho(b)} = GF(\mathcal{P}) \cdot GF(\mathcal{Q}).
\]

\( \square \)

**Theorem 2.4.** A poset with a symmetric saturated chain decomposition has the Sperner property.

**Proof.** If a poset \( \mathcal{P} \) has a SSCD then the largest Whitney number is \( W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}) \). We need to prove that every antichain has size less than or equal to \( W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}) \).

Let \( A = \{a_1, a_2, \ldots, a_l\} \) be an antichain and \( C = \{C_1, C_2, \ldots, C_k\} \) be a SSCD of \( \mathcal{P} \). Since \( C \) contains all elements of \( \mathcal{P} \), suppose, without loss of generality that \( a_1 \in C_1 \). All elements in \( C_1 \) are comparable with \( a_1 \). Hence, without loss of generality, say that \( a_2 \in C_2 \). Since \( a_3 \) is not comparable with neither \( a_1 \) nor \( a_2 \) it cannot be in \( C_1 \) or \( C_2 \), say \( a_3 \) is in \( C_3 \). We can conclude, by reordering the \( C \)’s as necessary, that \( a_i \in C_i \) for each \( i \leq l \), therefore \( l \leq k \).

It is clear from the definition of SSCD that the biggest Whitney number, the size of the middle layer(s), is equal to the number of chains in a SSCD. Therefore,

\[
l \leq k = W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}) \implies l \leq W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}),
\]

which completes the proof. \( \square \)
Chapter 3

Additional properties of posets

3.1 Division and Boolean posets

**Theorem 3.1.** A division poset $D_n$ is isomorphic to the direct product of chains $C = (1 < p_1 < p_2^2 < \cdots < p_1^{a_1}) \times (1 < p_2 < p_2^2 < \cdots < p_2^{a_2}) \times \cdots \times (1 < p_k < p_k^2 < \cdots < p_k^{a_k})$, where $n = p_1^{a_1}p_2^{a_2} \cdots p_k^{a_k}$ is the prime factorization of $n$.

**Proof.** Let $I$ be a function from $D_n$ to $C$ given by

$$I(p_1^{b_1}p_2^{b_2} \cdots p_k^{b_k}) = (p_1^{b_1}, p_2^{b_2}, \ldots, p_k^{b_k}).$$

It is clear that the function $I$ is one-to-one. Let $p = p_1^{b_1}p_2^{b_2} \cdots p_k^{b_k}$ and $q = p_1^{c_1}p_2^{c_2} \cdots p_k^{c_k}$, where $b_i \leq c_i$ for each $i$. Apparently, $q$ is divisible by $p$, $q/p = p_1^{c_1-b_1}p_2^{c_2-b_2} \cdots p_k^{c_k-b_k}$ where each $c_i - b_i \geq 0$ and therefore $p \leq_D n$.

Since $I(p) = (p_1^{b_1}, p_2^{b_2}, \ldots, p_k^{b_k})$ we can construct a chain $((p_1^{b_1}, p_2^{b_2}, \ldots, p_k^{b_k})) \leq (p_1^{c_1}, p_2^{c_2}, \ldots, p_k^{c_k}) \leq \cdots \leq (p_1^{c_1}, p_2^{c_2}, \ldots, p_k^{c_k}) = I(q)$ in $C$, therefore $I(p) \leq_C I(q)$ from the transitive property of the relation.

To finish the proof we need to show that $I(p) \leq_C I(q)$ implies $p \leq_D n$ $q$. To show this, again suppose that $I(p) = (p_1^{b_1}, p_2^{b_2}, \ldots, p_k^{b_k})$ and $I(q) = (p_1^{c_1}, p_2^{c_2}, \ldots, p_k^{c_k})$, where $b_i \leq c_i$ for each $i$. It is clear, since $I$ is one-to-one, that $p$ divides $q$. ∎
We can find the generating function for a division poset by combining Theorems 3.1 and 2.3.

**Corollary 3.2.** The generating function for a division poset $D_n$, where $n = p_1^{a_1}p_2^{a_2} \cdots p_k^{a_k}$ is the prime factorization of $n$, is given by

$$GF(D_n) = \prod_{i=1}^{k} (1 + q + q^2 + \cdots + q^{a_i}).$$

For example the generating function for $D_{36}$ is

$$GF(D_{36}) = GF(D_{2^2 \cdot 3^2}) = (1 + q + q^2) \cdot (1 + q + q^2) = 1 + 2q + 3q^2 + 2q^3 + q^4.$$

We can compare this rank function with a diagram of the division poset $D_{36}$ in Figure 1.1(b), there is one element with rank zero, two with rank one and three with rank two.

Now, we may focus on Boolean posets and show that a Boolean poset has the same properties as a division poset. In fact, a Boolean poset is a special case of a division poset $D_n$.

**Theorem 3.3.** A Boolean poset $B_k$ is isomorphic to $D_n$, where $n = p_1 p_2 \cdots p_k$ and each $p_i$ is a distinct prime number.

**Proof.** Since elements of $B_k$ are subsets of $\{1, 2, \ldots, k\}$, we can introduce an one-to-one function $I$ from $B_k$ into $D_n$. Say

$$I(\{a_1, a_2, \ldots, a_i\}) = p_{a_1}p_{a_2}\cdots p_{a_i},$$

where $I(\emptyset) = 1$. Consider two subsets of $B_k$, $A$ and $B$, such that $A \subseteq B$. Suppose that $A = \{a_1, a_2, \ldots, a_i\}$ and $B = \{a_1, a_2, \ldots, a_i, b_1, b_2, \ldots, b_j\}$ since $A$ is a subset of $B$. We have to show that $I(A) \leq_{D_n} I(B)$, i.e., that the function $I$ respects the order. Observe that

$$I(A) = p_{a_1}p_{a_2}\cdots p_{a_i} = \frac{p_{a_1}p_{a_2}\cdots p_{a_i}p_{b_1}p_{b_2}\cdots p_{b_j}}{p_{b_1}p_{b_2}\cdots p_{b_j}} = \frac{I(B)}{p_{b_1}p_{b_2}\cdots p_{b_j}}.$$
Thus
\[
\frac{I(B)}{I(A)} = p_{b_1} p_{b_2} \cdots p_{b_j},
\]
which indicates that \(I(A)\) divides \(I(B)\), therefore \(I(A) \leq_{D_n} I(B)\). \(\square\)

Now we know that a Boolean poset is a ranked poset with a SSCD. Using Corollary 3.2 the generating function for \(B_n\) is given by
\[
GF(B_n) = \prod_{i=1}^{n} (1 + q) = \sum_{i=0}^{n} \binom{n}{i} q^i,
\]
which explains why a sequence of Whitney numbers corresponds to a row in the Pascal’s triangle.

**Theorem 3.4.** Given a division poset \(D_n\), where \(n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}\) is the prime factorization of \(n\), we can find an isomorphism \(I\) between \(D_n\) with a standard order and \(D_n\) with a reversed order.

**Proof.** We will prove that this isomorphism is given by
\[
I(p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}) = \frac{n}{p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}}.
\]
The function \(I\) is one-to-one, therefore we have to check only that if \(b \leq D_n c\) then \(I(b) \geq D_n I(c)\). Since \(c\) is divisible by \(b\) we have
\[
\frac{c}{b} = \frac{1}{p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}} \cdot \frac{n}{p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}} = \frac{I(b)}{I(c)},
\]
as required. \(\square\)

Given a poset \(P\), the poset with the same underlying set but reversed order is called the **dual** of \(P\), \(P^*\). If a poset and its dual are isomorphic, then this poset is called **self–dual**.
Theorems 2.2 and 3.1 imply that each division poset has a SSCD, therefore each division poset has the Sperner property by Theorem 2.4. In particular, the Boolean poset $B_n$ satisfies the Sperner property, proving Theorem 2.1.

3.2 Inversion posets and Young’s lattices

Let $S(m + n, m)$ be a poset of all bit strings of length $m + n$ with $m$ ones. We define a partial order which is the same as for an inversion poset $I_n$. That is, $p \leq q$ if $q$ is created by some number of adjacent transpositions of elements in $p$, where the second element is larger than the first. The smallest and the greatest elements are $0000 \cdots 000 \underbrace{11 \cdots 11}_n$ and $11 \cdots 111 \underbrace{000 \cdots 000}_m$, respectively.

**Theorem 3.5.** $S(m + n, m)$ and $L(m, n)$ are isomorphic.

**Proof.** Let I be a function from $L(m, n)$ to $S(m + n, m)$ given by

$$I\left((a_1, a_2, \ldots, a_n)\right) = \frac{1 \cdots 101 \cdots 10}{a_1} \frac{101 \cdots 1}{a_2 - a_1} \frac{1 \cdots 10}{a_3 - a_2} \cdots \frac{1 \cdots 101}{a_n - a_{n-1}} \frac{1}{m - a_n}$$

To show that $I$ is one-to-one suppose that $a = (a_1, a_2, \ldots, a_n), b = (b_1, b_2, \ldots, b_n)$ and $I(a) = I(b)$. Then

$$I(a) = I(b)$$

$$\frac{1 \cdots 101 \cdots 10}{a_1} \frac{101 \cdots 1}{a_2 - a_1} \frac{1 \cdots 10}{a_3 - a_2} \cdots \frac{1 \cdots 101}{a_n - a_{n-1}} \frac{1}{m - a_n} = \frac{1 \cdots 101 \cdots 101 \cdots 10}{b_1} \frac{101 \cdots 1}{b_2 - b_1} \frac{1 \cdots 10}{b_3 - b_2} \cdots \frac{1 \cdots 101}{b_n - b_{n-1}} \frac{1}{m - a_n}$$

If the two strings are the same, then the number of ones separated by a zero has to be
the same. Hence, we get this system of equations

\[
\begin{align*}
a_1 &= b_1 \\
a_2 - a_1 &= b_2 - b_1 \\
a_3 - a_2 &= b_3 - b_2 \\
&\vdots \\
a_n - a_{n-1} &= b_n - b_{n-1} \\
m - a_n &= m - b_n.
\end{align*}
\]

We can use the solution of the first equation, \(a_1 = b_1\), to solve the second equation and get \(a_2 = b_2\). It is easy to see that the solution of this system is \(a_i = b_i\) for \(1 \leq i \leq n\) and therefore \(a = b\).

To show that the function \(I\) preserves order, let \(a \leq_L b\). \(I(a)\) is less than or equal to \(I(b)\) if the number of ones on the left from each zero in \(I(a)\) is less than or equal to the number of ones on the left from each zero in \(I(b)\). For \(I(a)\), the number of ones before the first zero is \(a_1\), before the second zero it is \(a_1 + a_2 - a_1 = a_2\), there are \(a_3\) ones before the third zero, etc., up to \(m\). Similarly, the sequence for \(b\) is \(b_1, b_2, \ldots, m\). Each \(a_i\) is less than or equal to \(b_i\) since \(a \leq_L b\) and therefore \(I(a) \leq I(b)\).

The opposite, \(I(a) \leq I(b)\) implies \(a \leq_L b\) comes from the same effort of comparing number of ones which precede a zero.

The function \(I\) from the proof above is one-to-one and therefore it has an inverse. The function \(I^{-1}\) takes a bit string \(s\) and returns a vector containing only those elements from the inversion sequence of \(s\) which correspond to zeros in \(s\). An example of an isomorphism between \(L(2, 3)\) and \(S(5, 2)\) can be seen on Figure 3.1.

Kyle Krueger in his Master’s project\[8\] proved that \(\binom{m+n}{m}_q\) is the generating function of \(S(m+n, m)\). This result together with the previous theorem shows that the generation function for \(L(m, n)\) is

\[GF(L(m, n)) = \binom{m+n}{m}_q.\]
Theorem 3.6. The generating function for an inversion poset for a multiset $I_N$, where $N = \{n_{l_1,1}, n_{l_1,2}, \ldots, n_{l_1,1}, n_{l_2,1}, n_{l_2,2}, \ldots, n_{l_2,1}, n_{l_k,1}, n_{l_k,2}, \ldots, n_{l_k,1}\}$ is

$$
\left( \sum_{i=1}^{k} l_i \right)_{q} \left( l_{1}, l_{2}, \ldots, l_{k} \right)_{q}.
$$

Proof. We prove this theorem using strong induction on $k$, the number of distinct elements in $N$. Let $k = 2$ be our base case. Krueger in [8] provides the formula for the generating function in this base case as a $q$-binomial coefficient,

$$
\binom{l_1 + l_2}{l_1}_q = \frac{(l_1 + l_2)_q!}{l_1! \cdot l_2!}.
$$

This is the same formula that we get from our hypothesis, since

$$
\left( \sum_{i=1}^{2} l_i \right)_{q} \left( l_{1}, l_{2} \right)_{q} = \frac{(l_1 + l_2)_q!}{l_1! \cdot l_2!}.
$$
Assume that the formula holds for each $k$ between 1 and $m$. The generating function for $I_N$, where $N = \{n_1, n_2, \ldots, n_{l_1}, n_2, n_2, \ldots, n_{l_2}, \ldots, n_m, n_m, \ldots, n_m\}$, can be found in a few steps. First, let us recall a definition of the generating function for the inversion poset

$$GF(I_N) = \sum_{w \in I_N} q^{\text{inv}(w)},$$

where the inversion number of $w$ is a sum of its inversion sequence. Additionally, we introduce a bijection $\varphi : I_N \rightarrow W' \times W''$, where $\varphi(w) = (w', w'')$. The permutation $w'$ is a copy of $w$ without any $n_m$'s and $w''$ is a copy of $w$ where each element smaller than $n_m$ is replaced by zero. For example if $N = \{1, 1, 2, 2, 3, 3, 4, 4\}$ and $w = (3, 1, 2, 4, 1, 2, 3, 2)$ then $w' = (3, 1, 2, 1, 2, 3, 2)$, $w'' = (0, 0, 4, 0, 0, 4, 0, 0)$, $\text{inv}(w) = 13$, $\text{inv}(w') = 7$ and $\text{inv}(w'') = 6$.

This bijection splits the inversion number of $w$ into two parts. The inversion number of $w''$ expresses the contribution of the $n_m$'s to the inversion number of $w$ and the inversion number of $w'$ indicates the contribution of all other elements. We can represent this relation by the following equation,

$$\text{inv}(w) = \text{inv}(w') + \text{inv}(w'').$$

It is apparent that $\varphi$ is an injective function. To show that it is also surjective, given a pair $w', w''$ we can find $w$ by replacing all zeros in $w''$ with elements from $w'$ in a given order.

This bijection allows to rewrite the generating function for $I_N$,

$$\sum_{w \in I_N} q^{\text{inv}(w)} = \sum_{w \in I_N} q^{\text{inv}(w') + \text{inv}(w'')} = \sum_{(w', w'') = \varphi(w)} q^{\text{inv}(w')} \cdot q^{\text{inv}(w'')}$$

$$= \sum_{w' \in W'} q^{\text{inv}(w')} \cdot \sum_{w'' \in W''} q^{\text{inv}(w'')}.$$
Since $W'$ contains all permutations of $\{n_1, \ldots, n_1, n_2, \ldots, n_2, \ldots, n_{m-1}, \ldots, n_{m-1}\}$, the generating function will be the same as for $I_{\{n_1, \ldots, n_1, n_2, \ldots, n_2, \ldots, n_{m-1}, \ldots, n_{m-1}\}}$ and therefore by the induction assumption,

$$\sum_{w' \in W'} q^{\text{inv}(w')} = \left( \sum_{i=1}^{m-1} l_i \right)_{q=q} \left( l_1, l_2, \ldots, l_{m-1} \right)_{q} \frac{\left\{ \sum_{i=1}^{m-1} l_i \right\}_q!}{\left\{ l_1 \right\}_q \cdot \left\{ l_2 \right\}_q \cdots \left\{ l_{m-1} \right\}_q!}. $$

Similarly, $W''$ contains all permutations of $\{0, 0, \ldots, 0, n_m, \ldots, n_m\}$ and the generating function will be the same as the generating function for $I_{\{0, 0, \ldots, 0, n_m, \ldots, n_m\}}$. This function is our base case, thus

$$\sum_{w'' \in W''} q^{\text{inv}(w'')} = \left( \sum_{i=1}^{m-1} l_i + l_m \right)_{q=q} \left( l_1, l_2, \ldots, l_{m-1} \right)_{q} \frac{\left\{ \sum_{i=1}^{m} l_i \right\}_q!}{\left\{ \sum_{i=1}^{m-1} l_i \right\}_q \cdot \left\{ l_m \right\}_q!}. $$

Using the general definition of a generating function for an inversion poset we can describe the generating function for $I_{\{n_1, n_1, \ldots, n_1, n_2, n_2, \ldots, n_2, \ldots, n_{m-1}, n_{m-1}, \ldots, n_{m-1}\}}$ as a product of the q-multinomial coefficient and the q-binomial coefficient,

$$\frac{\left\{ \sum_{i=1}^{m-1} l_i \right\}_q!}{\left\{ l_1 \right\}_q \cdot \left\{ l_2 \right\}_q \cdots \left\{ l_{m-1} \right\}_q!} \cdot \frac{\left\{ \sum_{i=1}^{m} l_i \right\}_q!}{\left\{ l_1 \right\}_q \cdot \left\{ l_2 \right\}_q \cdots \left\{ l_m \right\}_q!} = \frac{\left\{ \sum_{i=1}^{m} l_i \right\}_q!}{\left\{ l_1 \right\}_q \cdot \left\{ l_2 \right\}_q \cdots \left\{ l_m \right\}_q!} = \left( \sum_{i=1}^{m} l_i \right)_{q=q} \left( l_1, l_2, \ldots, l_m \right)_{q},$$

as required.
Theorem 3.7. An inversion poset for a multiset $I_N$ is self–dual.

Proof. Let $F : I_N \to I_N^*$ be given by

$$F((p_1, p_2, \ldots, p_m)) = (p_m, p_{m-1}, \ldots, p_2, p_1).$$

Let $a$ and $b$ be two permutations of $I_N$ such that $F(a) = F(b)$. Elements $a$ and $b$ must be the same, since their reverses are the same and therefore $F$ is one-to-one.

Let $a \leq b$, thus $b$ can be constructed from $a$ by some number of swaps of consecutive elements where the element on the right is greater than the element on the left. Let $C = (a, c_1, c_2, \ldots, c_m, b)$ be a saturated chain between $a$ and $b$. Since $c_m < b$, there has to be a pair of consecutive elements in $b$ where the element on the right is less than the element on the left. Thus, we know that $F(b) \leq F(c_m)$ since $F$ reverses the order of $b$ and $c_m$. Similarly, we can show that $F(c_m) \leq F(c_{m-1}) \leq \cdots \leq F(c_1) \leq F(a)$ and therefore $F(b) \leq F(a)$. 

The next theorem introduces another isomorphism between inversion posets which is convenient when a SSCD is being found.

Theorem 3.8. The inversion poset $I_{\{n_1, n_2, \ldots, n_k\}}$ is isomorphic to the dual of the inversion poset $I_{\{n_k, n_{k-1}, \ldots, n_1\}}$.

Proof. For the sake of simplicity let $\overline{n_i} = n_{k-i+1}$ and $m = \sum_{i=1}^{k} l_i$. Let $F : I_{N_1} \to I_{N_2}$ be a function given by

$$F((p_1, p_2, \ldots, p_m)) = (\overline{p_1}, \overline{p_2}, \ldots, \overline{p_m}),$$

where $N_1 = \{n_{1}, n_{2}, \ldots, n_{k} \} \ldots n_{k, n_{k}}$ and $N_2 = \{n_{k}, n_{k-1}, \ldots, n_{1} \} \ldots n_{1, n_{1}}$. Consider two permutations $a$ and $b$ of $I_{N_1}$ such that $F(a) = F(b)$. It is clear from the definition of $F$ that since $F(a) = F(b)$ then $a = b$ and therefore $F$ is one-to-one.
(a) The dual of the inversion poset $I_{\{1,1,2,3\}}$. 

(b) The inversion poset $I_{\{1,2,3,3\}}$.

Figure 3.2: An illustration of an isomorphism between $I_{\{1,1,2,3\}}$ and $I_{\{1,2,3,3\}}$.

To prove the isomorphism we have to show that $a \leq b$ if and only if $F(a) \geq F(b)$. Since we showed that $F$ is one-to-one it is enough to show that $a < b$ if and only if $F(a) > F(b)$. To show this, we will prove that $a < b$ if and only if $F(a) > F(b)$ because these statements are equivalent since there is always a saturated chain between $a$ and $b$ if $a < b$.

Let $a = (a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_m)$, where $a_i < a_{i+1}$ and suppose that $a < b$, with $b = (a_1, a_2, \ldots, a_{i+1}, a_i, \ldots, a_m)$.

$$F(b) = (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_{i+1}}, \overline{a_{i+2}}, \ldots, \overline{a_m})$$

Let’s focus on the pair $(\overline{a_{i+1}}, \overline{a_i})$. Since $a_i < a_{i+1}$ we can say that $n_{j_1} = a_i < a_{i+1} = n_{j_2}$, where $j_1 < j_2$. Then $\overline{n_{j_1}} = \overline{n_{j_2}} = n_{k-j_i+1}$ and $\overline{a_{i+1}} = \overline{n_{j_2}} = n_{k-j_2+1}$. It follows from the inequality $j_1 < j_2$ that $k - j_2 + 1 < k - j_1 + 1$, $n_{k-j_2+1} < n_{k-j_1+1}$ and hence $\overline{a_{i+1}} < \overline{a_i}$. Therefore, we can conclude that

$$F(b) = (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_{i+1}}, \overline{a_{i+2}}, \ldots, \overline{a_m}) \prec (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_{i+1}}, \ldots, \overline{a_m}) = F(a),$$

26
as required.

If we follow previous steps in the opposite direction we get that $F(a) \geq F(b)$ implies $a \leq b$, which completes the proof.

The previous theorem implies that $I_{\{1,2,3\}}$ is isomorphic to the dual of $I_{\{1,2,3,3\}}$. Additionally, the dual of the inversion poset $I_{\{1,2,3,3\}}$ is isomorphic to $I_{\{1,2,3,3\}}$, because the inversion poset is self–dual. Therefore, Theorems 3.7 and 3.8 yield the following corollary.

**Corollary 3.9.** The inversion poset $I_{\{n_1,\ldots,n_1,n_2,\ldots,n_2\ldots,n_k,\ldots,n_k\}}$ is isomorphic to the inversion poset $I_{\{n_1,\ldots,n_1,n_2,\ldots,n_2\ldots,n_k,\ldots,n_k\}}$. 

...
Chapter 4

The algorithm

We described, with a proof, an algorithm for finding a SSCD for a division poset in Chapter 3. We introduced Inverse posets and Young’s lattices but we didn’t indicate any decomposition for these posets. The reason for this is that there is no general proof that these posets have a SSCD. Stanley gave a combinatorial proof that a Young’s lattice, $L(m, n)$, is rank unimodal and has the Sperner property in [9]. Although, it was proven by Sylvester in 1878 that $L(m, n)$ is rank unimodal, Stanley gave the first combinatorial proof. He also conjectured that $L(m, n)$ has a SSCD for every positive $m$ and $n$. This conjecture has not yet been proven for a general case, only cases where $\min(m, n) \leq 4$ were proven [3].

We will introduce an enhanced algorithm for finding a SSCD for a general ranked poset $\mathcal{P}$ based on the previous work of Katsumata in [7]. The algorithm starts with $W_{\lfloor \rho(\mathcal{P})/2 \rfloor}$ chains located in the middle of $\mathcal{P}$ since the center of every symmetric saturated chain is located in the middle layer(s). It will create chains of length 0 or 1 based on the number of layers and then it will augment some of these chains in a way that they will remain symmetric and saturated.

Our algorithm differs significantly from the algorithm provided by Katsumata, since it uses a flow algorithm for the augmentation of chains and it can work with any poset.
given by its adjacency list, not just with an inversion poset $I_n$. If the program finds a
SSCD of the poset, it saves this decomposition into a text file and creates four pictures
of Hasse diagrams with highlighted chains. Pictures for selected posets are displayed in
appendix $B$.

4.1 Finding a flow in a graph

A directed graph can represent many structures from everyday life, like public trans-
portation, supply chains, road maps, computer networks or simplified water supply
system. Each of these structures can be represented as a system of pipes, connected
with joints, with a flow of an incompressible fluid. We will focus on a system with only
one point of entry and only one exit point for the fluid and we will try to investigate
what the maximal flow through the system is.

This abstraction gives rise to three constraints on the flow based on physics. There is
a limitation on how fact we can transport the fluid through a pipe based on the material
of the pipe, its radius and a viscosity of the fluid and other factors. This limitation is
called a capacity and each pipe can have a different capacity. The next constraint is
called antisymmetry and it means that a flow in a pipe is uniform – the flow is always
in only one direction. The last constraint is that fluid can leave the system only at one
joint and it can enter only at one joint and these joints have to be different. The exact
definitions follow.

Let a directed graph $G = (V, E)$ with a capacity function $c : V^2 \to \mathbb{R}^+_0$, where
$c(e) = 0$ if $e \notin E$, is given. Furthermore, denote sizes of $V$ and $E$ by $n$ and $m$,
respectively. A flow between two distinct vertices, source $s$ and sink $t$, is given by a
function $f : V^2 \to \mathbb{R}^+_0$ such that $\forall e \in E$:

$$f(e) \leq c(e) \quad \text{(capacity constraint)}, \quad (4.1)$$

$$f((u, v)) = -f((v, u)) \text{ where } (u, v) = e \quad \text{(antisymmetry constraint)}, \quad (4.2)$$
\[
\sum_{(u,v) \in V \setminus \{s,t\}} f((u,v)) = 0 \quad \text{(flow conservation constraint).} \quad (4.3)
\]

The value \(|f|\) is defined as the total flow into the sink, i.e. \(|f| = \sum_{u \in V} f((u,t))\). A maximum flow is a flow of maximum value. We define a residual capacity \(c_r(e)\) to be a difference between a capacity of an edge and a flow through an edge, \(c_r(e) = c(e) - f(e)\). An edge is saturated if its residual capacity is zero.

Many algorithms are available for finding a maximum flow in a graph [5]. We will focus on two types of flow algorithms, on the Ford–Fulkerson algorithm and the Push–Relabel algorithm.

**The Ford–Fulkerson algorithm**

This algorithm was first published in 1956 and improved versions of this algorithm were published by Dinic in 1970 and by Edmonds and Karp in 1972 [5]. This algorithm works as follows:

1. \(f((u,v)) \leftarrow 0\) for every pair of vertices in \(V\)

2. while there exist a path \(p\) between \(s\) and \(t\) such that \(c_r(e) > 0\) for every edge in this path \(p:\)

   (a) denote the smallest residual capacity in \(p\) by \(r\)

   (b) for each edge \((u,v)\) in \(p:\)

       i. \(f((u,v)) \leftarrow f((u,v)) + r\)

       ii. \(f((v,u)) \leftarrow f((v,u)) - r\)

The Ford–Fulkerson algorithm finds a path through unsaturated edges in each iteration and it adds this path to the flow. This path is frequently called an augmenting path and the performance of this algorithm depends on an algorithm used for finding this path. Finding a path in a graph is generally implemented as a depth–first or breadth–first search on vertices. Both of these methods have their advantages. The Ford–Fulkerson
algorithm has a running time of $O(n^2 m)$ or $O(nm^2)$ depending on whether a breadth-first or a depth-first search is used, respectively [6].

Using the Ford–Fulkerson algorithm, we have to find a new path between $s$ and $t$ in every cycle. This disadvantage is addressed in the next maximum flow algorithm.

The Push–Relabel algorithm

This algorithm does not find only one path from the source to the sink as the Ford–Fulkerson algorithm, but it pushes through the graph as much flow as is possible. For this purpose it manages a preflow and a vertex labeling $d$. Preflow is a function from $V^2$ to $\mathbb{R}$ which satisfies equations (4.1) and (4.2) for every edge in the graph. Preflow does not have to conserve the flow, it allows an incoming flow into a vertex to be bigger than an outgoing flow from the vertex. The algorithm starts with a preflow and it improves it into a maximal flow. Any valid preflow can be used, but commonly a preflow $f_p$ where $f_p((s,p)) = c((s,p))$ for every $(s,p) \in E$, where $s$ is the source, and $f_p(e) = 0$ for other edges is used.

After an initialization the algorithm either pushes a flow through a vertex or it updates the labeling. This vertex labeling provides estimates for a distance from the source and a distance to the sink for each vertex. The first implementation of this algorithm, by Goldberg in 1985, had a running time of $O(n^3)$ [6]. One year later, Goldberg and Tarjan published an implementation with a running time of $O(nm \log(n^2/m))$ [5].

4.2 Algorithm description

Our algorithm, similar to an algorithm of Katsumata, is implemented in three stages. These stages are making a poset, finding a SSCD and arrangement of chains.

1. Firstly, we create the Hasse diagram of a given poset or we read the poset structure from files. We represent the poset as a directed graph, e.g. as a list of vertices and an adjacency list, $V$ and $E$, respectively. This graph can be partitioned into
levels since we are representing the Hasse diagram and we store these levels in another list to improve the performance. Denote the rank of the poset by \( k \) and let \( N = \{N_0, N_1, \ldots, N_k\} \) be the collection of level sets. Additionally, we will create an empty list \( C \) for storing a SSCD.

When the first stage is finished, the list \( N \) has \( k+1 \) elements in it: \( N_0, N_1, \ldots, N_k \). Each of these lists contains vertices with the same rank.

2. If \( k \) is odd, there are two middle levels in the poset, we have to find a perfect matching between them. This matching yields the two middle elements from each chain and it can be found by making a graph \( H \). This graph contains vertices from \( N_{(k-1)/2}, N_{(k+1)/2} \) and two other vertices \( s \) and \( t \). We add all edges between \( N_{(k-1)/2} \) and \( N_{(k+1)/2} \) from \( E \) to \( H \), connect \( s \) with every vertex from \( N_{(k-1)/2} \) and connect each vertex from \( N_{(k+1)/2} \) to \( t \). Finding a maximum flow in \( H \) between \( s \) and \( t \), where the capacity of every edge is 1, yields the matching. If the flow value is equal to \( |N_{(k-1)/2}| \), we erase edges with no flow in \( H \), all incidence edges of \( s \), \( t \) and the vertices \( s \) and \( t \). Edges which are left are saved into \( C \). If the value is smaller, the poset does not have a SSCD.

If \( k \) is even, we fill \( C \) with vertices from \( N_{k/2} \).

For \( i = \lfloor k/2 \rfloor, \lfloor k/2 \rfloor - 1, \lfloor k/2 \rfloor - 2, \ldots, 1 \) perform following:

(a) Make a graph \( H \) which contains vertices from \( N_{i-1}, N_i, N_{k-i}, N_{k-(i-1)} \) and two extra vertices \( s \) and \( t \). Connect \( s \) with each vertex in \( N_{i-1} \) and each vertex in \( N_{k-(i-1)} \) with \( t \). If there is an edge in \( E \) between vertices from \( N_{i-1} \) and \( N_i \), connect the corresponding vertices in \( H \). Similarly, for layers \( N_{k-i} \) and \( N_{k-(i-1)} \). Layers \( N_i \) and \( N_{k-i} \) are connected using chains in \( C \), each chain which starts in \( N_i \) and ends in \( N_{k-i} \) will be represented as an edge in \( H \).

(b) Find a maximum flow between \( s \) and \( t \), where the capacity of every edge
is one. If the value of the flow is equal to $|N_i|$, we remove all edges without any flow, vertices $s$ and $t$ and their adjacent edges. The next step is extending the chains in $C$, which correspond to the edges between $N_i$ and $N_{k-i}$, with the flow. The list $C$ contains now a SSCD for subgraph induced by $N_{i-1}, N_i, N_{i+1}, \ldots, N_{k-(i-1)}$ levels of $G$. If the value is smaller, the poset does not have to have a SSCD.

3. A symmetric saturated chain decomposition is stored in $C$.

Let us follow steps of this algorithm on $I_4$. We start by creating the Hasse diagram of the poset and dividing its vertices into seven level sets: $N_0, N_1, \ldots, N_6$ as shown below.

In the next step we save $N_3$ into $C$ and set $i = 3$ since the rank of $I_4$ is 6. A graph $H$ contains vertices from levels $N_2, N_3, N_4$ and $N_5$ of the original graph. The edges between $N_2$ and $N_3$ are determined by the original graph, likewise the edges between $N_3$ and $N_4$. Additionally, corresponding vertices in $N_3$ and $N_4$ are connected by an edge. The next picture contains a diagram of $H$ with a maximum flow for $i = 3$. The edges with a nonzero flow are denoted by a thicker line.
When we find a maximum flow, we remove all edges without any flow and vertices $s$ and $t$. Remaining chains are used for extending chains in $C$. The list $C$ is shown beneath.

In the next iteration, when $i = 2$, the graph $H$ will contain levels $N_1, N_2, N_4$ and $N_5$. Additional vertices $s$ and $t$ are connected to $N_1$ and $N_5$, respectively. The diagram of $H$ with a maximum flow is shown below.
Removing unsaturated edges and vertices s and t from H yields three saturated chains of length 3. These three chains are used to extend some of the existing chains in C. The list C after the extension is illustrated below.

In the last iteration, the graph H contains levels N₀, N₁, N₅ and N₆ since \( i = 1 \).

A maximum flow in G yields only one chain because \( N₀ \) and \( N₆ \) contain only one vertex. Therefore, we expand only one chain in C, this is the longest chain in \( I₄ \).

When the last chain is extended, C will contain a SSCD of \( I₄ \).

The running time of this algorithm is polynomial, since it does not include backtracking. For this reason, the algorithm might not find a SSCD even if a SSCD exists.
If the algorithm finds a maximal flow, with its value smaller than the number of neighbors of $s$ or $t$, it fails to find a SSCD. The algorithm has a running time of $O(n^3m)$ or $O(n^2m^2)$, using the Ford–Fulkerson algorithm, depending on whether a breadth–first or depth–first search is used. The running time is $O(n^2m \log(n^2/m))$ if the Push–Relabel algorithm is used.

**Anomalies**

Some posets do not have a SSCD. If we run the described algorithm on these posets we should be able to recognize this. One of the first steps of the algorithm, if we run it on the poset shown in Figure 1.1(d), is constructing the graph in Figure 4.1 and finding a maximal flow. Two maximal flows are highlighted in this picture and as we can see, there is always one edge adjacent to $s$ or $t$ with no flow. The algorithm continues with removing $s$, $t$ and edges without any flow, this step will continue without any problems. But contracting all remaining edges will not create symmetric saturated chains and the algorithm fails. Actually, the algorithm fails on this poset every time; there is at least one edge adjacent to $s$ or $t$ with no flow and our implementations of this algorithm contains such a check. This behavior is needed, since there is no SSCD of this poset.

As mentioned before, the described algorithm does not include backtracking. The algorithm processes a poset from middle layer(s) outward and it does not return to already processed layers. This is the main reason why the running time is polynomial,
but on the other hand this may cause additional problems. While a SSCD contains a maximal flow between layers which are symmetric to the middle layer(s), not every maximal flow between these layers is extendable to a SSCD. There are two choices for a maximal flow between the middle layers for the poset shown in Figure 4.2. If the algorithm picks the maximal flow which contains the edge (5, 9), then it fails to find a SSCD because there will be always two unconnected vertices as in Figure 4.2(a). On the other hand if we choose the second maximal flow between middle layers, we would be able to extend this chains into a SSCD, this is shown in Figure 4.2(b).

There is no simple solution for this problem, the algorithm would have to contain backtracking to solve this and the total running time of such an algorithm would be significantly larger. Fortunately, we were able to find a SSCD for all important posets we ran this algorithm on, even with this imperfection. This was surprising and led us to additional statistics described in the next chapter focused on maximal chains in a poset.
Chapter 5

Results

Two computer programs, which are listed in appendix C, were written using Python and C++. Both of these programs use advanced graph libraries – NetworkX and PyGraphviz in Python and Boost libraries in C++. Both programs can find a SSCD for an inversion poset given its multiset or for a Young’s lattice given its two constants $m$ and $n$. The program written in C++ can additionally find a SSCD for a Boolean algebra and it can read a poset from files. To make sure that our programs provide correct results we checked some decompositions by another program. This additional program checked that the decomposition contained each element of a posed exactly once and that the chains were valid and symmetric.

While the Boost libraries implement the Push–Relabel algorithm, the NetworkX library in Python does not implement this algorithm. Unfortunately, NetworkX implements only the Ford–Fulkerson algorithm with a depth–first search approach. The running time of this algorithm allowed us to find a SSCD, in a reasonable time, for all inversion posets where the multiset contained less than 11 elements. This was a big improvement from the work of Katsumata but we were able to successfully run this program on an inversion poset with 11 distinct elements after improving the Ford–Fulkerson algorithm.
We observed that the performance of the Ford–Fulkerson algorithm with the depth–first search varies significantly. The algorithm is fast at the beginning but it slows down rapidly. On the other hand, the breadth–first search is slower than the depth–first search for paths at the beginning, but its performance does not change over time. It seems that the best solution can be achieved by combining these two approaches. Our implementation of the Ford–Fulkerson algorithm uses the depth–first search first and it switches to the breadth–first search when the breadth–first search is faster.

Additionally, we managed to improve the breadth–first search as well. The regular breadth–first search from a source in a graph would visit all neighbors of the source first, then it would visit neighbors of those neighbors, etc. Our improved implementation of the breadth–first search randomly chooses some of source’s neighbors and performs full breadth–first search on them. If it will not find a path from the source to the sink it continues with the next part of nonvisited neighbors of the source until it finds a path from the source to the sink. We achieved the best performance when the algorithm processed 2% of source’s neighbors at a time.

The implementation in C++ does not need this improvement because Boost libraries contain an implementation of the Push–Relabel algorithm. This flow algorithm is faster than our improved Ford–Fulkerson algorithm. Since this program is faster than implementation in Python, we extended the input options for it. It can find a SSCD for inversion posets, Young’s lattices, Boolean algebras and it can read a poset from files with its structure.

As mentioned in Chapter 3, we can use Corollary 3.9 when testing all inversion posets. Suppose that we would like to find a SSCD for all inversion posets where the multiset contains 4 distinct elements and its length is 6. Let us denote the elements as 1, 2, 3 and 4, where the order is naturally 1 < 2 < 3 < 4. There are 10 different possible multisets. This means that we would have to run the program ten times, but using Corollary 3.9 we can reduce the number of instances to 6 because some of these multisets will create isomorphic posets. A list with these combinations of multisets follows.
<table>
<thead>
<tr>
<th>Poset name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean algebra $B_n$</td>
<td>$n \leq 25$</td>
</tr>
<tr>
<td>Inversion poset for a multiset $I_N$</td>
<td>$</td>
</tr>
<tr>
<td>Young’s lattice $L(m, n)$</td>
<td>$m + n \leq 30$</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of posets with the maximal size for which a symmetric saturated chain decomposition was found by our program.

1. $\{1, 1, 1, 2, 3, 4\}, \{1, 2, 3, 4, 4, 4\}$
2. $\{1, 2, 2, 2, 3, 4\}, \{1, 2, 3, 3, 3, 4\}$
3. $\{1, 1, 2, 2, 3, 4\}, \{1, 2, 3, 3, 4\}$
4. $\{1, 1, 2, 3, 3, 4\}, \{1, 2, 2, 3, 4, 4\}$
5. $\{1, 1, 2, 3, 4, 4\}$
6. $\{1, 2, 2, 3, 3, 4\}$

Each multiset in one row will yield the same poset, up to isomorphism. There are 2047 multisets with less than 12 elements, but only 1086 of them are different up to isomorphism.

As mentioned before, the program written in C++ is faster than the program in Python and therefore all our tests were performed with this program. Table 5.1 contains all posets we tested and for which we found a SSCD. It is surprising that the program never failed and always found a SSCD for all tested inversion posets, Young’s lattices and Boolean algebras since we provided a poset for which our program fails. It may be possible to extend these boundaries a little bit, but this depends on the computer which runs the program.
Table 5.2: Number of different longest chains in inversion posets $I_n$, where $4 \leq n \leq 10$.

### 5.1 Future work

There are several ways this work might be extended. From a computational point of view, the main reason we could not run our programs on bigger posets was the amount of memory needed. Our implementations construct a poset and keep it as list of vertices and list of edges in the memory. However, because we have to keep only three layers of the poset in memory, the whole poset can be saved in external memory and loaded into internal memory only when needed. This approach, when using a compression, should be investigated in the future.

The memory issue may have another solution. Distributing the vertex and edge lists over more computers should allow to improve the upper bounds we set in this paper. Distributed flow algorithms are available, although the required communication can be a new limiting factor.

We modified our program to count the number of maximal chains in inversion posets $I_n$ and to find a SSCD given a maximal chain. The number of maximal chains corresponds to the number of different standard Young tableaux of shape $(n-1, n-2, \ldots, 2, 1)$ and it can be found by the Hook formula \[ \cdot \]. This is another piece of evidence that
our program performs correctly. The program did not find a SSCD for some maximal chains; we call these chains bad chains. Table 5.2 shows how many maximal chains \( I_n \) has and for small \( n \), how many bad chains \( I_n \) has. For example, in \( I_4 \), 4 of the 16 maximal chains cannot be extended to a SSCD by our algorithm. Since our algorithm can fail to find a SSCD, table 5.2 provides only upper bounds on the number of bad chains. However, we checked that the exact number of bad chains for \( I_4 \) is 4.

Another way to find a SSCD can be by finding symmetric chains from the longest to the shortest. This approach itself may not perform well, but it may be combined with the approach used in our algorithm, since it seems that the majority of the longest chains will not make the rest of the poset undecomposable into symmetric saturated chains.

From a mathematical point of view, Stanley’s conjecture that there is a SSCD for any Young’s lattice as well as for any inversion poset is still open. However, some progress has been made in determining the Sperner property and other combinatorial properties of these and similar posets [3].
References


Appendix A

Program interface

We explain how to compile and run programs we developed and what each input parameter means in this chapter. In our description we will assume that we work on a Debian-based Linux machine.

A.1 Python

To successfully run this program we need to have files `diagram.py` and `main.py` inside a folder and install two additional Python libraries, NetworkX and PyGraphviz. The program can be called from a terminal by using this command

```sh
$ python main.py [−h] [−−inversion 1 2 3 4 . . .] [−−young m n] [−−no_pict]
```

The optional arguments are

- `−−h` or `−−help`
  - This argument prints a help message and exits the program.

- `−−inversion`
  - This argument starts the program for an inversion poset and a name of the minimal vertex with the smallest rank follows as a list of numbers, each
separated by a space. For example a SSCD for $I_5$ can be found by this command

\[ \$ \text{python main.py --inversion 1 2 3 4 5} \]

- **--young**
  
  Two numbers, $m$ and $n$, separated by a space follow this argument. The program tries to find a SSCD for a Young's lattice $L(m,n)$. For example a SSCD for $L(5,5)$ can be found by this command

\[ \$ \text{python main.py --young 5 5} \]

- **--no_pict**
  
  The program automatically stores the decomposition in a text file and it creates pictures of the Hasse diagram of the poset with the decomposition highlighted. When using this argument, no pictures will be created. Generating pictures for big posets like $I_{10}$ is time consuming and the usability of such pictures is questionable.

If the program finds a SSCD it saves it together with its output into a text file and creates four pictures, depending on the **--no_pict** argument.

- **.FULL.png** Contains all vertices with labels. Each level is ordered exactly as it is saved in the memory.

- **.FULL_NO_LABELS.png** Contains all vertices without labels. Each level is ordered exactly as it is saved in the memory.

- **.SPARSE.png** Contains all vertices with labels. Each level is ordered such that the final graph has less intersecting edges.
“...SPARCE_NO_LABELS.png” Contains all vertices without labels. Each level is ordered such that the final graph has less intersecting edges.

A.2 C++

To be able to compile the program we need to install Boost libraries. Optionally, we can install the OpenMP library if we have more CPUs available. The program can be compiled by this command

```
$ g++ -o main main.cpp node.cpp node.h -lboost_program_options
  -lboost_serialization -lboost_system -lboost_filesystem
  -I/usr/include/python2.7 -lpython2.7 -fopenmp -O3
```

assuming that Python 2.7 is installed on the system. This will generate an executable file named “main” which contains our program. The program can be called by using this command

```
$ ./main [--bool n] [--file name] [--inversion 1 2 3 4 ...] [--young m n]
  [--no_pict] [--help]
```

The program in C++ shares some arguments with the program in Python and their effects are the same. These arguments are --inversion, --young, --no_pict and --help. The rest of arguments are described below.

- --bool
  - This argument together with a number $n$ will find a SSCD for a Boolean algebra $B_n$. The program call for $B_6$ is
    
    ```
    $ ./main --bool 6
    ```

- --file
  
  A string (name) follows this argument. The current folder has to contain two files, “name_levels” and “name_nodes”. As their names may suggest,
“name_levels” contains nodes separated into layers and “name_nodes” contains neighbors of each vertex corresponding to the line number. Vertices have to be identified by sequential numbers starting from zero. For example the poset, for which our algorithm fails, from Figure 4.2 can be saved into two files:

**my_levels:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1,2,3</td>
</tr>
<tr>
<td>2</td>
<td>4,5,6,7</td>
</tr>
<tr>
<td>3</td>
<td>8,9,10,11</td>
</tr>
<tr>
<td>4</td>
<td>12,13,14</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

**my_nodes:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,2,3</td>
</tr>
<tr>
<td>1</td>
<td>4,5,6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9,10</td>
</tr>
<tr>
<td>6</td>
<td>10,9</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>13,14</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

As we see, the last vertex has to be connected to itself. The algorithm will fail to find a SSCD for this configuration, but the poset has a SSCD.
If we change the sixth line of my\_nodes to “9,10” instead of “10,9” the algorithm will find the SSCD because it will find a maximal flow which is shown in Figure 4.2(b). The program can be called by

```
$ ./main --file my
```

Additionally, all pictures are generated by the Python class diagram.py and therefore this class has to be in the same folder as the executed main file if the argument --no\_pict is not used. If the argument --no\_pict is used, then program generate the same pictures as the program written in Python.
Appendix B

Performance of programs and generated figures

We provide running times for our programs in selected posets in Table B.1 and Figures B.1 and B.2. Note that these times do not include the time needed to generate the poset in memory.

<table>
<thead>
<tr>
<th>Program version</th>
<th>Time needed to find a SSCD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>C++ Inversion poset, $I_n$</td>
<td>1 s</td>
</tr>
<tr>
<td>Python, Ford–Fulkerson from NetworkX</td>
<td>1 s</td>
</tr>
<tr>
<td>Python, improved Ford–Fulkerson</td>
<td>1 s</td>
</tr>
<tr>
<td>Young’s lattice, $L(m,n)$</td>
<td></td>
</tr>
<tr>
<td>$(m,n) = (7,7)$</td>
<td>(7,7)</td>
</tr>
<tr>
<td>C++</td>
<td>1 s</td>
</tr>
<tr>
<td>Python, Ford–Fulkerson from NetworkX</td>
<td>2.1 s</td>
</tr>
<tr>
<td>Python, improved Ford–Fulkerson</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

Table B.1: Performance of each program on selected posets.

When we focus on the graphs which correspond to the program written in C++, it appears that the last three points on each graph form a line. Since the axes of both graphs have a logarithmic scale, we can estimate the program complexity using the
slopes of these lines. The slope for the line which approximates the performance for inversion posets is 1.07; this implies that the running time is $O(n^{1.07})$. Similarly, the slope of the line which approximates the last three points for Young’s lattices is 1.14 which corresponds to $O\left((\frac{m+n}{n})^{1.14}\right)$. Using these estimations we can conclude that if we had a computer with enough memory, if the program did not fail, we would need 9.2 and 22.2 hours to find a SSCD for $I_{12}$ and $L(16,16)$, respectively.

Figure B.1: Performance of programs on inversion posets.

Figure B.2: Performance of programs on Young’s lattices.
Selected figures generated by the program are displayed below and on the following pages.

Figure B.3: Inversion poset, $I_4$
Figure B.8: Boolean algebra, $B_3$
Figure B.9: Boolean algebra, $B_3$
Figure B.11: B₅: 5-SPARSE.png
Figure B.12: Young’s lattice, \( L(4, 2) \)
Figure B.13: Young’s lattice, $L(2, 4)$

(a) 2,2,2,2_FULL.png

(b) 2,2,2,2_SPARSE.png

(c) 2,2,2,2_FULL_NO_LABELS.png

(d) 2,2,2,2_SPARSE_NO_LABELS.png
Appendix C

Source codes

The purpose of this section is to provide two implementations, in Python and in C++, of the algorithm given in the text.

C.1 Python

To run the Python implementation we need two external libraries NetworkX and PyGraphviz. The Ford–Fulkerson algorithm is implemented in the NetworkX library and the PyGraphviz library creates figures of the poset and its symmetric saturated chain decomposition.

The first script tries to find a SSCD. If this script finds a SSCD, it saves the decomposition into a text file and calls the second script (diagram.py) to generate figures of the poset and its decomposition. The first script accompanied with comments is listed below.

main.py:

```python
#!/usr/bin/python

# list of libraries
from array import array
import networkx as nx
import itertools
```
```python
import time
import collections
import sys
import math
import random
import argparse
import diagram

global gCount

class Node:
    # each node has 4 properties
    # name: name of the node
    # level: the number of the level
    # neighbors: list of neighbors in the next level
    # nameOfMatchingNode: name of the matched node
    def __init__(self, name, level):
        self.name = name
        self.level = level
        self.neighbors = list()
        self.nameOfMatchingNode = name
    # checks if there is possible inversion of its elements
    def isLast(self):
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                return False
        return True
    # generate next inversion
    def generateNext(self, dictOfNextLevel):
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                tmpName = list(self.name)
                tmp = self.name[i]
                tmpName[i] = tmpName[i+1]
                tmpName[i+1] = tmp
                self.neighbors.append(tuple(tmpName))
                if ~dictOfNextLevel.has_key(tuple(tmpName)):
                    dictOfNextLevel[tuple(tmpName)] = Node(tmpName, self.level+1)
    @property
    def Name(self):
        return tuple(self.name)
    @property
    def NameOfMatchingNode(self):
        return tuple(self.nameOfMatchingNode)

# logger for easier handling with output
def loggerOut(*message):
    msg = ''
    for m in message:
        msg = ''.join([msg, str(m)])
    print msg
    f.write(''.join([msg, '\n']))
sys.stdout.flush()

# constructing an Inverse poset given a multiset
def makeGraphOfInversePartiallyOrderedMultiSet(rootList):
    listOfLevels = [dict(), dict()]
    root = Node(rootList, 0)
```
listOfLevels[0][tuple(root.name)] = root
root.generateNext(listOfLevels[1])
level = 1
while (len(listOfLevels[level]) > 1) or (not listOfLevels[level].values()[0].isLast()):
    listOfLevels.append(dict())
d = listOfLevels[level]
for node in d.itervalues():
    node.generateNext(listOfLevels[level+1])
level += 1
return listOfLevels

## constructing a Young lattice given its size

```
def makeGraphOfYoungsLattice(m, n):
    listOfLevels = [dict() for x in xrange(m*n+1)]
    tmp = [0 for i in xrange(n)]
    root = Node(tmp, 0)
    listOfLevels[0][root.Name] = root
    for i in xrange(len(listOfLevels) - 1):
        for node in listOfLevels[i].values():
            if sum(node.Name) < m*n:
                for ii in xrange(n):
                    if (node.Name[ii] < m) and (ii == n-1) or (node.Name[ii] < node.Name[ii+1]):
                        new = list(node.name)
                        new[ii] += 1
                        newNode = Node(new, i+1)
                        if listOfLevels[i+1].has_key(newNode.Name) == True:
                            node.neighbors.append(listOfLevels[i+1][newNode.Name].Name)
                        else:
                            node.neighbors.append(newNode.Name)
                        listOfLevels[i+1][newNode.Name] = newNode
    return listOfLevels
```

## Expanding the graphs for additional layers.

```
def expandGraph(graph, middleLayer, listOfLevels, intActualLayer):
    if (len(graph) == 1) and (intActualLayer == -1): return
    graph.clear()
    for node in listOfLevels[intActualLayer].values():
        graph.add_edges_from([(node.NameOfMatchingNode, neighbor.NameOfMatchingNode) for neighbor in node.neighbors], capacity=1.0)
    for node in listOfLevels[len(listOfLevels) - intActualLayer - 2].values():
        graph.add_edges_from([(node.NameOfMatchingNode, neighbor.Name) for neighbor in node.neighbors], capacity=1.0)
```

## Adding 'source' and 'sink' nodes for a flow algorithm.

```
def addEndNodes(graph, bottomLevel, upperLevel):
    graph.add_edges_from([('rootDown', node) for node in bottomLevel], capacity=1.0)
    graph.add_edges_from([('node', 'rootUp') for node in upperLevel], capacity=1.0)
```

## Removing edges without any flow.

```
def removeExtraEdges(graph, flow, middleLayer):
    graph.remove_nodes_from([('rootUp', 'rootDown')])
    for nodeTmp in graph.successors((node.NameOfMatchingNode, 'u')):
        if flow[(node.NameOfMatchingNode, 'u')] == 0:
            graph.remove_edges_from([(nodeTmp.Name, 'u')])
```
for nodeTmp in graph.predecessors((node.NameOfMatchingNode, 'd')):
    if flow[nodeTmp][(node.NameOfMatchingNode, 'd')] == 0:
        graph.remove_edge(nodeTmp,(node.NameOfMatchingNode, 'd'))

# Contracting edges with a flow − expanding a list with the SSCD.
def contractEdges(graph, bottomLayer, middleLayer, upperLayer):
    graphNew = nx.DiGraph()
    for node in middleLayer.values():
        if len(graph.successors((node.NameOfMatchingNode, 'u'))) > 0:
            graph.add_path((graph.predecessors((node.NameOfMatchingNode, 'd'))[0], node.NameOfMatchingNode, graph.successors((node.NameOfMatchingNode, 'u'))[0]))
        else:
            graph.add_node(node.NameOfMatchingNode)
            for node in bottomLayer.keys():
                neigh = graph.neighbors(node)[0]
                graph.remove_nodes_from(((node.NameOfMatchingNode, 'u'), node))
                nodeNew = (node, neigh, graph.neighbors(neigh)[0])
                graph.remove_nodes_from((nodeNew[1:3]))
                graphNew.add_node(nodeNew[2])
            upperLayer[nodeNew[0]].nameOfMatchingNode = nodeNew
        return graphNew, graph.nodes()

result = None

# Finding a path between source and sink using depth − first approach.
def find_pathDepthFirst(graph, visitedNodes, source, sink, path):
    result = None
    visitedNodes.append(source)
    if source == sink:
        return path
    for successor in graph.successors(source):
        residual = graph.edge[source][successor]['capacity'] - graph.edge[source][successor]['flow']
        if residual > 0 and not successor in visitedNodes:
            if not successor in path:
                result = find_pathDepthFirst(graph, visitedNodes, successor, sink, path + [successor])
            if result != None:
                return result
    for predecessor in graph.predecessors(source):
        residual = graph.edge[predecessor][source]['flow']
        if residual > 0 and not predecessor in visitedNodes:
            if not predecessor in path:
                result = find_pathDepthFirst(graph, visitedNodes, predecessor, sink, path + [predecessor])
            if result != None:
                return result

# Finding a path between source and sink using enhanced breadth − first approach.
This function process nodes attached to the source in blocks. Each block contain certain percentage of neighbors to source. The breadth − first search then continue with those nodes. If no path to the sink is found, then the function investigate next block of neighbors.
def find_pathBreadthFirst(graph, sink, actualLayer, layers, percentage):
if percentage == 0:
    raise Exception('Percentage cannot be 0!')
visitedNodes = dict()
## erase visited tokens in graph at the beginning
if actualLayer == 0:
    if not layers.has_key(1):
        layers[1] = list()
    for source, path in layers[0]:
        visitedNodes[source] = True
        for successor in graph.successors(source):
            residual = graph.edge[source][successor][\'capacity\'] - graph.edge[source][successor][\'flow\']
            if (residual > 0) and (not successor in path) and (not visitedNodes.has_key(successor)):
                layers[successor] = True
                layers[1].append([successor, path+[successor]])
        actualLayer += 1
## change from percentage to a number of nodes
if (actualLayer == 1) and (percentage < 1):
    percentage = math.ceil(len(layers[1]) * percentage)
while actualLayer > 0:
    newNodes = False
    if not layers.has_key(actualLayer+1):
        layers[actualLayer+1] = list()
    ## neighbors of the source node
    if actualLayer == 1:
        for i in xrange(int(percentage)):
            if len(layers[actualLayer]) == 0:
                break
            item = layers[actualLayer].pop(random.choice(xrange(len(layers[actualLayer]))))
            source = item[0]
            path = item[1]
            if source == sink:
                return path
        for successor in graph.successors(source):
            residual = graph.edge[source][successor][\'capacity\'] - graph.edge[source][successor][\'flow\']
            if residual > 0 and (not successor in path) and (not visitedNodes.has_key(successor)):
                layers[successor] = True
                layers[actualLayer+1].append([successor, path+[successor]])
                visitedNodes[successor] = True
                newNodes = True
    else:
        for source, path in layers[actualLayer]:
            if source == sink:
                return path
        for successor in graph.successors(source):
residual = graph.edge[source][successor][’capacity’] - graph.
    edge[source][successor][’flow’]
    if residual > 0 and (not successor in path) and (not
    visitedNodes.has_key(successor)):
        layers[actualLayer+1].append((successor, path+[successor]))
        visitedNodes[successor] = True
    newNodes = True
    for predecessor in graph.predecessors(source):
        residual = graph.edge[predecessor][source][’flow’]
        if residual > 0 and (not predecessor in path) and (not
        visitedNodes.has_key(predecessor)):
            layers[actualLayer+1].append((predecessor, path+[predecessor]))
            visitedNodes[predecessor] = True
            newNodes = True
        if newNodes:
            actualLayer += 1
        else:
            actualLayer -= 1
    return None

### Ford–Fulkerson algorithm with adaptive switching between depth and breadth
search

def myFord_fulkerson(graph, source, sink):
    for x in graph.edge:
        for y in graph.edge[x]:
            graph.edge[x][y][’flow’] = 0
    path = find_pathDepthFirst(graph, [], source, sink, [source])
    tmpTime = time.time()
    tmp = find_pathBreadthFirst(graph, source, 0,
        {0:[[source],[source]]}, .02)
    timeBreadth = time.time() - tmpTime
    value = 0
    breadth = False
    while path != None:
        value += 1
        for i in xrange(1, len(path)):
            if graph.edge[path[i-1]][path[i]][’flow’] = 1
                graph.edge[path[i-1]][path[i]][’flow’] = 0
        if breadth:
            path = find_pathBreadthFirst(graph, sink, 0,
                {0:[[source],[source]]}, .02)
        else:
            start = time.time()
            path = find_pathDepthFirst(graph, [], source, sink, [source])
            if time.time() - start > timeBreadth*2:
                breadth = True
                loggerOut(’—Breadth-search activated!—’)
    flow = dict()
    for x, y in graph.edges():
        if flow.has_key(x):
            flow[x][y] = f
        else:
            flow[x] = y: f
    return value, flow
def extendAndFlow(graph, middleLayer, bottomLayer):
    for node in middleLayer.values():
        neighDown = graph.predecessors(node.NameOfMatchingNode)
        neighUp = graph.successors(node.NameOfMatchingNode)
        graph.remove_node(node.NameOfMatchingNode)
        graph.add_edge((node.NameOfMatchingNode, 'd'), (node.NameOfMatchingNode, 'u'), capacity=1.0)
        graph.add_edges_from([(x, (node.NameOfMatchingNode, 'u')) for x in neighUp], capacity=1.0)
        graph.add_edges_from([(x, (node.NameOfMatchingNode, 'd')) for x in neighDown], capacity=1.0)
        value, flow = myFordFulkerson(graph, 'rootDown', 'rootUp')
        return value, flow

# Main body of the program.
def main(f, listOfLevels):
    gCount = 0
    start = time.time()
    numberOfLayers = len(listOfLevels) - 1
    decomposition = list()
    graph = nx.DiGraph()
    middleLayer = list()
    if numberOfLayers % 2 == 0:
        # there is one layer in the middle
        loggerOut('---One layer in the middle---')
        intActualLayer = numberOfLayers / 2
        decomposition = list()
        graph = nx.DiGraph()
        middleLayer = list()
        loggerOut('---Middle part start---', (time.time() - start), "seconds.")
        else:
            # There are two middle layers -> find matching, create a graph with 3 layers
            loggerOut('---Two layers in the middle---')
            intActualLayer = numberOfLayers / 2
            for item in listOfLevels[intActualLayer].values():
                for itemNeighbor in item.neighbors:
                    tmpNode = listOfLevels[intActualLayer + 1].get(itemNeighbor)
                    graph.add_edge(item.Name, tmpNode.Name)
                    middleLayer.append((item.Name, tmpNode.Name))
                    loggerOut('---Middle part done---', (time.time() - start), "seconds.")
                    for item in matching.keys():
                        if listOfLevels[intActualLayer].has_key(item):
                            newNode = (item, matching[item])
                            listOfLevels[intActualLayer].get(item).nameOfMatchingNode = newNode
                            listOfelevens[intActualLayer + 1].get(matching[item]).nameOfMatchingNode = newNode
                            middleLayer.append(newNode)
                            intActualLayer += 1
                            tmp.add_nodes_from(middleLayer)
                            graph = tmp
                            return value, flow

# Contract the matching into symmetric chains.
for item in matching.keys():
    if listOfLevels[intActualLayer].has_key(item):
        newNode = (item, matching[item])
        listOfelevens[intActualLayer].get(item).nameOfMatchingNode = newNode
        listOfelevens[intActualLayer + 1].get(matching[item]).nameOfMatchingNode = newNode
        middleLayer.append(newNode)
        intActualLayer += 1
        tmp.add_nodes_from(middleLayer)
        graph = tmp
### Add bottom neighbors for a matching.
```python
for item in listOfLevels[intActualLayer].values():
    for itemNeighbor in item.neighbors:
        tmpNode = listOfLevels[intActualLayer + 1].get(itemNeighbor).NameOfMatchingNode
        graph.add_edge(item.Name, tmpNode)
```

### Add upper neighbors for a matching.
```python
for item in listOfLevels[numberOfLayers - intActualLayer - 1].values():
    for itemNeighbor in item.neighbors:
        tmpNode = listOfLevels[numberOfLayers - intActualLayer].get(itemNeighbor).Name
        graph.add_edge(item.NameOfMatchingNode, tmpNode)
```

```python
while (intActualLayer >= 0):
    loggerOut('--ActualLayer is ', intActualLayer, ' / ', numberOfLayers, '--', (time.time() - start), ' seconds. ')
    addEndNodes(graph, listOfLevels[intActualLayer].keys(), listOfLevels[numberOfLayers - intActualLayer].keys())
    loggerOut('--flows start --', (time.time() - start), ' seconds. ')
    value, flow = extendAndFlow(graph, listOfLevels[intActualLayer + 1], listOfLevels[intActualLayer].keys())
    if value < len(listOfLevels[intActualLayer]):
        raise Exception('NO matching')
    loggerOut('--flows end --', (time.time() - start), ' seconds. ')
    removeExtraEdges(graph, disconnectedNodes = contractEdges(graph, listOfLevels[intActualLayer], listOfLevels[intActualLayer + 1], listOfLevels[numberOfLayers - intActualLayer]))
    decomposition.extend(disconnectedNodes)
    intActualLayer -= 1
```

```python
logOut('--expandGraph start --', (time.time() - start), ' seconds. ')
expandGraph(graph, middleLayer, listOfLevels, intActualLayer)
logOut('--expandGraph done --', (time.time() - start), ' seconds. ')
```

```python
if type(i[0]) == tuple:
    msg = ''
    for m in i:
        msg = ' ' . join([msg, str(m)])
else:
    msg = str(i)
```

```python
loggerOut('Decomposition: ')
```

```python
### Processing user input, running the main function and generating graphs
parser = argparse.ArgumentParser(description='Program for finding a symmetric saturated chain decomposition, created as part of Master Thesis by Ondrej Zjevik at University of Minnesota. Usage: %prog [-h] [--inversion ...] [--young m n]')
parser.add_argument('--inversion', type=int, nargs='* ', help='element of Young s lattice', metavar='1234567... ', dest='root')
parser.add_argument('--young', type=int, nargs=2, help='constants of a Young s...')
parser.add_argument('--no_pict', dest='picture', default=True, const=False, action='store_const', metavar='', help='will not generate pictures')
args = parser.parse_args()
```
The second script written in Python generates pictures from an output from the first script or from the script written in C++. The content of the second script follows.

diagram.py:

```python
#!/usr/bin/python
from array import array
import pygraphviz as pgv
from random import randrange

color = True;

class Node:
    def __init__(self, name, level):
        self.name = name
        self.level = level
        self.neighbors = list()
        self.nameOfMatchingNode = name
    def isLast(self):
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                return False
        return True
    def generateNext(self, dictOfNextLevel, listOfNextLevel, graph):
        graph.add_node(self.Name)
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                tmpName = list(self.name)
                tmp = self.name[i]
                tmpName[i] = tmpName[i+1]
                tmpName[i+1] = tmp
                self.neighbors.append(tuple(tmpName))
                graph.add_edge(self.Name, tuple(tmpName), weight=1, style='solid')
                if not dictOfNextLevel.has_key(tuple(tmpName)):
                    tmpNode = Node(tmpName, self.level+1)
                    dictOfNextLevel[tuple(tmpName)] = tmpNode
                    listOfNextLevel.append(tmpNode)
                    graph.add_node(self.Name)
        #...
```

The second script written in Python generates pictures from an output from the first script or from the script written in C++. The content of the second script follows.
for i in xrange(1, size+1):
    if self.name.count(i) == 0:
        tmpName = list(self.name)
        tmpName.append(i)
        tmpName.sort()
        self.neighbors.append((tuple(tmpName),
            graph.add_edge(self.Name, tuple(tmpName),
                weight=1, style='solid'))
        if ~dictOfNextLevel.has_key(tuple(tmpName)):
            tmpNode = Node(tuple(tmpName), self.level+1)
            dictOfNextLevel[tuple(tmpName)] = tmpNode
            listOfNextLevel.append(tmpNode)

@property
def Name(self):
    return tuple(self.name)
@property
def NameOfMatchingNode(self):
    return tuple(self.nameOfMatchingNode)

def makeBool(graph, size):
    dictOfLevels = [dict(), dict()]
    listOfLevels = [list(), list()]
    root = Node(tuple(), 0)
    dictOfLevels[0][tuple(root.name)] = root
    listOfLevels[0].append(root)
    root.generateNextBool(dictOfLevels[1], listOfLevels[1], graph, size)
    level = 1
    while (len(dictOfLevels[level]) > 1):
        dictOfLevels.append(dict())
        listOfLevels.append(list())
        d = dictOfLevels[level]
        for node in d.itervalues():
            node.generateNextBool(dictOfLevels[level+1], listOfLevels[level+1],
                graph, size)
        level += 1
    sparseGraph = graph.copy()
    for level in xrange(1, len(listOfLevels)):
        for node in dictOfLevels[level-1].keys():
            for node2 in dictOfLevels[level].keys():
                if graph.has_edge(node2, node) == False:
                    graph.add_edge(node, node2, weight=1, style='invis')
    return sparseGraph

def makeInversePartiallyOrderedMultiSet(graph, root):
    dictOfLevels = [dict(), dict()]
    listOfLevels = [list(), list()]
    root = Node(root, 0)
    dictOfLevels[0][tuple(root.name)] = root
    listOfLevels[0].append(root)
    root.generateNext(dictOfLevels[1], listOfLevels[1], graph)
    level = 1
    while (len(dictOfLevels[level]) > 1) or (not dictOfLevels[level].values()[0].isLast()):
        dictOfLevels.append(dict())
        listOfLevels.append(list())
        d = dictOfLevels[level]
        for node in d.itervalues():
            node.generateNext(dictOfLevels[level+1], listOfLevels[level+1],
                graph)
        level += 1
    sparseGraph = graph.copy()
for level in xrange(1, len(listOfLevels)):
    for node in dictOfLevels[level-1].keys():
        for node2 in dictOfLevels[level].keys():
            if graph.has_edge(node2, node) == False:
                graph.add_edge(node, node2, weight=1, style='invis')
    return sparseGraph

def makeYoungsLattice(graph, m, n):
    dictOfLevels = [dict() for x in xrange(m*n+1)]
    tmp = [0 for i in xrange(n)]
    root = Node(tmp, 0)
    dictOfLevels[0][root.Name] = root
    for i in xrange(len(dictOfLevels)-1):
        for node in dictOfLevels[i].values():
            if sum(node.Name) < m*n:
                for ii in xrange(n):
                    if (node.Name[ii] < m) and (ii == n-1) or (node.Name[ii] < node.Name[ii+1]):
                        newNode = list(node.name)
                        newNode[ii] += 1
                        newNode = Node(newNode.Name, node.Name, weight=1, style='solid')
                        if dictOfLevels[i+1].has_key(newNode.Name) == True:
                            node.neighbors.append(dictOfLevels[i+1][newNode.Name].Name)
                        else:
                            node.neighbors.append(newNode.Name)
                        dictOfLevels[i+1][newNode.Name] = newNode
    sparseGraph = graph.copy()
    for level in xrange(1, m*n):
        for node in dictOfLevels[level-1].keys():
            for node2 in dictOfLevels[level].keys():
                if graph.has_edge(node2, node) == False:
                    graph.add_edge(node2, node, weight=1, style='invis')
    return sparseGraph

def makeFromFile(graph, fileName):
    listOfLevels = []
    f = open(fileName + "levels", 'r')
    level = -1;
    for line in f:
        line.strip()
        level += 1;
        listOfLevels.append(list())
    for node in line.split(', '):
        level, append(int(node))
    f = open(fileName + "nodes", 'r')
    nodeId = -1;
    for line in f:
        nodeId += 1;
        if nodeId > level: continue
        line.strip()
        for neighbor in line.split(', '):
            graph.add_edge(int(nodeId), int(neighbor), weight=1, style='solid')
    sparseGraph = graph.copy()
    for level in xrange(1, len(listOfLevels)):
```python
for node in listOfLevels[level-1]:
    for node2 in listOfLevels[level]:
        if graph.has_edge((node2,),(node,)) == False:
            graph.add_edge((node,),(node2,), weight=1, style='invis')
return sparseGraph

def makeGraphOfBool(oldFile, size):
    graph = pgv.AGraph(rankdir='BT')
sparseGraph = makeBool(graph, size)

decomp = False;
i = 0
d = dict()
f = open(oldFile.name, 'r')
generatePictures(f, graph, sparseGraph)

def makeGraphOfInversePartiallyOrderedMultiSet(oldFile, root):
    graph = pgv.AGraph(rankdir='BT')
sparseGraph = makeInversePartiallyOrderedMultiSet(graph, root)

decomp = False;
i = 0
d = dict()
f = open(oldFile.name, 'r')
generatePictures(f, graph, sparseGraph)

def makeGraphOfYoungsLattice(oldFile, m, n):
    f = open(oldFile.name, 'r')
    graph = pgv.AGraph()
    numberOfLayers = m*n
    sparseGraph = makeYoungsLattice(graph, m, n)
    generatePictures(f, graph, sparseGraph)

def makeGraphFromFile(decomposition, fileName):
    graph = pgv.AGraph(rankdir='BT')
sparseGraph = makeFromFile(graph, fileName)
    f = open(decomposition.name, 'r')
    generatePictures(f, graph, sparseGraph)

def generatePictures(f, graph, sparseGraph):
    decomp = False;
i = 0
d = dict()
for line in f:
    i += 1
    line.strip()  
    if decomp:
        line = str.replace(line, '(', '('
        line = str.replace(line, ')', ')
        line = str.replace(line, '\n', '')
        if line.count(')\n') == 0:
            line = str.replace(line, '\n\n', '')
        for part in line.split('[')[1:]:
            if len(part) > 0:
                lineList.append(tuple(map(int, part.split(','))))
```
else:
    lineList.append(tuple())
c = "%#s" % "".join( [hex(randrange(50, 255))[2:] for i in range(3)])
if (color):
    graph.node_attr["style"] = "filled"
    graph.get_node(lineList[0]).attr["fillcolor"] = c
if (color):
    sparseGraph.node_attr["style"] = "filled"
sparseGraph.get_node(lineList[0]).attr["fillcolor"] = c
for ii in xrange(1, len(lineList)):
    graph.get_node(lineList[ii]).attr["fillcolor"] = c
if (color):
    graph.add_edge(lineList[ii -1], lineList[ii], weight=1, color=c, penwidth=2)
else:
    graph.add_edge(lineList[ii -1], lineList[ii], weight=1)
if (color):
    sparseGraph.add_edge(lineList[ii -1], lineList[ii], weight=1, color=c, penwidth=2)
else:
    sparseGraph.add_edge(lineList[ii -1], lineList[ii], weight=1)
if (line == 'Decomposition:\n'):
    decomp = True

graphNoLabels = graph.copy()
graphNoLabels.node_attr["label"] = '_'
sparseGraphNoLabels = sparseGraph.copy()
sparseGraphNoLabels.node_attr["label"] = '_'
graph.layout(prog='dot')
graph.draw(str.replace(f.name, '.txt', '')+"_FULL.png")
sparseGraph.layout(prog='dot')
sparseGraph.draw(str.replace(f.name, '.txt', '')+"_SPARSE.png")
graphNoLabels.layout(prog='dot')
graphNoLabels.draw(str.replace(f.name, '.txt', '')+"_FULL_NOLABELS.png")
sparseGraphNoLabels.layout(prog='dot')
sparseGraphNoLabels.draw(str.replace(f.name, '.txt', '')+"_SPARSE_NO LABELS.png")

C.2 C++

We present the source code for the C++ language in this section. The program contains two classes, main and node. Each instance of the node class represents one vertex of the poset and each instance stores an id, vertex label, level index and a list of its neighbors. The classes are given below.

node.h:

```cpp
#ifndef NODE_H
#define NODE_H
```
#include <vector>
#include <string>
#include <map>
#include <fstream>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/set.hpp>
#include <boost/archive/text_ostreamarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/make_shared.hpp>
using namespace std;

typedef unsigned int IdType;
typedef unsigned char NameElementType;

class Node {
public:
    unsigned char level; // Good for up to 255 levels
    IdType id; // Can store up to 12! elements
    IdType boolSize; // store the number of elements in a set
    vector<NameElementType> name;
    vector<IdType> neighbors;

    Node();
    Node(std::vector<NameElementType> name, int level, IdType id);
    ~Node();
    void generateNextBool( map<vector<NameElementType>, boost::shared_ptr<Node>>*>& tmp_level, vector<vector<IdType>>* adjacentMatrix, vector<IdType>* nextLevel, std::ofstream* fileWithNodeNames);
    void generateNextInverse( map<vector<NameElementType>, boost::shared_ptr<Node>>*>& tmp_level, vector<vector<IdType>>* adjacentMatrix, vector<IdType>* nextLevel, std::ofstream* fileWithNodeNames);
    void generateNextYoung( map<vector<NameElementType>, boost::shared_ptr<Node>>*>& tmp_level, vector<vector<IdType>>* adjacentMatrix, vector<IdType>* nextLevel, std::ofstream* fileWithNodeNames, int m);
    string getName();
    string getNameYoung();
};

#endif // NODEH
```cpp
// libraries:
#include "node.h"
#include <vector>
#include <string>
#include <boost/concept_check.hpp>
#include <boost/lexical_cast.hpp>
using namespace std;

Node::Node(vector<NameElementType> name, int level, IdType id)
{
    this->name = name;
    this->level = level;
    this->id = id;
}

// Default constructors:
Node::Node()
{
}

Node::~Node()
{
}

// generate next vertices in the boolean lattice
void Node::generateNextBool(map<vector<NameElementType>, boost::shared_ptr<Node>> *tmpLevel, vector<vector<IdType>> **adjacentMatrix, vector<IdType>**nextLevel, std::ofstream** fileWithNodeNames)
{
    for (NameElementType i = 1; i <= this->boolSize; i++)
    {
        if (name.size() == 0 || find(name.begin(), name.end(), i) == name.end())
        {
            vector<NameElementType> tmp;
            for (int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
            sort(tmp.begin(), tmp.end());
            if (tmp.size() == 0){
                boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp, level+1, adjacentMatrix->size());
                newNode->boolSize = this->boolSize;
                string name = newNode->getName();
                int stringSize = (boolSize <= 9 ? 2+boolSize*2 : 2+9*2+(boolSize-9)*3);
                (*fileWithNodeNames)<< name;
                (*fileWithNodeNames)<< endl;
                fileWithNodeNames->flush();
                adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
                nextLevel->push_back(adjacentMatrix->size());
                adjacentMatrix->push_back(new vector<IdType>());
                (*tmpLevel)[tmp] = newNode;
            } else{
                adjacentMatrix->at(id)->push_back(tmp_level->at(tmp)->id);
            }
        }
    }
}

// generate next vertices using only inversion on an instantiation
```
void Node::generateNextInverse(map<vector<NameElementType>, boost::shared_ptr<Node>>* tmp_level, vector<vector<IdType>>* adjacentMatrix, vector<IdType>* nextLevel, std::ofstream* fileWithNodeNames)
{
    for (unsigned int i = 0; i < name.size() - 1; i++)
    {
        if (name.at(i) < name.at(i+1))
        {
            vector<NameElementType> tmp;
            for (int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
            NameElementType t = tmp.at(i);
            tmp.at(i) = tmp.at(i+1);
            tmp.at(i+1) = t;
            if (tmp.level->count(tmp) == 0){
                boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp, level+1, adjacentMatrix->size());
                string name = newNode->getName();
                (*fileWithNodeNames) << name << endl;
                fileWithNodeNames->flush();
                adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
                nextLevel->push_back(new vector<IdType>());
                (*tmp_level)[tmp] = newNode;
            }
            else{
                adjacentMatrix->at(id)->push_back(tmp_level->at(tm)->id);
            }
        }
    }
}

// generate next vertices in a Young's lattice
void Node::generateNextYoung(map<vector<NameElementType>, boost::shared_ptr<Node>>* tmp_level, vector<vector<IdType>>* adjacentMatrix, vector<IdType>* nextLevel, std::ofstream* fileWithNodeNames, int m)
{
    for (int i = 0; i < name.size(); i++)
    {
        if (name.at(i) < m & ( i == 0 || name[i-1] > name[i] )
        {
            vector<NameElementType> tmp;
            for (int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
            tmpt.level->count(tmp) == 0){
                boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp, level+1, adjacentMatrix->size());
                string name = newNode->getName();
                (*fileWithNodeNames) << name << endl;
                fileWithNodeNames->flush();
                adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
                nextLevel->push_back(new vector<IdType>());
                (*tmp_level)[tmp] = newNode;
            }
            else{
                adjacentMatrix->at(id)->push_back(tmp_level->at(tm)->id);
            }
        }
    }
}

// return name as a string
string Node::getName()
```cpp
{ string ret = "(";
 for(int i = 0; i < (int)name.size() - 1; i++){
    ret += boost::lexical_cast<string>((int)name.at(i));
    ret += ",";
 }
    if(name.size() > 0) ret += boost::lexical_cast<string>((int)name.at(name.size() - 1));
    ret += ")";
    return ret;
}
// return name as a string
string Node::getNameYoung()
{
    string ret = "(";
 for(unsigned int i = name.size() - 1; i > 0; i--){
    if((int)name.at(i) < 10) ret += "0";
    ret += boost::lexical_cast<string>((int)name.at(i));
    ret += ",";
 }
    if((int)name.at(0) < 10) ret += "0";
    ret += boost::lexical_cast<string>((int)name.at(0));
    ret += ")";
    return ret;
}
```
main.cpp:

```cpp
// libraries:
#include "algorithm"
#include <iostream>
#include <fstream>
#include <limits>
#include <time.h>
#include <iterator>
#include <vector>
#include <map>
#include <boost/program_options.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/algorithm/string.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/graph/boykov_kolmogorov_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/properties.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/serialization/map.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/shared_ptr.hpp>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/archive/binary_oarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>
#include "boost/graph/graph_traits.hpp"
#include "node.h"
#include <Python.h>
using std::vector;
using namespace boost;
namespace po = boost::program_options;
typedef adjacency_list_traits < vecS, vecS, directedS > Traits;

// vertex structure
struct Vertex{
    IdType id;
    unsigned char level;
    long distance;
    default_color_type color;
    Traits::edge_descriptor predecessor;
};

// edge structure
struct Edge{
    long capacity;
    long residual_capacity;
    Traits::edge_descriptor reverse;
};
typedef adjacency_list < vecS, vecS, directedS, Vertex, Edge> Graph;
```
typedef Graph::vertex_descriptor NodeId;
typedef Graph::edge_descriptor EdgeId;

Graph graph;

vector<vector<IdType>*> listOfLevels;
time_t start_time;
// contains ids of neighbors of a node
vector<vector<IdType>*> arrayOfEdges;

void removeUnusedEdges(Graph* graph, property_map<Graph, long Edge::*>::type* m_e_c, property_map<Graph, long Edge::*>::type* m_e_r_c, int bottom_layer, int upper_layer);

// output file
static std::ofstream* output;

// logger
static void loggerOut(std::string message){
cout << message << "\n";
*output << message << "\n";
}

unsigned int charInLine = 0;
// helper class
std::istream& GotoLine(std::istream& file, unsigned int num){
if (charInLine == 0){
file.seekg(std::ios::beg);
string line;
getline(file, line);
charInLine = file.tellg();
}
file.seekg(charInLine + num);
return file;
}

// add an edge between two vertices and set its parameters
EdgeId* AddEdge(Graph::vertex_descriptor &v1, Graph::vertex_descriptor &v2, const int capacity, Graph* g)
{
  EdgeId e1 = add_edge(v1, v2, *g).first;
  EdgeId e2 = add_edge(v2, v1, *g).first;
  g->operator[](e1).capacity = 1;
  g->operator[](e1).reverse = e2;
  g->operator[](e2).capacity = 0;
  g->operator[](e2).reverse = e1;
  g->operator[]((v2).predecessor = e2;
  return &e1;
}

// Uses Push–Relabel flow algorithm to find a matching between middle layers.
void findMatching(Graph* graph, int layer)
{
  loggerOut("\n--Middle part start--\n");
  Graph g = *graph;
  int l = listOfLevels.size() - layer - 1;
  NodeId sink = add_vertex(*graph);
  graph->operator[](sink).id = -1;
  for (vector<IdType>::const_iterator it = listOfLevels[l]->begin(), e = listOfLevels[l]->end(); it != e; ++it)
  {
// add the upper layer
    NodeId n = add_vertex(*graph); //(*it) - (*listOfLevels[l]->begin()) + 1;
    //id of a node in the graph
    graph->operator[](n).id = *it;
AddEdge(n, sink, 1, graph);
}

NodeId src = add_vertex(*graph);
for (vector<NodeId>::const_iterator it = listofLevels[l]->begin(), e =
  listofLevels[l+1]->begin(); it != e; ++it)
{
    //add bottom layer and connect nodes with its neighbors
    add_vertex(*graph);
    NodeId n = (*it) - (*listofLevels[l]->begin()) + 2 + listofLevels[l+1]->size
    (); //id of a node in the graph
    NodeId neigbour = arrayofEdges[*it]->begin(), ee
    = arrayofEdges[*it]->end(); neigbour != ee; ++neigbour)
    { NodelId nn = (*neigbour) - (*listofLevels[l+1]->begin()) + 1;
      AddEdge(n, nn, 1, graph);
    }
}

  property_map<Graph, long Edge::*>::type map_edge_capacity
    (get(&Edge::capacity,*
    graph) );
  property_map<Graph, long Edge::*>::type map_edge_residual_capacity
    (get(&Edge::residual_capacity,*graph) );
  property_map<Graph, Graph::edge_descriptor Edge::*>::type map_edge_reverse
    (get
    (&Edge::reverse,*graph ) );
  property_map<Graph, vertex_index_t>::type map_vertex_index
    = get(
    vertex_index_t,*graph);

  long flow = 0;
time_t time_start, time_end;
double time_push;
time(&time_start);
flow = push_relabel_max_flow(*graph, src, sink, map_edge_capacity,
  map_edge_residual_capacity, map_edge_reverse, map_vertex_index);
time(&time_end);
time.push = diff_time(time_end, time_start);
time(&time_start);
loggerOut("---Middle_part_done---"+lexical_cast<string>(&time.push));
removeUnusedEdges(graph,&map_edge_capacity,&map_edge_residual_capacity, 1, 1+1) ;

// removes edges without any flow
void removeUnusedEdges(Graph *graph, property_map<Graph, long Edge::*>::type*
  m_e_c, property_map<Graph, long Edge::*>::type* m_e_r_c, int bottom_layer,
  int top_layer){
  property_map<Graph, long Edge::*>::type map_edge_capacity = *m_e_c;
  property_map<Graph, long Edge::*>::type map_edge_residual_capacity = *m_e_r_c;
  graph_traits<Graph>::vertex_iterator u_iter, u_end, u_it;
  graph_traits<Graph>::edge_iterator ei, e_end;
  int id_bottom_first = listofLevels[bottom_layer]->at(0), id_bottom_last =
    listofLevels[bottom_layer]->at(listOfLevels[bottom_layer]->size()-1);
  int id_top_first = listofLevels[top_layer]->at(0), id_top_last = listofLevels[
    top_layer]->at(listOfLevels[top_layer]->size()-1);
```cpp
int id_before_top_first = listOfltents[top_layer-1]->at(0), id_before_top_last = listOfltents[top_layer-1]->at(listOfltents[top_layer-1]->size()-1);

for(int i = id_bottom_first; i <= id_bottom_last; i++) arrayOfEdges[i]->clear(); // erase all links for bottom layer
for(int i = id_before_top_first; i <= id_before_top_last; i++) arrayOfEdges[i]->clear(); // erase all links to upper layer

for(boost::tie(ei, e_end) = edges(*graph); ei != e_end; ei++)
if( (map_edge_capacity[ei] == 1 && (map_edge_capacity[ei] - map_edge_residual_capacity[ei]) == 1 ){
    // keep the edge - save the connected node into arrayOfEdges as the only neighbor
    IdType id_source = graph->operator[](source(ei, *graph)).id;
    IdType id_target = graph->operator[](target(ei, *graph)).id;
    if(id_source >= id_bottom_first && id_source <= id_bottom_last && id_target != -1 && id_target != id_source){
        // edge goes from bottom layer
        arrayOfEdges[id_source]->push_back(id_target);
    } else
    if(id_target >= id_top_first && id_target <= id_top_last && id_target != -1 && id_target != id_source){
        // edge goes to the top layer
        arrayOfEdges[id_source]->push_back(id_target);
    }
}
}

// extends the middle layer and run the flow algorithm on the constructed graph
int extendAndFlow(Graph* graph, int layer) // layer is the # of actual layer and it’s going to zero
{
    // flow starts
    int upperLayer = listOfltents.size()-1 - layer;
    NodeId sink = add_vertex(*graph);
    graph->operator[](sink).id = -1;
    NodeId src = add_vertex(*graph); // source node
    graph->operator[](src).id = -1;

    // upper layer
    for (vector<IdType>::const_iterator it = listOfltents[upperLayer]->begin(); it != listOfltents[upperLayer]->end(); ++it)
    {
        NodeId n = add_vertex(*graph);
        graph->operator[](n).id = *it;
        AddEdge(n,sink,1,graph); // edge in the middle
    }

    // middle layer
    upperLayer--;
    for (vector<IdType>::const_iterator it = listOfltents[upperLayer]->begin(); it != listOfltents[upperLayer]->end(); ++it)
    {
        NodeId n = add_vertex(*graph);
        graph->operator[](n).id = *it;
        NodeId nn = add_vertex(*graph);
        graph->operator[](nn).id = *it;
        AddEdge(n,nn,1,graph); // edge in the middle
    }
}```
for (vector<IdType>::const_iterator neighbour = arrayOfEdges[*it]->begin(), ee = arrayOfEdges[*it]->end(); neighbour != ee; ++neighbour) {
    NodeId tmp = vertex((*neighbour) - (*listOfLevels[upperLayer+1]->begin()) + 2,*graph);
    AddEdge(nn,tmp,1,graph);
}

//bottom layer
for (vector<IdType>::const_iterator it = listOfLevels[layer]->begin(), e = listOfLevels[layer]->end(); it != e; ++it) {
    NodeId n = add_vertex(*graph);
    graph->operator[](n).id = *it;
    AddEdge(src,n,1,graph);
    for (vector<IdType>::const_iterator neighbor = arrayOfEdges[*it]->begin(), ee = arrayOfEdges[*it]->end(); neighbor != ee; ++neighbor) {
        NodeId nn = (*neighbor);
        for (int i = layer; i < upperLayer-1; i++) nn = arrayOfEdges[nn]->at(0);
        nn = vertex(2*(nn - listOfLevels[upperLayer]->at(0))+2+listOfLevels[upperLayer+1]->size(),*graph);
        AddEdge(n,nn,1,graph);
        graph->operator[](nn).id = *neighbor;
    }
}

property_map<Graph, long Edge::*>::type map_edge_capacity = get(&Edge::capacity,*graph);
property_map<Graph, long Edge::*>::type map_edge_residual_capacity = get(&Edge::residual_capacity,*graph);
property_map<Graph, Graph::edge_descriptor Edge::*>::type map_edge_reverse = get(&Edge::reverse,*graph);
property_map<Graph, vertex_index_t>::type map_vertex_index = get(vertex_index_t,*graph);

long flow = 0;
time(time_start, time_end);
double time_push;
time(&time_start);
flow = push_relabel_max_flow(*graph,src,sink,map_edge_capacity, map_edge_residual_capacity, map_edge_reverse, map_vertex_index);
time(&time_end);
time_push = diff_time(time_end, time_start);
time(&time_start);
loggerOut("---flow_end---"+lexical_cast<string>(time_push));
removeUnusedEdges(graph,&map_edge_capacity,&map_edge_residual_capacity,layer, upperLayer+1);
return flow;

} // the main class, which tries to find a Symmetric Saturated Chain Decomposition of poset saved in listOfLevels and in arrayOfEdges
void findSSCD(Graph *g, boost::program_options::variables_map vm){
    Graph graph = *g;
```cpp
int layer;
string name = "nodes.dat";

if (listOfLevels.size() % 2 == 0)
{
    // Two layers in the middle
    loggerOut("--Two_layers_in_the_middle--");
    // layer is the number of the actual layer; layer goes to zero
    layer = listOfLevels.size()/2 - 1;
    graph.clear();
    findMatching(&graph, layer--);
}
else
{
    // One layer in the middle
    loggerOut("--One_layer_in_the_middle--");
    layer = listOfLevels.size()/2 - 1;
    graph.clear();
    extendAndFlow(&graph, layer--);
}
while (layer >= 0){
    graph.clear();
    loggerOut("Actual_layer_is"+lexical_cast<string>(layer)+"/"+lexical_cast<string>(listOfLevels.size()));
    if (extendAndFlow(&graph, layer--) != listOfLevels[layer+1]->size()){
        loggerOut("--Possibly_no_symmetric_chain_decomposition! Terminating...--");
        output->flush();
        exit(EXIT_FAILURE);
    }
}

// Print founded symmetric chains decomposition
std::ifstream fileWithNodeNameRead(name.c_str());
std::ifstream& nodeNames = fileWithNodeNameRead;
name = "decomposition.dat.tmp";
std::ofstream fileWithTmpDecomposition(name.c_str(), ios::trunc);
string line;

time(&tmp);
loggerOut("--Total_time:\"+lexical_cast<string>(tmp-start_time)+"--");

loggerOut("Decomposition:");
if (listOfLevels.size() % 2 == 1){
    for (vector<IdType>::const_iterator it = listOfLevels.begin(); it != listOfLevels.end(); ++it){
        if (arrayOfEdges[*it]->size() == 0){
            GotoLine(nodeNames, *it);
            getline(nodeNames, line);
            trim(line);
            (vm.count("file") ? (fileWithTmpDecomposition << "(" << *it << ")" << endl) : (fileWithTmpDecomposition << line << endl));
        }
    }
    for (int i = 0; i < listOfLevels.size()/2; i++){
```
for (vector<IdType>::const_iterator it = listOfLevels[i]->begin(), e =
    listOfLevels[i]->end(); it != e; ++it) {
    IdType tmp = *it, tmp_new;
    if (arrayOfEdges[tmp]->size() > 0) {
        fileWithTmpDecomposition << endl;
    }
    bool print = false;
    if (arrayOfEdges[tmp]->size() > 0) {
        do {
            print = true;
            GotoLine(nodeNames, tmp);
            getline(nodeNames, line);
            trim(line);
            (vm.count("file") ? (fileWithTmpDecomposition << "(" << tmp << ")") :
                (fileWithTmpDecomposition << line);
            tmp_new = arrayOfEdges[tmp]->at(0);
            arrayOfEdges[tmp]->clear();
            tmp = tmp_new;
        } while (arrayOfEdges[tmp]->size() > 0);
        if (print && tmp != arrayOfEdges.size() - 1) {
            GotoLine(nodeNames, tmp);
            getline(nodeNames, line);
            trim(line);
            (vm.count("file") ? (fileWithTmpDecomposition << "(" << tmp << ")") :
                (fileWithTmpDecomposition << line);
        }
    }
}
fileWithTmpDecomposition.close();

for(int i = 0; i < arrayOfEdges.size(); i++) delete arrayOfEdges[i];
for(int i = 0; i < listOfLevels.size(); i++) delete listOfLevels[i];

// erase empty lines printed to decomposition.dat.tmp
name = "decomposition.dat.tmp";
std::ifstream fileWithTmpDecompositionRead(name.c_str());
std::ofstream decomposition = fileWithTmpDecompositionRead;
name = "decomposition.dat";
std::ofstream fileWithDecomposition(name.c_str(), ios::trunc);
while (getline(decomposition, line)) {
    if (line.size() > 0) {
        fileWithDecomposition << line << endl;
        loggerOut(line);
    }
}
fileWithTmpDecompositionRead.close();
remove("decomposition.dat.tmp");

fileWithNodeNameRead.close();
fileWithDecomposition.close();
output->flush();
output->close();
delete output;
remove("nodes.dat");
remove("decomposition.dat");
// helper to load the Inverse poset into the memory

void makeBoolPoset(Graph *g, boost::program_options::variables_map vm) {
    Graph graph = *g;
    int layer;
    string name = "nodes.dat";
    std::ofstream fileWithNodeName(name.c_str(), ios::trunc);
    boost::shared_ptr<Node> root;

    vector<NameElementType> rootName;
    root = boost::make_shared<Node>(rootName, 0, 0);
    root->boolSize = vm["bool"].as<int>();
    string fileName = boost::lexical_cast<string>(root->boolSize) + ".txt";
    output = new std::ofstream(fileName.c_str(), ios::trunc);
    arrayOfEdges.push_back(new vector<IdType>());

    name = root->getName();
    int stringSize = (root->boolSize <= 9 ? 2 + root->boolSize * 2 : 2 + 9 * 2 + (root->boolSize - 9) * 3);
    fileWithNodeName << name;
    for (int k = 0; k < stringSize - name.length(); k++) fileWithNodeName << "\n";
    fileWithNodeName << endl;
    map<vector<NameElementType>, boost::shared_ptr<Node>>* layerAct = new map<vector<NameElementType>, boost::shared_ptr<Node>>;
    layerAct->operator[](root->name) = root;

    vector<IdType>* tmp_level = new vector<IdType>();
    tmp_level->push_back(root->id);
    listOfLevels.push_back(tmp_level);
    map<vector<NameElementType>, boost::shared_ptr<Node>>* layerNext;
    map<vector<NameElementType>, boost::shared_ptr<Node>>::const_iterator it, e;

    // generating all elements
    while (true) {
        layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node>>();
        tmp_level = new vector<IdType>();
        for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it)
            { if (tmp_level->size() == 0) break;
                listOfLevels.push_back(tmp_level);
                layerAct->clear();
                delete layerAct;
                layerAct = layerNext;
            }

        // include small loop for the top node for printing decomposition
        arrayOfEdges[arrayOfEdges.size() - 1]->push_back(arrayOfEdges.size() - 1);
        delete layerAct;
        fileWithNodeName.close();
        // calling Python diagram class to generate pictures
        if (listOfLevels.size() > 1) {
    
}
findSSCD(&graph, vm);
if (vm.count("no_picture") == 0) {
    Py_Initialize();
    PyRun_SimpleString("import os, sys");
    PyRun_SimpleString("sys.path.append(os.getcwd())");
    PyRun_SimpleString("import diagram");
    ofstream py("python.in");
    py << "f=fopen(" << fileName << ",r")\n.close()\ndiagram.
    makeGraphOfBool(f,"<< root->boolSize <<")";
    py.close();
    FILE *fp = fopen("python.in", "r");
    PyRun_SimpleFile(fp, "python.in");
    fclose(fp);
    Py_Finalize();
    remove("python.in");
} else
    loggerOut("−−The poset has only one vertex. The SSCD is obvious.−−");
}

void makeInversePoset(Graph *g, boost::program_options::variables_map vm) {
    Graph graph = *g;
    int layer;
    string name = "nodes.dat"
    std::ofstream fileWithNodeName(name.c_str(), ios::trunc);
    boost::shared_ptr<Node> root;
    vector<NameElementType> rootName(vm["inversion"].as<vector<int>>().begin(),vm["inversion"].as<vector<int>>().end());
    root = boost::make_shared<Node>(rootName, 0.0);
    string fileName = root->getName().substr(1,root->getName().size() - 2) + ".txt";
    ofstream output = new std::ofstream(fileName.c_str(), ios::trunc);
    arrayOfEdges.push_back(new vector<IdType>());
    fileWithNodeName << root->getName() << endl;
    map<vector<NameElementType>, boost::shared_ptr<Node>> *layerAct = new map<vector<NameElementType>, boost::shared_ptr<Node>>;
    layerAct->operator[] (root->name) = root;
    vector<IdType> *tmp_level = new vector<IdType>();
    tmp_level->push_back(root->id);
    listOfLevels.push_back(tmp_level);
    map<vector<NameElementType>, boost::shared_ptr<Node>> *layerNext;
    map<vector<NameElementType>, boost::shared_ptr<Node>> *layerAct;
    map<vector<NameElementType>, boost::shared_ptr<Node>> *layerNext;
    vector<IdType> *tmp_level = new vector<IdType>();
    tmp_level->push_back(root->id);
    listOfLevels.push_back(tmp_level);
    // generating all elements
    while (true) {
        layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node>>();
        tmp_level = new vector<IdType>();
        for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it) {
            it->second->generateNextInverse(layerNext, &arrayOfEdges, tmp_level, &
            fileWithNodeName);
        }
        if (tmp_level->size() == 0) break;
        listOfLevels.push_back(tmp_level);
    }
layerAct->clear();
delete layerAct;

layerAct = layerNext;

// include small loop for the top node for printing decomposition
arrayOfEdges[arrayOfEdges.size()-1]->push_back(arrayOfEdges.size()-1);
delete layerAct;
fileWithNameName.close();

// calling Python diagram class to generate pictures
if (listOfLevels.size() > 1){
    findSSCD(&graph, vm);
    if (vm.count("nopict") == 0){
        Py::Initialize();
        PyRun_SimpleString("import_os,sys");
        PyRun_SimpleString("sys.path.append(os.getcwd())");
        PyRun_SimpleString("import_diagram");
        ofstream py("python.in");
        py << "f = open(" << fileName << "," << "t")\n.close()\ndiagram.
        makeGraphOfInversePartiallyOrderedMultiSet(f, "< < root->getName().substr(1,root->getName().size()-2)<stdio"]");
        py.close();
        FILE *fp = fopen("python.in", "r");
        PyRun_SimpleFile(fp,"python.in");
        fclose(fp);
        Py::Finalize();
        remove("python.in");
    }
}
else
    loggerOut("--The poset has only one vertex. The SSDC is obvious.--");

// helper to load the Young's lattice into the memory
void makeYoungsLattice(Graph *g, boost::program_options::variables_map vm){
    Graph graph = *g;
    int layer;
    string name = "nodes.dat";
    ofstream fileWithNameName(name.c_str(), ios::trunc);
    boost::shared_ptr<Node> root;
    int m = vm["young"].as<vector<int>>()[0], n = vm["young"].as<vector<int>>()[1];
    vector<NameElementType> rootName;
    for (int i = 0; i < n; i++){
        rootName.push_back(0);
    }
    root = boost::make_shared<Node>(rootName, 0, 0);
    string fileName = "";
    for (int i = 1; i < n; i++) fileName += lexical_cast<string>(m) + ",";
    fileName += lexical_cast<string>(m) + ".txt";
    output = new std::ofstream(fileName.c_str(), ios::trunc);
    arrayOfEdges.push_back(new vector<IdType>());
    fileWithNameName << root->getNameYoung() << endl;
map<vector<NameElementType>, boost::shared_ptr<Node> >* layerAct = new map<vector<NameElementType>, boost::shared_ptr<Node> >;
layerAct->operator[](root->name) = root;

vector<IdType>* tmp_level = new vector<IdType>();
tmp_level->push_back(root->id);
listOfLevels.push_back(tmp_level);
map<vector<NameElementType>, boost::shared_ptr<Node> >* layerNext;

layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node> >();
tmp_level = new vector<IdType>();
for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it) {
  it->second->generateNextYoung(layerNext, arrayOfEdges, tmp_level, &fileWithNodeName, m);
}
if (tmp_level->size() == 0) break;
listOfLevels.push_back(tmp_level);
layerAct->clear();
delete layerAct;
layerAct = layerNext;

while (true) {
  layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node> >();
tmp_level = new vector<IdType>();
  for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it) {
    it->second->generateNextYoung(layerNext, arrayOfEdges, tmp_level, &fileWithNodeName, m);
  }
  if (tmp_level->size() == 0) break;
  listOfLevels.push_back(tmp_level);
  layerAct->clear();
  delete layerAct;
  layerAct = layerNext;
}

arrayOfEdges[arrayOfEdges.size() - 1]->push_back(arrayOfEdges.size() - 1);
delete layerAct;
fileWithNodeName.close();

if (listOfLevels.size() > 1) {
  findSSCD(&graph, vm);
  if (vm.count("no.pict") == 0) {
    Py_Initialize();
    PyRun_SimpleString("import os, sys");
    PyRun_SimpleString("sys.path.append(os.getcwd())");
    PyRun_SimpleString("import diagram");
    ofstream py("python.in");
    py << "f = open(" << fileName << ",r")\n    diagram.makeGraphOfYoungsLattice(f,"+lexical_cast<string>(m)+","+lexical_cast<string>(n)+")
    py.close();
    FILE *fp = fopen("python.in", "r");
    PyRun_SimpleFile(fp,"python.in");
    fclose(fp);
    Py_Finalize();
    remove("python.in");
  }
  else
    loggerOut("--The poset has only one vertex. The SSCD is obvious.--");
}

// helper to load a poset from the given file into the memory
void readPosetFromFile(Graph* g, boost::program_options::variables_map vm){
// Load all nodes
Graph graph = *g;
string fileName = vm["file"].as<string>() + ".txt";
output = new std::ofstream(fileName.c_str(), ios::trunc);
string name = vm["file"].as<string>() + "_nodes";
std::ifstream ifFile(name.c_str());
string line;
while (getline(ifFile, line))
{
    vector<string> strs;
    boost::split(strs, line, boost::is_any_of(" ", ")");
    arrayOfEdges.push_back(new vector<IdType>());
    for (int i = 0; i < strs.size(); i++)
    {
        if (strs[0].size() == 0) continue;
        if (strs.at(i).at(strs.at(i).size() - 1) == 13) strs.at(i).resize(strs.at(i).size() - 1);
        arrayOfEdges[arrayOfEdges.size() - 1].push_back(lexical_cast<IdType>(strs.at(i)));
    }
}
ifFile.close();

// load the levels structure
name = vm["file"].as<string>() + ".levels";
ifFile.open(name.c_str());
isFile.copyfmt(ifFile);
while (getline(isFile, line))
{
    vector<string> strs;
    boost::split(strs, line, boost::is_any_of(" ", ")");
    listOfLevels.push_back(new vector<IdType>());
    for (int i = 0; i < strs.size(); i++)
    {
        if (strs.at(i).at(strs.at(i).size() - 1) == 13) strs.at(i).resize(strs.at(i).size() - 1);
        listOfLevels[listOfLevels.size() - 1].push_back(lexical_cast<IdType>(strs.at(i)));
    }
}

// calling Python diagram class to generate pictures
if (listOfLevels.size() > 1)
{
    findSSCD(&graph, vm);
    if (vm.count("no_picture") == 0)
    {
        Py_Initialize();
        PyRun_SimpleString("import os, sys");
        PyRun_SimpleString("sys.path.append(os.getcwd())");
        PyRun_SimpleString("import diagram");
        ofstream py("python.in");
        py << "f = open(" << fileName << ",r")" << close() << 
            "makeGraphFromFile(f, "" +vm["file"].as<string>() + "+")";
        py.close();
        FILE *fp = fopen("python.in", "r");
        PyRun_SimpleFile(fp, "python.in");
        fclose(fp);
        Py_Finalize();
        remove("python.in");
    }
    else
    {
        loggerOut("--The poset has only one vertex. The SSDC is obvious.--");
    }
}
```cpp
#include <boost/program_options.hpp>

int main(int argc, char** argv)
{
    time(&start_time);

    boost::program_options::options_description desc("Allowed options");
    desc.add_options()
        ("bool", po::value<int>(), "set the number of elements in a set; --bool A")
        ("file", po::value<std::string>(), "--file [name] \nRead a poset from files. There have to be two files in the current folder, \n[name]_nodes and \n[name]_levels. Each line of \n[name]_nodes contains a list of neighbors of node with id equal to the line number starting from zero. Each line of \n[name]_levels contains a list of nodes in the layer which number corresponds to the number of the current line starting from zero. \n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n##