Sorting and Selection Problems in Partially Ordered Sets

Merkouris Papamichail
AL1.20.0018

Examination committee:
Stavros Kolliopoulos, DIT, NKUA.
Ioannis Emiris, DIT, NKUA.
Archontia Giannopoulou, DIT, NKUA.

Supervisor:
Stavros Kolliopoulos, Professor,
Department of Informatics and Telecommunications,
National and Kapodistrian University of Athens.

Μεταπτυχιακό Πρόγραμμα «Αλγόριθμοι, Λογική και Διακριτά Μαθηματικά» – 2016

ABSTRACT

In this thesis we present some results from the literature regarding sorting and selection problems in partially ordered sets. In the sorting problem we are given a partially ordered set $\mathcal{U}$ and access to an oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \nsim\}$. We are able to compare two elements of $\mathcal{U}$, only by querying the oracle function. Our goal is to retrieve the underlying unknown partial order. In the k-selection problem, we are required to find the k-smallest elements in the same setting. In particular we examine two models, the Width-Based Model [1] and the Forbidden Comparisons Model [2]. In the Width-Based Model we are additionally given an upper bound w on the width of the underlying poset. The main result we present in this setting is the query-optimal Entropy Sort algorithm with $O(n \log n{+}nw)$ query complexity, but exponential overall time [1]. We also examine some randomised and deterministic algorithms in this setting for the selection problem. On the other hand, the Forbidden Comparisons Model, the oracle function is defined slightly differently, i.e. $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, where $c(a, b) =\perp$ when we are not allowed to compare the two elements a, b; we deduce their relation (if possible) through transitivity. We are also given an undirected comparison graph $G = (V, E)$, where there exists the edge $\{a, b\}$ if the two elements can be compared, i.e. $c(a, b) \neq\perp$. Moreover, we denote with q the number of missing edges, i.e. $q = \binom{|V|}{2} - |E|$. In this setting we present the algorithm in [3] with $O((q{+}n) \log(n^2/q))$ query complexity and $O(n^\omega)$ time complexity, where ω is the exponent of the matrix multiplication. Lastly, we examine the special cases of chordal and comparability graphs, where we present an algorithm, also due to [3], with $O(n \log n)$ query and $O(n^\omega)$ time complexity, respectively.

ΣΥΝΟΨΗ

Σε αυτή την εργασία παρουσιάζουμε κάποια αποτελέσματα από την βιβλιογραφία σχετικά με προβλήματα ταξινόμησης και επιλογής σε μερικώς διατεταγμένα σύνολα. Στο πρόβλημα ταξινόμησης μας δίνεται ένα μερικώς διατεταγμένο σύνολο $\mathcal{U}$ και, επιπλέον, μια συνάρτηση μαντείου $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$. Έχουμε την δυνατότητα να συγκρίνουμε δύο στοιχεία του $\mathcal{U}$, μόνο μέσω ερωτημάτων στη συνάρτηση μαντείου. Ο σκοπός μας είναι να ανακατασκευάσουμε την υποκείμενη, άγνωστη μερική διάταξη. Στο πρόβλημα k-επιλογής, μας ζητείτε να βρούμε τα k-μικρότερα στοιχεία στο ίδιο πλαίσιο. Ειδικότερα, εξετάζουμε δύο μοντέλα, το Μοντέλο Φραγμένου Πλάτους [1] και το Μοντέλο Απαγορευμένων Συγκρίσεων [2]. Στο Μοντέλο Φραγμένου Πλάτους μας δίνεται επιπλέον ένα άνω φράγμα w στο πλάτος της μερικής διάταξης. Το κύριο αποτέλεσμα που εξετάζουμε, σε αυτό το πλαίσιο, είναι ο βέλτιστος, ως προς τα ερωτήματα, Αλγόριθμος Entropy Sort, με $O(n \log n + nw)$ πολυπλοκότητα ερωτημάτων αλλά εκθετική πολυπλοκότητα χρόνου. Επιπλέον, εξετάζουμε κάποιους τυχαιοκρατικούς και ντετερμινιστικούς αλγορίθμους σε αυτό το πλαίσιο για το πρόβλημα επιλογής. Από την άλλη πλευρά, στο πλαίσιο του Μοντέλου Απαγορευμένων Συγκρίσεων, η συνάρτηση μαντείου ορίζεται κάπως διαφορετικά, δ.δ. $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, όπου $c(a,b) =\perp$ όταν δεν μας επιτρέπεται να συγκρίνουμε τα δύο στοιχεία a, b· συμπεραίνουμε αυτές τις σχέσεις (αν υπάρχουν) μέσω της μεταβατικότητας. Μας δίνεται, επίσης, ένα μη-κατευθυνόμενο γράφημα συγκρίσεων $G = (V, E)$, όπου υπάρχει η ακμή $\{a, b\}$ αν τα δύο στοιχεία μπορούν να συγκριθούν, δ.δ. $c(a,b) \neq\perp$. Επιπλέον, με q συμβολίζουμε τον αριθμό των ακμών που λείπουν, δ.δ. $q = \binom{|V|}{2} - |E|$. Σε αυτό το πλαίσιο παρουσιάζουμε τον αλγόριθμο από το [3] με $O((q+n)\log(n^2/q))$ πολυπλοκότητα ερωτημάτων και $O(n^\omega)$ χρονική πολυπλοκότητα, όπου ω είναι ο εκθέτης του πολλαπλασιασμού πινάκων. Τέλος, εξετάζουμε τις ειδικές περιπτώσεις χορδικών γραφημάτων και γραφημάτων συγκρισιμότηστας, όπου δείχνουμε αλγορίθμους, πάλι από το [3], με $O(n \log n)$ πολυπλοκότητα ερωτημάτων και $O(n^\omega)$ χρονική πολυπλοκότητα.

CONTENTS

Prologue 1

1 Introduction 3
1.1 Relations . . . 3
1.2 Graph Theory . . . 4
1.3 Posets . . . 6
1.3.1 Graphical Representation of a Poset . . . 7
1.3.2 k-Smallest Elements . . . 7
1.3.3 Mirsky's and Dilworth's Theorems . . . 9
1.4 Sorting and Selection Problems . . . 11
1.4.1 Sorting in Totally Ordered Sets . . . 12
1.4.2 Selection in Totally Ordered Sets . . . 13

2 Algorithms in Flows, Matchings and Posets 17
2.1 Networks and Flows . . . 17
2.1.1 Flow Networks . . . 17
2.1.2 Cuts in Networks . . . 20
2.2 Matchings . . . 21
2.2.1 Hall's Theorem . . . 23
2.2.2 König's Theorem . . . 25
2.3 Finding a Minimum Chain Decomposition . . . 26

3 Width-Based Model: Sorting Algorithms 29
3.1 Representing a poset: The ChainMerge data structure . . . 30
3.2 Bin-Insertion Sort . . . 31
3.2.1 Greedy Counter-Example . . . 33
3.3 Entropy Sort . . . 34
3.3.1 Weighted Binary Search . . . 35
3.3.2 The Algorithm . . . 37
3.3.3 Analysis . . . 37
3.4 Merge Sort . . . 40
3.4.1 Merge Sort . . . 40
3.4.2 Peeling Algorithm . . . 41

3.4.3 Lifting the Known Width Hypothesis 44
3.5 Conclusions 45

4 Width-Based Model: Selection Algorithms 47
4.1 Upper Bounds 48
4.1.1 Selection Problem 48
4.1.2 k-Selection Problem 49
4.2 Lower Bounds 52
4.2.1 Selection Problem 52
4.2.2 k-Selection Problem 54
4.3 Conclusions 55

5 Forbidden Comparisons Model 57
5.1 Graph Orientations & Tournaments 58
5.1.1 Layer Decomposition & Linear Extension 60
5.2 Comparability & Comparison Graphs 61
5.2.1 Gallai's Theorem 62
5.3 Sorting Under Forbidden Comparisons 63
5.3.1 Upper Bounds 64
5.3.2 Lower Bounds 67
5.4 Special Cases 67
5.4.1 Chordal Graphs 69
5.4.2 Comparability Graphs 69
5.5 Conclusions 71

6 Conclusion 73

Bibliography 77

Index 81

PROLOGUE

Sorting problems have been in the heart of computer science since its conception in the middle of twentieth century. A considerable amount of time is spent in undergraduate courses to study efficient sorting algorithms. In general, these algorithms deal with totally ordered sets, as a list of integers. Moreover, the notion of efficiency that is utilized is the worst case time complexity of an algorithm. In this thesis we move in a different direction from these traditional computer science topics. We will generalize our setting to include *partially ordered sets*. Additionally the measure of efficiency will be the number of calls to an oracle function. For some partially ordered set $\mathcal{U}$, our algorithms will only have access to an oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$, returning " $\preceq$ " if a is *smaller than, or equal to* b, " $\succ$ " if a is *greater than* b, and " $\not\sim$ " if the two arguments are not related.

We will consider two basic models. The first is due to Daskalakis et al. [1], where we only have access to the oracle function; we are also given some constant w, an upper bound on the *width* of the poset. We examine query-optimal or time-efficient upper and lower bounds, with respect to the constant w. We call this setting *Width-Based Model*. On the other hand, we consider a model due to Banerjee et al. [2], where we are given access to the oracle, but also to a *comparison graph* $G = (\mathcal{U}, E)$. We are allowed to *only call* the oracle for two elements, that are connected in G. There our parameter will be $q = \binom{n}{2} - |E|$, the number of*forbidden comparisons*. We call this setting *Forbidden Comparisons Model*. We will also examine some additional results due to Biswas et al. [3] in this model.

We organize this thesis as follows. In Chapter 1 we introduce the basic concepts and notions of partial orders. In this chapter, we prove the elementary theorems on poset theory of Dilworth and Mirsky regarding the width of a partial order. In Chapter 2 we present the algorithmic aspects of posets based on Ford's and Fulkerson's *network flows*. There, we present an additional *algorithmic* proof of Dilworth's Theorem. Our proof will induce an algorithm for finding a *minimum chain decomposition* of a poset. We use these algorithms as a subroutine in the sequel. In Chapters 3 and 4 we explore the results of Daskalakis' paper [1] on the Width-Based Model. In Chapter 3 we consider sorting algorithms on posets and examine an optimal algorithm, with respect to the oracle calls; the Entropy Sort. In Chapter 4 we examine the k-selection problem and its algorithms; we also provide some elementary lower bounds. Subsequently, in Chapter 5 we examine the results of Banerjee et al. [2] and Biswas et al. [3] on the Forbidden Comparisons Model. Banerjee's et al. work present an efficient algorithm on this model for the sorting problem. On the other hand, the Biswas et al. paper gives some interesting lower bounds and improvements on Banerjee's algorithm on special cases. Lastly, in Chapter 6, we summarize the above results and present some routes for future work.

This thesis has been written in partial fulfillment of the requirements for Master's Degree in Algorithms, Logic, and Discrete Mathematics, of National and Kapodistrian University of Athens and National Technical University of Athens. I would like to thank my advisor prof. Stavros Kolliopoulos for his guidance and insights throughout the writing of this diploma thesis, and for turning my attention to such an intriguing topic. Additionally, I would like to thank the other two members of the three-member committee, prof. Ioannis Emiris and prof. Archontia Giannopoulou for directing me to discover new and exiting areas in my post-graduate studies.

Athens,
September 2022

CHAPTER 1

INTRODUCTION

In this chapter we discuss some fundamental notions that we will use throughout this thesis. We begin by formally defining the notion of a *relation* in Section 1.1. Moreover, in this section we discuss *equivalence*, *partial* and *total* order relations. In Section 1.2 we introduce the basic concepts and notions of Graph Theory. *Graphs* will provide an intuitive way to encode and gain a deeper understanding of a given partial order. Subsequently, in Section 1.3 we define the discrete structure of a *partially ordered set*, the core notion of this essay. There we prove the key theorems of Mirsky and Dilworth regarding partially ordered sets. Lastly, in Section 1.4 we introduce the algorithmic setting of our interest and establish the *query complexity* as a measure of an algorithm's efficiency.

1.1 Relations

We begin this section by defining the core mathematical notion of a *relation*. Next we focus our attention in relations with interesting properties, such as *equivalence*, *partial* and *total* order relations. Lastly, we provide an elegant method to compute the *transitive closure* on any relation, using matrix multiplication.

Definition 1.1 (Binary Relation)**.** Let $\mathcal{U}$ be a finite set. Also, let $\mathcal{R} \subseteq \mathcal{U} \times \mathcal{U}$ be a subset of pairs on $\mathcal{U}$. We call the set of pairs $\mathcal{R}$ a *binary relation* on $\mathcal{U}$.

A binary relation is an abstract notion, that may represent any mathematical relation between two objects.There are some properties of a binary relation that introduce an additional order to the relation.

Definition 1.2 (Relation Properties)**.** Let $\mathcal{U}$ be a finite set, and $\mathcal{R} \subseteq \mathcal{U} \times \mathcal{U}$ be a binary relation on $\mathcal{U}$. We define the following properties for $\mathcal{R}$.

1. We call the relation $\mathcal{R}$ *reflexive*, if for every $x \in \mathcal{U}$, we have $(x, x) \in \mathcal{R}$.
2. We call the relation $\mathcal{R}$ *symmetric*, if for two elements $x, y \in \mathcal{U}$, where $(x, y) \in \mathcal{R}$, we have $(y, x) \in \mathcal{R}$.
3. We call the relation $\mathcal{R}$ *antisymmetric*, if for two *distinct* elements $x, y \in \mathcal{U}$, $x \neq y$, where $(x, y) \in \mathcal{R}$, we have $(y, x) \notin \mathcal{R}$.

4. We call the relation $\mathcal{R}$ *transitive*, if for some elements $x, y, z \in \mathcal{U}$, where $(x, y) \in \mathcal{R}$ and $(y, z) \in \mathcal{R}$, we have $(x, z) \in \mathcal{R}$.

We may use the properties of Definition 1.2 to define *equivalence* and *order* relations.

Definition 1.3 (Equivalence Relation)**.** Let $\mathcal{U}$ be a finite set, and $\mathcal{R}_{\asymp} \subseteq \mathcal{U} \times \mathcal{U}$ a binary relation of $\mathcal{U}$. We call the relation $\mathcal{R}_{\asymp}$ an *equivalence* relation, if it is *reflexive*, *symmetric* and *transitive*.

Definition 1.4 (Order Relations)**.** Let $\mathcal{U}$ be a finite set, and $\mathcal{R}_{\preceq} \subseteq \mathcal{U} \times \mathcal{U}$ a binary relation of $\mathcal{U}$. We call the relation $\mathcal{R}_{\preceq}$ *partial order* if it is *reflexive*, *antisymmetric* and *transitive*. If for every two elements of the universe $x, y \in \mathcal{U}$, we have either $(x, y) \in \mathcal{R}_{\preceq}$, or $(y, x) \in \mathcal{R}_{\preceq}$, we will be calling $\mathcal{R}_{\preceq}$ a *total* order on $\mathcal{U}$.

We may enforce the transitive property to any relation, by applying the *transitive closure* operator.

Definition 1.5 (Transitive Closure)**.** Let $\mathcal{U}$ be a finite set and $\mathcal{R} \subseteq \mathcal{U} \times \mathcal{U}$ a binary relation on $\mathcal{U}$. Let $[\mathcal{R}] \supseteq \mathcal{R}$ be the *smallest* super set of $\mathcal{R}$, such that the transitive property holds. We will be calling $[\mathcal{R}]$ the *transitive closure* of $\mathcal{R}$, while we denote with $[\cdot]$ the *transitive closure operator* .

There is a simple and elegant algorithm to compute the transitive closure on any finite relation. Let $\mathcal{R}$ be a binary relation on a finite set $\mathcal{U}$. We denote with $A_{\mathcal{R}} \in \{0, 1\}^{|\mathcal{U}| \times |\mathcal{U}|}$ the *adjacency* matrix of the relation $\mathcal{R}$, i.e.

$$a_{x,y} = \begin{cases} 1, & \text{if } (x, y) \in \mathcal{R} \\ 0, & \text{otherwise} \end{cases}$$

We have that $A_{[\mathcal{R}]} = A_{\mathcal{R}}^{|\mathcal{U}|}$, where the matrix multiplications is applied on the field of Boolean Algebra, i.e. $\mathbb{B} = \langle \{0, 1\}, \vee, \wedge \rangle$. In $\mathbb{B}$ the logical operation *or* $\vee$ replaces the addition, and the logical operation *and* $\wedge$ replaces the multiplication. Hence, the formula that gives us the element $a_{x,y}$ of $A_{\mathcal{R}}^2$ becomes,

$$a_{x,y} = \bigvee_{k \in |\mathcal{U}|} (a_{x,k} \wedge a_{k,y}). \tag{1.1}$$

Equation 1.1 describes the fact that if (x, k) and (k, y) are elements of $\mathcal{R}$, then (x, y) should also be an element of $\mathcal{R}$. Thus we apply the transitivity in elements of "distance" two. The reason for multiplying the adjacency matrix $n = |\mathcal{U}|$ times with itself, is so that we *connect* the elements x_1, x_n, in a sequence of the form $(x_1, x_2), (x_2, x_3), \ldots,$ (x_{n-1}, x_n) in $\mathcal{R}$, i.e. of distance n. If ω is the exponent of the matrix multiplication complexity $O(n^{\omega})$, then the above algorithm consumes $O(n^{\omega+1})$ time[1].

1.2 Graph Theory

In this section we present some basic definitions and results regarding *graphs*. These Graph Theory notions will provide us with the basic mathematical tools, that we will be using throughout this thesis. We begin our presentation with the formal definition of a graph.

[1] Currently the best known algorithm for matrix multiplication is due to Alman and Williams, where $\omega < 2.37286$ [4].

Definition 1.6 (Graph)**.** Let V be a finite set, and E be a collection of subsets of V. We call the pair $G = (V, E)$, a *graph*, when E contains only subsets of size two, i.e.

$$E \subseteq \{\{v, u\} \mid v, u \in V\}$$

We will be referring to the elements of V, as *vertices* or *nodes*. On the other hand, we call the elements of E *edges*. Often, we will denote the set of vertices $V(G)$ and the set of edges $E(G)$, for a given graph G.

Let $G = (V, E)$ be a graph and $v \in V$ some node, we call *neighborhood* of v, the set of vertices $N(v)$ that are adjacent to v. We call the number $d(v)$ of the neighbors of v, *degree* of v. Similarly, we denote with $\partial(v)$ the *set of edges that are incident* to v. An elementary result that connects the sum of degrees of a graph to the number of edges is known as *Handshake Lemma*.

Lemma 1.7 (Handshake Lemma)**.** Let $G = (V, E)$ be a graph, then the following is true.

$$\sum_{v \in V} d(v) = 2|E|$$

Let $H = (V', E')$ be another graph, with $V' \subseteq V$ and $E' \subseteq E$. We call H a *subgraph* of G and denote it with $H \subseteq G$. A special case of subgraphs are the *induced* graphs. Let $V'' \subseteq V$ be a subset of vertices, then we denote with $G[V'']$ the graph $G'' = (V'', E'')$, where E'' is the maximum subset of edges of E, that have *both* endpoint in V''. The *complementary graph* $\overline{G} = (V, \overline{E})$, of some graph $G = (V, E)$, is the graph that contains all the edges that do not belong in E. Lastly, let $F = (\widetilde{V}, \widetilde{E})$ be a graph. We say that F, G are *isomorphic* if and only if there is some bijection $\phi \colon \widetilde{V} \to V$, such that $\{v, u\}$ is an edge of G if and only if $\{\phi(v), \phi(u)\}$ is an edge of F.

We now define some elementary classes of graphs. Let $G = (V, E)$, where we can write the vertices of V as a sequence $v_1, v_2, \dots, v_n$ where the only edges are between v_i, v_{i+1} for $i \in [n-1]$. We call this class of graphs a *path* , and we will often denote it with P_n or simply P. On the other hand, let's assume that there is the sequence $v_1, v_2, \dots, v_n$ of the vertices, as defined before; but also v_1, v_n are adjacent. We call this type of graph a *circle* or *circuit*, and denote it with C_n or simply C. Lastly, let $G = (V, E)$ be a *complete graph*, in the sense that for every $v_1, v_2 \in V$, with $v_1 \neq v_2$ we have $\{v_1, v_2\} \in E$. We will be calling such a graph, a *clique*. We denote the clique on n vertices with K_n.

Observe that the discrete structure of a graph lets us encode the very basic notion of relation, we just describe which elements of V are related. We provide no information regarding the nature of the relation. A more informative discrete structure, regarding the encoding of a relation is a *directed* graph.

Definition 1.8 (Directed Graph)**.** Let V be a finite set, and $A \subseteq V \times V$ a subset of pairs on V. We call the pair $D = (V, A)$, a *directed* graph on V. We call the elements of V nodes, while the elements of A *arcs*. Often, we will denote the set of vertices $V(D)$ and the set of arcs $A(D)$, for a given directed graph D.

All the notions we defined for the undirected graphs are transferred to directed graphs. Although, in directed graphs we have two notions of degree. Let $D = (V, A)$ and v some node of D. We call *in-degree* $d_{\text{in}}(v)$ of v the number of edges of the form (u, v), for some vertex u. Likewise we call *out-degree* $d_{\text{out}}(v)$ the number of edges of the form (v, u), for some vertex u. Similarly we define the notions of *in-neighborhood* $N_{\text{in}}(v)$ and *out-neighborhood* . Lastly, we denote with $\partial_{\text{in}}(v)$ the *ingoing* and with ∂_{out} the *outgoing* arcs of v.

Note that a directed graph D is essentially just a binary relation on the set of vertices V, see Definition 1.1. Since the binary relations are the main focus of this essay, the directed graphs provide a helpful model for our analysis. Observe that a directed graph allows us to express many important properties of a binary relation. One such example is the *reflexivity*, where we add a *loop* to our graph of the form (v, v) to the set of arcs. If the underlying relation is *symmetric*, then we force *parallel* arcs of the form $(v, u), (u, v)$. On the other hand, if the underlying relation is *antisymmetric*, then we disallow parallel arcs.

1.3 Posets

In this section we introduce the notion of a *partially ordered set*, or *poset*, which is the main focus of this paper. Firstly, we present the formal definition of the poset and other related terms. Then, we present two important results, due to Dilworth and Mirsky respectively.

Definition 1.9 (Poset)**.** Let $\mathcal{U}$ be a finite set. Also, let $\mathcal{R}_{\preceq} \subseteq \mathcal{U} \times \mathcal{U}$ be a *partial order*. We call the pair $\langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$ a *partially ordered set*, or *poset*.

Often, for two elements $x, y \in \mathcal{U}$, with $(x, y) \in \mathcal{R}_{\preceq}$, we will be using the *infix* notation, i.e. $x \preceq y$. When, for two elements $x, y \in \mathcal{U}$, we have $x \preceq y$, but $x \neq y$, we will write $x \prec y$. Also, when $x \preceq y$, we will often be saying that x is *smaller* than y, or x *precedes* y. On the other hand, if neither $x \preceq y$, nor $x \succeq y$, we call x, y *parallel*, and denote it with $x \parallel y$. Lastly, if two elements x, y are *related*, i.e. either $x \preceq y$, or $y \preceq x$, we write $x \sim y$. Note, that Definition 1.9 allows, multiple *minimal* or *maximal* elements to exist. We give a formal definition.

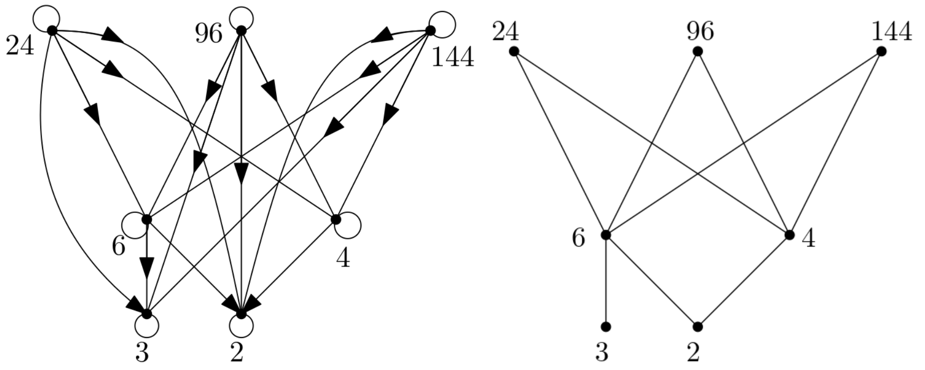
Definition 1.10 (Maximal/Minimal Element)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset on a finite set $\mathcal{U}$. Let $m \in \mathcal{U}$ be an element of the poset, where $m \preceq x$, for every $x \in \mathcal{U}$. We will call m a *minimal* element of $\mathcal{U}$, with respect to the relation $\preceq$. On the other hand, let $M \in \mathcal{U}$ be some element, such that $M \succeq x$, for every $x \in \mathcal{U}$. We call M *maximal* element of $\mathcal{U}$, with respect to the relation $\preceq$.

When the partial order $\preceq$ is clear from context, we will just say that m is a minimal element of $\mathcal{U}$. Similarly, for a maximal element M of $\mathcal{U}$. If a partial order has a *single* minimal element, we call it *minimum*. Similarly, if a poset has a unique maximal element, we will be calling it *maximum*. .

If we restrict our attention to only pairwise related elements of $\mathcal{U}$, we enforce the behaviour of a total order. On the other hand, we can define subsets of $\mathcal{U}$ where no element relates to another. We give the following definition.

Definition 1.11 (Chain/Antichain)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset on a finite set $\mathcal{U}$. Also, let $S \subseteq \mathcal{U}$ be a subset of elements, where for every $x, y \in S$, we either $x \preceq y$, or $y \preceq x$. We will be calling the set S a *chain* of the poset $\langle \mathcal{U}, \preceq \rangle$. On the other hand, we will calling a subset $T \subseteq \mathcal{U}$, an *antichain*, if for every $x, y \in T$, neither $x \preceq y$, nor $y \preceq x$.

Observe that, since the chains and anticahins are sets, we can (partially) order them with the subset relation $(\subseteq)$ and define a *maximal chain* and a *maximal antichain* of



(a) The directed graph representation of the poset of Example 1.12.

(b) The Hasse diagram representation of the poset of Example 1.12.

Figure 1.1: Different representations of the partially ordered set of Example 1.12.

a poset. Note that, in general, two maximal chains will not be of the same size. We call the maximal chain with the greater cardinality, a *maximum* chain. Similarly, for the *maximum* antichain. For a poset $\langle \mathcal{U}, \preceq \rangle$, we will denote with height$(\mathcal{U}, \preceq)$ the number of elements in a maximum chain. On the other hand, we denote with width$(\mathcal{U}, \preceq)$ the number of elements in a maximum antichain.

1.3.1 Graphical Representation of a Poset

There are many ways to represent a poset. Perhaps the most straightforward way is with a directed graph. Let $D = (V, A)$ be a directed graph, where $V = \mathcal{U}$ and $A = \mathcal{R}_{\preceq}$, where $\langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$ be a partial relation. Observe that for every vertex $v \in V$, of the directed graph D, we would have a loop $(v, v) \in A$, from reflexivity. On the other hand, we wouldn't have any parallel arcs, due to antisymmetry. Lastly, for every two arcs of the form $(x, y), (y, z) \in A$ we would have the arc $(x, z) \in A$, due to transitivity.

Observe that encoding a poset with a directed graph contains redundancy. Since, every vertex will have a loop we can omit these arcs. Also, we may omit the transitive arcs, since we can deduce them from context. Lastly, if we draw our graph carefully, in order to draw a vertex x higher than y, if $(y, x) \in A$. This way, we may "forget" the directions of the edges. We will call this type of "graph" a *Hasse diagram*.

Example 1.12. We follow the example of [5]. Let $\mathcal{U} = \{2, 3, 4, 6, 24, 96, 144\}$. We define the poset $\langle \mathcal{U}, | \rangle$, where $|$ is the *"divides"* relation, i.e. $a \mid b$ if there is a number c, such that $b = c \cdot a$. Observe that $|$ is a patial order, since it's reflective, antisymmetric and transitive. In Figure 1.1 we give the two different representations of a poset, as a directed graph and as a Hasse diagram. Observe that $2, 3$ are minimal elements, while $24, 96, 144$ are maximal. Moreover, the elements $6, 4$ are parallel $6 \parallel 4$. It is easy to see that all minimal and all the maximal elements will always be parallel in a poset.

1.3.2 k-Smallest Elements

We would like to generalize the notion of minimal elements of Definition 1.10 and talk about the k*-smallest elements* of a poset. We will provide a recursive, constructive definition and a "declarative", non-constructive definition. Before we do that, we introduce the notion of a *subposet*. We give the following definition.

Definition 1.13 (Subposet)**.** Let $P = \langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$ be a poset on a finite set $\mathcal{U}$, and $\mathcal{R}_{\preceq} \subseteq \mathcal{U} \times \mathcal{U}$ be the partial order. Also, let $\mathcal{V} \subseteq \mathcal{U}$ be a a subset of our universe. We call subposet, *induced* on the elements of $\mathcal{V}$ the poset $P' = \langle \mathcal{V}, \mathcal{R}'_{\preceq} \rangle$ where,

$$\mathcal{R}'_{\preceq} = \{(v_1, v_2) \in \mathcal{V} \times \mathcal{V} \mid (v_1, v_2) \in \mathcal{R}_{\preceq}\}.$$

We $\mathcal{R}'_{\preceq}$ the *restriction* of $\mathcal{R}_{\preceq}$ on $\mathcal{V}$, and we will denote it with $\mathcal{R}'_{\preceq} = \mathcal{R}_{\preceq} \mid \mathcal{V}$. When its clear from the context, we will abuse the notation and denote with $\langle \mathcal{V}, \preceq \rangle$ the subposet of $\langle \mathcal{U}, \preceq \rangle$ to ease the formalization.

Note that the notion of subposet its symmetrical to the notion of the *induced graph*. If we represent a poset $\langle \mathcal{U}, \preceq \rangle$ with a directed graph $D = (\mathcal{U}, E)$, where the edges represent the $\preceq$ relation, then $G[\mathcal{V}]$ will be the representation of the subposet $\langle \mathcal{V}, \preceq \rangle$. We now introduce the notion of the k-smallest elements of a poset, which will formalize and extent our intuition about the k smallest elements in a totally ordered set. We give the recursive, constructive definition firstly.

Definition 1.14 (k-Smallest Elements, Recursive Definition)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset on a finite set $\mathcal{U}$. Also, let be $k \geq 1$ be a natural number. With minimal$(\mathcal{U}, \preceq)$ we denote the *minimal elements* of the poset $\langle \mathcal{U}, \preceq \rangle$. We define the set S_k of the k smallest elements recursively. In the following definition, $\mathcal{U}_1 = \mathcal{U}$ and $\mathcal{U}_k = \mathcal{U}_{k-1} \setminus$ minimal$(\mathcal{U}_{k-1}, \preceq)$.

$$\left.\begin{array}{rl} S_1 = & \text{minimal}(\mathcal{U}_1, \preceq) \\ S_k = & \text{minimal}(\mathcal{U}_{k-1}, \preceq) \end{array}\right\} \tag{1.2}$$

Definition 1.14 extends Definition 1.10; S_1 will contain the *minimal* elements of the poset. Note that Definition 1.14 is essentially an algorithm for computing the k-smallest elements. In each iteration, we compute the minimal elements and remove the from the poset. We repeat this process k times. The set containing the k smallest elements will consist of the minimal elements of the *last* iteration. Intuitively, we "prune" the poset, from the bottom up. The process of Definition 1.14 will be used in the proofs of Theorems 1.16 and 1.17 in the sequel. Additionally, we will see some algorithms based in the above definition in Chapter 4.

For the non-constructive definition, we introduce the notion of the *height* of an element $u \in \mathcal{U}$, for a poset $\langle \mathcal{U}, \preceq \rangle$. Consider such an element $u \in \mathcal{U}$, let $C \subseteq \mathcal{U}$ be a *maximal* chain in $\langle \mathcal{U}, \preceq \rangle$, where for each $v \in C$, we have $u \succ v$. We denote with height$(u) = |C|$ *the height of* u *in* $\langle \mathcal{U}, \preceq \rangle$. We now proceed with the declarative definition.

Definition 1.15 (k-Smallest Elements, Declarative Definition)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset on a finite set $\mathcal{U}$. Also, let be $k \geq 1$ be a natural number. We denote with S'_k the k *smallest elements* in $\langle \mathcal{U}, \preceq \rangle$ where,

$$S'_k = \{u \in \mathcal{U} \mid \text{height}(u) = k - 1\}. \tag{1.3}$$

Note that we demand the elements of S'_k to have height $k - 1$, because we want S_1 to contain the *minimal* elements. The minimal elements of a poset have zero height, i.e. height$(u) = 0$. Lastly, observe that the sets S_k, S'_k of the above definitions contain the same elements. We can prove this fact by induction on k. If $u \in S_k$ of Definition 1.14, then there is at least one element v in $\mathcal{U}_k$ such that $u \succ v$. Otherwise, u would be

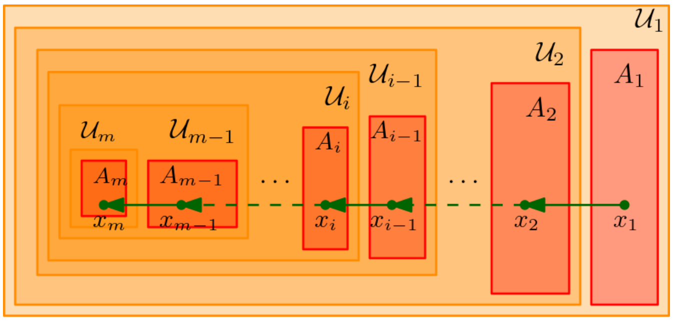


Figure 1.2: The construction of a chain for the proof of Theorem 1.16.

in S_{k-1}. Hence, $\text{height}(u) = \text{height}(v) + 1$. From the inductive hypothesis, we have that $S_k \subseteq S'_k$. For the other direction, note that the maximal chain C, of an element u, with height $\text{height}(u) = k - 1$, will contain at least one element of each of the sets $S_1, S_2, \dots, S_{k-1}$.

1.3.3 Mirsky's and Dilworth's Theorems

We are now on our way to prove the first important theorem in posets. In particular we will show two dual theorems, the first is due to Mirsky (1971) [6], while the second due to Dilworth (1950) [7]. Here we follow the presentation of [8] and give the most recent result first since it's much easier to prove.

Theorem 1.16 (Mirsky, 1971)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset. If $\text{height}(\mathcal{U}, \preceq) = n$, then, there is a partition of *antichains* $\mathcal{U} = A_1 \cup A_2 \cup \dots \cup A_n$.

Proof. We first construct a sequence of disjoint antichains A_i. We construct this sequence recursively. Let M_0 be the set of all maximal elements of $\mathcal{U}$. We set $A_1 = M_0$ and $\mathcal{U}_1 = \mathcal{U} \setminus M_0$. If $\mathcal{U}_i$ the remaining elements in some level of this recursive process, let M_i be the set of maximal elements of the poset $\langle \mathcal{U}_i, \preceq \rangle$. Set $A_{i+1} = M_i$ and $\mathcal{U}_{i+1} = \mathcal{U}_i \setminus M_i$. Since, $\mathcal{U}$ if finite, this process terminates after m steps and constructs a decomposition of antichains $\mathcal{U} = A_1 \cup A_2 \cup \dots \cup A_m$. It remains to show that $m = n$.

Note that $m \geq n$ is trivial, since any partition of $\mathcal{U}$ into antichains requires at least n antichains. We need only to show that $m \leq n$. We will construct a chain of length m, since n is the size of the greatest chain we will have proved the claim. We construct this chain recursively (see Figure 1.2). Let $x_m \in A_m$. We construct the chain of the form $x_i \succeq x_{i+1} \succeq \dots x_{m-1} \succeq x_m$, where $x_i \in A_i$. Then, there exists some element $x_{i-1} \in A_{i-1}$, where $x_{i-1} \succeq x_i$. Else, x_i would be a maximal element of $\mathcal{U}_{i-1}$. A contradiction, since $x_i \notin A_{i-1}$. There there is a chain of the form $x_1 \succeq x_2 \succeq \dots \succeq x_m$, and $m \geq n$.

□

For the theorem of Dilworth we introduce some notation. Let $\langle \mathcal{U}, \preceq \rangle$ be a poset and some element $x \in \mathcal{U}$. We denote with $L(x)$ the elements of $\mathcal{U}$, that *precede* x, i.e. $L(x) = \{y \in \mathcal{U} \mid y \prec x\}$, while $L[x] = L(x) \cup \{x\}$. On the other hand, we denote with $U(x)$ the elements of $\mathcal{U}$, that *succeed* x, i.e. $U(x) = \{y \in \mathcal{U} \mid x \prec y\}$, while $U[x] = U(x) \cup \{x\}$. Also, we denote with $I(x)$ the elements that are *parallel* to x, i.e. $I(x) = \{y \in \mathcal{U} \mid x \parallel y\}$. Additionally, we expand the above notations on subsets $S \subseteq \mathcal{U}$. With $L(S)$ we denote all the elements that precede some element of S.

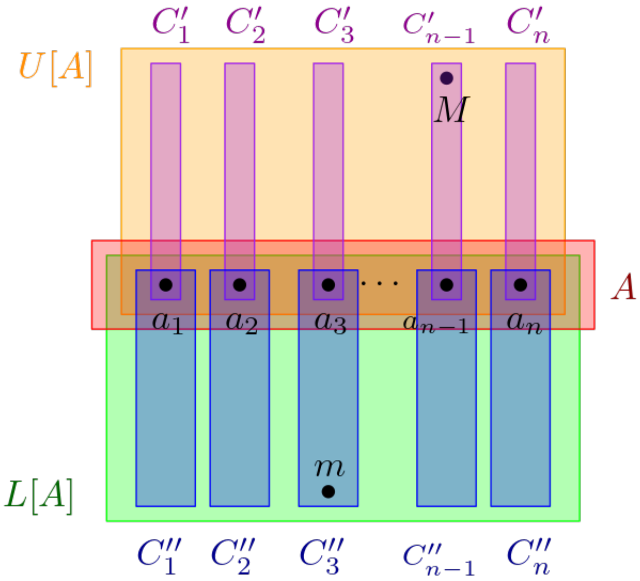


Figure 1.3: The construction of a chain partition for Theorem 1.17.

While with $U(S)$ we denote all elements that succeed some element of S. Lastly, with $L[S], U[S]$, we denote the sets $L[S] = L(S) \cup S$, and $U[S] = U(S) \cup S$, respectively.

We are now ready to prove present Dilworth's theorem.

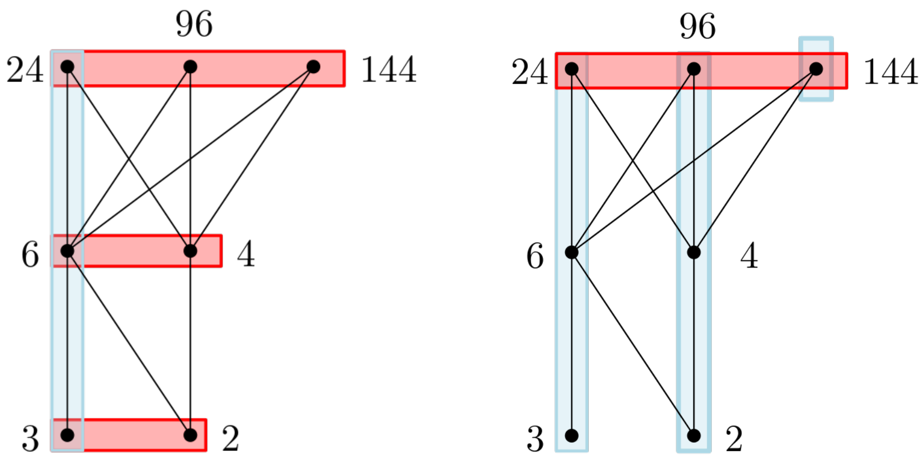
Theorem 1.17 (Dilworth, 1950)**.** Let $\langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$ be a poset. If $\text{width}(\mathcal{U}, \mathcal{R}_{\preceq}) = n$, then there is a partition of *chains* $\mathcal{U} = A_1 \cup A_2 \cup \cdots \cup A_n$.

Proof. We proceed by induction to the size of the universe $|\mathcal{U}|$. The result is trivial for $|\mathcal{U}| = 1$. We assume that the hypothesis holds for $|\mathcal{U}| \leq k$, and we prove the inductive step for $|\mathcal{U}| = k + 1$. Without loss of generality we assume that $\text{width}(\mathcal{U}, \preceq) > 1$; otherwise we have the trivial partition $\mathcal{U} = A_1$. Furthermore, let C be a non empty chain, then we may assume that the subposet $\langle \mathcal{U} \setminus C, \mathcal{R}_{\preceq}[\mathcal{U} \setminus C] \rangle$ has width *at least* $n-1$. To see this, let's assume for sake of contradiction that $\text{width}(\langle \mathcal{U} \setminus C, \mathcal{R}_{\preceq}[\mathcal{U} \setminus C] \rangle) = m < n-1$. Form the inductive hypothesis, there is a partition of $\mathcal{U} \setminus C = C_1 \cup C_2 \cup \cdots \cup C_m$. Then, $\mathcal{U} = C \cup C_1 \cup \cdots \cup C_m$ is a partition of $\mathcal{U}$ into $m + 1 < n$ chains. Hence, we arrive to a contradiction, since any partition of $\mathcal{U}$ in chains, should contains *at least* n chains. On the other hand, if the subposet $\langle \mathcal{U} \setminus C, \mathcal{R}_{\preceq}[\mathcal{U} \setminus C] \rangle$ has width *exactly* $n - 1$, then, from the inductive hypothesis the claim is proved. Therefore, henceforth we assume that for some non empty chain C, the subposet $\langle \mathcal{U} \setminus C, \mathcal{R}_{\preceq}[\mathcal{U} \setminus C] \rangle$ has width n.

If there is some *isolated* element $x \in X$, i.e. $I(x) = \mathcal{U} \setminus \{x\}$, then the subposet $\langle \mathcal{U} \setminus \{x\}, \mathcal{R}_{\preceq}[\mathcal{U} \setminus \{x\}] \rangle$ has width $n-1$. From the above argument, we have the desired partition, from the inductive hypothesis. Hence, we assume that there are no isolated elements.

Choose a minimal point m, and a maximal point M with $m \prec M$. The set $C = \{m, M\}$ is a chain. Let $Y = \mathcal{U} \setminus C$ and $Q = \mathcal{R}_{\preceq}[Y]$. Now, $\text{width}(Y, Q) = n$, hence $\langle Y, Q \rangle$ contains an *antichain* $A = \{a_1, a_2, \ldots, a_n\}$ of size n. Note that $U[A] \neq \mathcal{U}$, since $m \notin U[A]$, and $L[A] \neq \mathcal{U}$, since $M \notin L[A]$. Also, note that $L[A] \cap U[A] = A$.

By the inductive hypothesis, we know that we can partition each of $U[A]$ and $L[A]$



(a) Mirky's Theorem. A decomposition of the posets in antichains.

(b) Dilworth's Theorem. A decomposition of the poset in chains.

Figure 1.4: The figures of Example 1.18. The application of Mirksy's and Dilworth's theorems in the poset of Example 1.12.

into chains, i.e. $U[A] = C'_1 \cup \cdots \cup C'_n$ and $L[A] = C''_1 \cup \cdots \cup C''_n$. Also, without loss of generality $a_i \in C'_i \cap C''_i$. This, implies that $\mathcal{U} = (C'_1 \cup C''_1) \cup \cdots \cup (C'_n \cup C''_n)$ is the desired partition.

□

In the next example we show a chain and antichain decomposition of the poset of Example 1.12.

Example 1.18. We give an example of the poset $\langle \mathcal{U}, | \rangle$ of Example 1.12, where $\mathcal{U} = \{2, 3, 4, 6, 24, 96, 144\}$ and $|$ is the *divides* relation. Observe Figure 1.4a. There, we have a chain $C = \{3, 6, 24\}$, of size three, and a decomposition of the poset in three chains. On the other hand, in Figure 1.4b, we have an antichain $A = \{24, 96, 144\}$, of size three and a decomposition of the posets in three chains. Note that in general we wouldn't have $\text{width}(\mathcal{U}, \preceq) \neq \text{height}(\mathcal{U}, \preceq)$. In this example, $\text{width}(\mathcal{U}, \preceq) = \text{height}(\mathcal{U}, \preceq) = 3$. Observe the duality of Mirsky's and Dilworth's theorems.

1.4 Sorting and Selection Problems

In this section we will introduce the main computational problems that we will be considering *sorting* and *selection*. We will present these problem in the setting of total orders, in order to prepare the reader for the generalized version on partial orders, that we will examine in Chapters 3, 4, and 5. Moreover we introduce *query complexity* as a concept of efficiency.

Definition 1.19 (Query Complexity)**.** Consider a finite set $\mathcal{U}$. Let also, $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\prec\}$ be a binary function on $\mathcal{U}$ that *characterizes* an *ordering relation* (partial or total) $\mathcal{R}_{\preceq} \subseteq \mathcal{U} \times \mathcal{U}$ on $\mathcal{U}$. We call $c(\cdot, \cdot)$ an *oracle function* of an underlying ordering relation $\mathcal{R}_{\preceq} \subseteq \mathcal{U} \times \mathcal{U}$. Additionally, consider an algorithm $\mathcal{A}$ that *cannot* compare two elements of $\mathcal{U}$ directly, but uses the oracle function $c(\cdot, \cdot)$. Let $q(n)$ be the *number* *of calls to the oracle* $c(\cdot,\cdot)$, in an input instance of size n. We call $q(n)$ the *query complexity* of algorithm $\mathcal{A}$.

The notions of the *oracle function* $c(\cdot,\cdot)$ and the *query complexity* are at the core of this thesis. For the examined problems, we will try to present query-optimal algorithms and explore the time-complexity query-complexity trade-off that will frequently appear. Also, note that for an algorithm $\mathcal{A}$, if $t(n)$ and $q(n)$ denote its time and query complexity respectively, we will always have $t(n) \geq q(n)$. To understand this remember that in the traditional time-complexity setting we count the overall operation of an algorithm on an input instance of size n, *including* the calls to the oracle $c(\cdot,\cdot)$. Therefore, each lower bound to the query complexity of an algorithm, will be a lower bound to its time complexity. The relation between the time complexity $t(n)$ and the query complexity $q(n)$ will be more appeared in Chapter 3, where we will describe a *query-optimal* algorithm, that has *exponential* time complexity, $t(n) \gg q(n)$.

1.4.1 Sorting in Totally Ordered Sets

In this subsection we rethinking the traditional problem of sorting a list of integers in a setting of an oracle function. Imagine we are given a *totally ordered* set S, and an *oracle function* $c(\cdot,\cdot)$. For two elements $x, y \in S$ the oracle $c(x, y)$ will return $\leq$ if $x \leq y$, or $>$ if $x > y$. Our goal is to deduce the total order $\mathcal{R}_{\leq}$. Moreover we want to *minimize* the number of calls to the oracle $c(\cdot,\cdot)$.

Sorting on Totally Ordered Set	
Input:	1. A totally ordered set S. 2. An oracle function $c\colon S \times S \to \{\leq, >\}$.
Output:	The total order relation $\mathcal{R}_{\leq}$.

As it turns out the well known *Merge Sort* [9] is an optimal algorithm, *with respect to oracle calls*. It is easy to see, from the way Merge Sort operates, that in each step it makes a single comparison, or in our setting a single call to the oracle function. Hence, it makes $O(n \log n)$ total calls, where $n = |S|$. In the next theorem we prove that *every* sorting algorithm, in totally ordered sets, must perform $\Omega(n \log n)$ oracle calls.

Theorem 1.20. Let $\mathcal{U}$ a totally ordered set, and $c\colon \mathcal{U} \times \mathcal{U} \to \{\leq, >\}$ an oracle function for the (unknown) total order of S. Also, consider an algorithm that deduces the underlying total order $\mathcal{R}_{\leq}$, and can access the elements of $\mathcal{U}$ *only* through the oracle function. Such an algorithm must perform $\Omega(n \log n)$ oracle calls.

Proof. Consider a binary tree, where each internal node correspond to a call to the oracle $c(x, y)$. The execution of an algorithm will correspond to a root-leaf path. Each leaf will contain the result of the algorithm, namely the correct (sorted) permutation of the elements of $\mathcal{U}$. Depending on the choice of the underlying ordering each of the permutation may appear as a possible outcome of the algorithm. Hence, if $|\mathcal{U}| = n$, our tree will have $n!$ leaves. Since a tree of height h has 2^h leaves we have,

$$\begin{aligned}
2^h \geq n! \Rightarrow h &\geq \log(n!) \\
&= \log(n(n-1)(n-2)\cdots(2)) \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

□

Another way to see the above fact is from an information theoretic perspective. Since each of the $n!$ permutations, uniquely identifying the permutation requires $\log n! = \Omega(n \log n)$ bits, each comparison yields a single output bit. This way we obtain the desired lower bound. For a more in depth presentation of Information Theory and its connection to posets we refer to [10, 11]. In Chapters 3 and 4, 5 we will prove similar lower bounds for oracle models in posets.

1.4.2 Selection in Totally Ordered Sets

The other problem that we will examine in this thesis is about finding the smallest or the k-th smallest elements in partially ordered sets. These problems known as *Selection* and *k-Selection* problems respectively.

Selection on Totally Ordered Set	
Input:	1. A totally ordered set S. 2. An oracle function $c\colon S \times S \to \{\leq, >\}$.
Output:	The minimum element m.

k-Selection on Totally Ordered Set	
Input:	1. A totally ordered set S. 2. An oracle function $c\colon S \times S \to \{\leq, >\}$.
Output:	The k-th smallest element s.

There is a naive algorithm, that follows Definition 1.14 for the total orders, and solves the k-Selection problem in kn time. Just select the minimum element each time, and erase it from the input set. Repeat the process k times. Another straight forward approach would be to sort the input set in $n \log n$ time (and queries) and then select the desired element. Here we present a more sophisticated method due to Blum et al. (1973) [12]. The algorithm we present is known as *Median of Medians* and has time and query complexity of $O(n)$. The algorithm consists of two routines recursively calling each

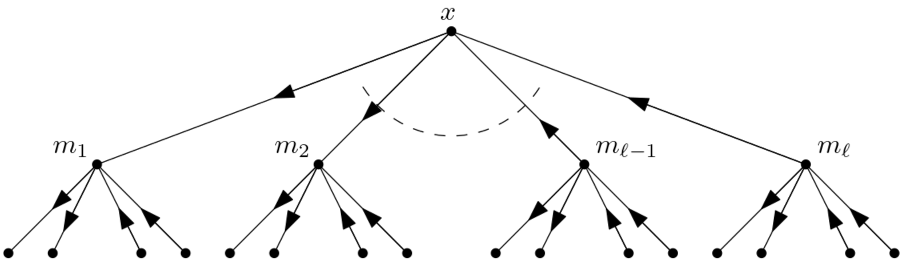


Figure 1.5: The structure of the *Median of Medians* algorithm for the k-Selection problem.

other. We assume at first that there exists a routine that provides us with a *"good"* guess of the median of the input S. Let x be our guess of the median of S. We will partition S in two sets $S_{\leq}, S_{>}$ containing the elements of S that are less or equal to x, and greater than x, respectively. We would like that our partition be somewhat balanced. Observe the following algorithm.

```
Algorithm 1: Select
  Input: 1. A set $S$
         2. An oracle function $c\colon S \times S \to \{\leq, >\}$
         3. A number $k \in [n]$
  Output: The $k$-th smaller element of $S$
  $x \leftarrow$ GuessMedian($S, c(\cdot,\cdot)$)
  $S_{\leq} \leftarrow \varnothing, S_{>} \leftarrow \varnothing$
  for each $s \in S$ :
      if $c(s,x) = " \leq "$ then
          $S_{\leq} \leftarrow S_{\leq} \cup s$
      else
          $S_{>} \leftarrow S_{>} \cup s$
  if $|S_{\leq}| > k$ then
      return Select($S_{\leq}, c(\cdot,\cdot), k$)
  else
      return Select($S_{>}, c(\cdot,\cdot), k - |S_{\leq}|$)
```

Note that in Algorithm 1 we make $O(n)$ calls to oracle. In order to make a good guess of the median, we will divide S into groups of 5 elements. We find the median for each group. Then we find the *median of medians* of the groups and return it as our guess (see Figure 1.5). We will demand that our guess will partition S into $S_{\leq}, S_{>}$ with $|S_{\leq}|, |S_{>}| \leq 0.7n$. We will prove this claim in the sequel; now we formalize the above idea in the following algorithm.

Observe the key idea of the above routines. We use the routine `GuessMedian`, to implement `Select`, and vise versa. Note that $\mathcal{C}$ has $n/5$ elements, hence we have $n/5$ iterations of the for-loop n Algorithm 2. For each iteration we compute the exact median among 5 elements, in constant $5^2 = 25$ time and query complexity. Therefore, the time

```
Algorithm 2: GuessMedian
  Input: 1. A set $S$
         2. An oracle function $c\colon S \times S \to \{\leq, >\}$
  Output: A guess $x$ on the median of $S$, such that $|S_{\leq}|, |S_{>}| \leq 0.7n$.
  Partition $S$ in to groups of 5 elements, and collect them into
    $\mathcal{C} = \{C_1, C_2, \dots, C_\ell\}$.
  $B \leftarrow \varnothing$
  for each $C_i \in \mathcal{C}$ :
      Find $m_i$ the (exact) median of $C_i$, using the oracle $c(\cdot,\cdot)$.
      $B \leftarrow B \cup m_i$
  return Select$(B, c(\cdot,\cdot), n/10)$
```

and query complexity of Algorithm 2 will be $n/5 = O(n)$.[2] We now prove the core claim, regarding the quality of our guess of the median.

Claim 1.21. Let S a finite set, with $|S| = n$. Also, let x be the guess of Algorithm 2 about the median of S. We partition S to $S_{\leq} = \{s \in S \mid s \leq x\}$ and $S_{>} = \{s \in S \mid s > x\}$. We have $|S_{\leq}|, |S_{>}| \leq 0.7n$.

Proof. Let $B = \{m_1, m_2, \dots, m_\ell\}$ and $\mathcal{C} = \{C_1, C_2, \dots, C_\ell\}$ be the sets as defined in Algorithm 2. For each median of a group m_i, where $m_i \leq x$, C_i will contribute 3 elements to $S_{\leq}$. Similarly, each median m_i with $m_i > x$ will contribute 3 elements to $S_{>}$. Since x is the median of B, we will have $\frac{n}{10}$ m_i smaller than x and $\frac{n}{10}$ m_i greater than x. Hence, we have $|S_{\leq}|, |S_{>}| \geq \frac{3}{10}n$. Because, $|S_{\leq}| + |S_{>}| = n$, then $|S_{\leq}| = n - |S_{>}|$ and $|S_{\leq}| \leq \frac{7}{10}n$. Similarly, we get $|S_{>}| \leq \frac{7}{10}n$. □

From Claim 1.21 and the fact that `GuessMedian` makes $n/5$ iterations, we get the follow recursive formula for the time and query complexity of Algorithm 1.

$$T(n) \leq T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + \lambda n, \tag{1.4}$$

for some constant λ. Solving (1.4), by induction, will give us the expected $O(n)$ in time and query complexity. In Chapter 5 we will utilise the same philosophy in a sorting algorithm on posets.

A lower bound to the problem of k-Selection in total orders is given in Theorem 1.22, due to Fussenger and Gabow (1979) [13]. We present here this theorem without a proof and we refer to [13] for a full presentation. In Chapter 4 we use this theorem to sketch a proof for a lower bound for the k-Selection problem in *partial orders*.

Theorem 1.22. The number of queries required to find the set of the k smallest elements of an n-element *total order* is at least $n - k + \log\left(\binom{n}{k-1}/k\right)$

[2] Here, we assume that n is divisible by 10, we can raise this assumption by taking the floor $\lfloor n/10 \rfloor$, without harming our analysis. We avoid that, to ease our notation.

CHAPTER 2

ALGORITHMS IN FLOWS, MATCHINGS AND POSETS

In this chapter we explore the algorithmic perspective of Dilworth's theorem and unravel the surprising connection of posets to flow networks. In Section 2.1 we introduce the fundamental notions of *flows networks*, we also present the *Ford's and Fulkerson's algorithm* for finding an optimal flow. In Section 2.2 we define a new class of graphs, the *bipartite graphs* and discuss its connection to flow networks. We establish the notion of *matchings* in bipartite graphs and use Ford's and Fulkerson's algorithm to find a maximum matching. Lastly, in Section 2.3 we use the results of the previous section to provide an algorithmic proof of Dilworth's theorem. Our algorithm will use flow networks to compute a *minimum decomposition of a poset into chains*.

2.1 Networks and Flows

In this section we discuss flow networks and some related fundamental results. We define the notion of flow network and present the problem of *Maximum Flow*. We give Ford's and Fulkerson's algorithm [14] for solving this problem. We will also introduce the problem of finding a *Minimum Cut* in network. We discuss the duality of those problem and the fundamental *Maximum Flow Minimum Cut* theorem in the area.

2.1.1 Flow Networks

We begin with a definition of *flow networks*. A flow network is a directed graph where each arc has a *capacity* and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the arc. Also, there is a *single* node which produces flow, the *source*, and a single node which consumes it, the *sink*. The main sources for this section are [15, 16], and we refer the reader to these textbooks for a more in-depth presentation. We give the following formal definition.

Definition 2.1 (Flow Network). Let $D = (V, A)$ be a directed graph and $s, t \in V$ be two nodes called *source* and *sink*, respectively, with $d_{\text{in}}(s) = 0$ and $d_{\text{out}}(t) = 0$. Also, let $c\colon A \to \mathbb{R}_{\geq 0}$ the *capacity* of each edge. We call the tuple $\mathcal{N} = \langle D, s, t, c\rangle$ a *flow network*.

We next define the notion of flow, or flow function on a network. The flow function captures the intuition of a liquid flowing through a network of pipes. We give a formal definition.

Definition 2.2 (Flow)**.** Let $\mathcal{N} = \langle D, s, t, c \rangle$ be a flow network, on the directed graph $D = (V, A)$, with capacity function $c(\cdot)$, source $s \in V$ and sink $t \in V$. We call a *flow* on the network $\mathcal{N}$ a function $f \colon A \to \mathbb{R}_{\geq 0}$, which respects the following axioms.

1. (Capacity Condition) For each arc $a \in A$, we have $0 \leq f(a) \leq c(a)$.
2. (Conservation Condition[1]) For each node $v \in V \setminus \{s, t\}$, we have

$$\sum_{u \in N_{\text{in}}(v)} f(u, v) = \sum_{u \in N_{\text{out}}(v)} f(v, u) \tag{2.1}$$

For a network $\mathcal{N} = \langle D, s, t, c \rangle$ and a flow function $f \colon A \to \mathbb{R}_{\geq 0}$, we define the *value* of the flow, as

$$v(f) = \sum_{v \in N_{\text{out}}} f(s, v).$$

Note that from the second axiom in Definition 2.2, the above sum quantifies the total flow in our network. Intuitively, $v(f)$ will be the *volume* of the liquid flowing in our network. In *Maximum Flow Problem* we are required to find some valid flow function $f(\cdot)$ that maximizes $v(f)$.

Maximum Flow Problem	
Input:	A flow network $\mathcal{N} = \langle D, s, t, c \rangle$
Output:	A valid flow function $f \colon A \to \mathbb{R}_{\geq 0}$ respecting Definition 2.2, maximizing $v(f)$.

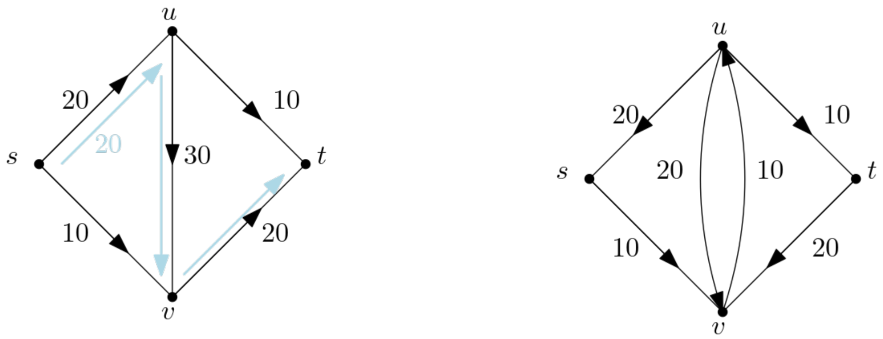
Before we present an algorithm for solving the above problem, we need to define the notion of a *residual network*.

Definition 2.3. Let $\mathcal{N} = \langle D, s, t, c \rangle$ be a flow network and $f \colon A \to \mathbb{R}_{\geq 0}$ be a valid flow function. We define the *residual network* $\mathcal{N}_f = \langle D_f, s, t, c_f \rangle$ as follows.

1. $V(D_f) = V(D)$, i.e. the set of the vertices remains the same.
2. For each arc $a = (u, v) \in A(D)$ with $f(a) < c(a)$, having $c(a) - f(a)$ *"residual"* units of flow, we add arc $a = (u, v)$ to $\mathcal{N}_f$ with capacity $c_f(a) = c(a) - f(a)$. We call these arc, *forward*. .
3. For each arc $a = (u, v) \in A(D)$ with $f(a) > 0$, there are $f(a)$ unit of flow that we can negate, driving flow in the opposite direction. Thus, we add the *backward* arc $a' = (v, u)$ to $\mathcal{N}_f$, with capacity $c_f(a') = f(a)$.

The notion of residual network captures the options we have in order to modify an existing flow. In fact, our algorithm will do exactly that; we will start with some initial flow f and in each iteration we will enhance our solutions, using the residual network. In Figure 2.1 we present a flow network. with an initial flow function and its residual network .

[1] Note the similarity with Kirchhoff's first law on electrical circuits; *"the algebraic sum of currents in a network of conductors meeting at a point is zero"*.



(a) A flow network. With light blue we denote the given flow f. We have $v(f) = 20$

(b) The residual network, for the given flow.

Figure 2.1: A network with an existing flow, on the left; and its residual network, on the right.

We are now ready to present Ford's and Fulkerson's algorithm. The algorithm will consists of two routines. The first routine `Augment` (presented in Algorithm 3) will take as input a give flow f and an *augmenting path* in the residual network. The routine will follow this path in order to increase the flow along its way. The second routine `Max-Flow` (presented in Algorithm 4), will just call `Augment` iteratively, while there are available augmenting path in the residual graph, while updating the flow function and computing the new residual network. The function $\texttt{bottleneck}(P, f)$ will return the minimum residual capacity along P.

```
Algorithm 3: Augment
  Input: 1. A residual network $\mathcal{N}_f$.
         2. The corresponding flow function $f$.
         3. a path $P \subseteq \mathcal{N}_f$.
  Output: An updated flow $f$, of increased value.
  $b \leftarrow \texttt{bottleneck}(P, f)$
  for each $a \in A(P)$ :
      if $a$ is a forward arc then
          $f(a) \leftarrow f(a) + b$
      else
          $f(a) \leftarrow f(a) - b$
  return $f$
```

A complete analysis of Ford's and Fulkerson's algorithm is beyond the scope of this thesis. Again we refer to [15, 16] for a more substantial analysis. Also, we will leave the discussion regarding the optimality of the solution for the next subsection. We note here that indeed the value of the flow is increased in each call of the `Augment` routine, since the flow can't exceed the total capacity $C = \sum_{a \in A} c(a)$, the algorithm will terminate after a finite number of steps. Regarding the complexity of the algorithm, we give the following theorem, without a proof.

```
Algorithm 4: Max-Flow
Input: A flow network $\mathcal{N} = \langle D, s, t, c \rangle$.
Output: A maximum flow for $\mathcal{N}$.
Initialize $f(a) \leftarrow 0$, for every arc $a \in A(D)$.
Compute the residual network $\mathcal{N}_f$.
while there is an augmenting $s, t$-path in $N_f$ do
    Let $P$ be an augmenting path.
    $f' \leftarrow$ Augment($\mathcal{N}_f, f, P$)
    $f \leftarrow f'$
    Compute the new residual network $\mathcal{N}_f$.
return $f$
```

Theorem 2.4. Let $\mathcal{N} = \langle D, s, t, c \rangle$ be a flow network. We also assume that the network has integer capacities, i.e. $c\colon A \to \mathbb{Z}_{\geq 0}$. Algorithm 4 computes a valid flow f in $O(mC)$ time.

We conclude this introduction to network flows with a remark regarding the integrality of the problem. Observe that, from the way Algorithm 4 works, if we have integer capacities, the resulting flow will also be integer. This fact is often referred to as *Integral Flow Theorem*; leaving the claim regarding the optimallity for the future, we give the following theorem. Again we omit any proof and constrain ourselves to an intuitive explanation, for a complete proof we refer to [16].

Theorem 2.5 (Integral Flow Theorem)**.** If the capacities of a network are integers, then exists an integral maximum flow.

2.1.2 Cuts in Networks

We continue this section, by introducing the notion of *cuts* in a network. Cuts will helps us define a more precise upper bound to the total flow of a network. We give the following definition.

Definition 2.6 (Cut)**.** Let $\mathcal{N} = \langle D, s, t, c \rangle$ be a flow network, on the directed graph $D = (V, A)$, with capacity function $c(\cdot)$, source $s \in V$ and sink $t \in V$. We call an s, t*-cut* a partition of the nodes in two sets S, T, with $s \in S$, $t \in T$, and $S \cup T = V$, while $S \cap T = \varnothing$. We denote an s, t-cut with (S, T), while we have for the capacity of the cut that,

$$c(S, T) = \sum_{a \in \partial_{\text{out}}(S)} c(a)$$

Note that an s, t-cut (S, T) *is a set of arcs* going from S to T. In the *Minimum Cut Problem* we are required to find an s, t-cut with the minimum capacity.

Minimum Cut Problem	
Input:	A flow network $\mathcal{N} = \langle D, s, t, c \rangle$.
Output:	A cut (S, T), with $S, T \subseteq V(D)$, and $s \in S, t \in T$ with the *minimal* capacity $c(S, T)$.

In Figure 2.2 we depict a network and the *minimum* s, t-cut. Observe that this cut is an upper bound to the value of the maximum flow $v(f) \leq c(S, T)$. In particular each

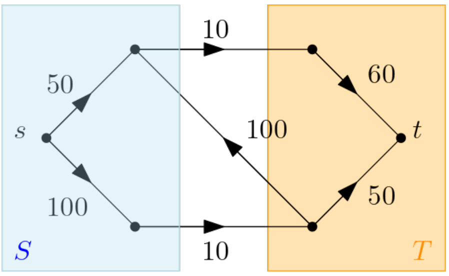


Figure 2.2: An example of a network cut (S, T), with capacity $c(S, T) = 20$.

cut, will be an upper bound to any feasible flow value. Hence, if v a feasible flow value and c a capacity of a cut in the same network, we have $v \leq c$. If $v^\star$ is a maximum flow, and $c^\star$ a minimum capacity, we get $v^\star \leq c^\star$. Evidently, the reverse inequality also holds. This fact is known as *Maximum-Flow Minimum-Cut* Theorem.

Theorem 2.7 (Maximum-Flow Minimum-Cut)**.** Let $\mathcal{N}$ be a a network. Also, let $v^\star$ be the value of a *maximum flow* and $c^\star$ be the capacity of a *minimum cut* on the same network $\mathcal{N}$. We have that,

$$v^\star = c^\star. \tag{2.2}$$

From Theorem 2.7 we get the optimality of Algorithm 4. Also, we can refine the computational complexity given by Theorem 2.4, with modifying the C, to be *the minimum s, t-cut*. We don't provide a formal proof of Theorem 2.7. For a full analysis we refer to [16]. We also note that with some prostprocessing, we can also get a minimum cut using Algorithm 4.

Theorem 2.8. Let $\mathcal{N} = \langle D, s, t, c \rangle$ be a flow network. Given a maximum flow function $f^\star$ we can compute a minimum cut in $O(m)$ time. Moreover, the set $S^\star$ of the minimum cut $(S^\star, T^\star)$, will be the subset of vertices $S^\star \subseteq V(D)$ which have *a path from the source s in the residual network $\mathcal{N}_f$.*

Before we conclude this brief discussion on network flows we give a Linear Programming argument, to support the Maximum Flow Minimum Cut Theorem. In actuality the Minimum Cut problem is a *dual linear program* of the Maximum Flow problem. Therefore, we can get the Equation 2.2 as an immediate consequence of the *Strong Duality Theorem of Linear Programming*. From linear programming we can also get a polynomial time algorithm for computing a maximum flow or a minimum cut, despite that we presented the classic Ford's and Fulkerson's algorithm as a more intuitive approach. In the following section, we will use network theory in order to prove Hall's and Königs Theorems, regarding bipartite graphs.

2.2 Matchings

We begin this section by defining another class of graphs, the *bipartite graphs*. We introduce here the bipartite graphs as *undirected* graphs but all of our observations are transferable in the directed graph setting. We give the following definition.

Definition 2.9 (Bipartite Graphs)**.** Let $G = (V, E)$ be a graph. We call G a *bipartite graph* is there is a partition of the nodes in two sets $X, Y \subseteq V$, with $X \cup Y = V$ and $X \cap Y = \varnothing$; but there are no edges inside $G[X]$ and $G[Y]$.

Let $G = (V, E)$ be a bipartite graph, with X, Y be a partition of the nodes respecting the conditions of Definition 2.9, we call X, Y the *parts* of the bipartite graph. We often use the notation $G = (X \uplus Y, E)$ to denote a bipartite graph with parts X, Y. A key property of bipartite graphs is that they cannot have *odd circles*. Observe that in a circle of a bipartite, the nodes will be of alternating parts of the graphs. Hence, in an odd circle we would have, necessarily two nodes edge connected of the same part. A contradiction. On the other hand, if we have only even circles in a graph, we can always partition its nodes in two parts[2]. Thus, we give the following proposition.

Proposition 2.10. A graph $G = (V, E)$ is a bipartite graph, *if and only if* has no odd circles.

We now define some key notions in bipartite graphs and graph theory. We call some set of nodes $I \subseteq V$ with no interior edges, namely the induced graph $G[I]$ has no edges, an *independent set* of G. Hence, in the above definition both A, B are independent sets of G. On the other hand, let $C \subseteq V$ be a set of nodes, such that for each edge e, either e is inside C, or is adjacent to some node of C. We call C a *vertex cover* of G. Observe that, if C is a vertex cover, then $V \setminus C$ is an independent set *and vice versa*. Again, both parts X and Y of a bipartite graph are vertex covers of the graph.

In the *Minimum Vertex Cover* problem, we are required to find a vertex cover of minimum size. For some graph G we denote the size of the minimum vertex cover with $\nu(G)$.

Minimum Vertex Cover	
Input:	A graph $G = (V, E)$.
Output:	A vertex cover $X \subseteq V$, of minimum size.

A core notion in bipartite graphs is a *matching*. A matching M is a subset of disjoint edges, i.e. for every $e_1, e_2 \in M$, we have $e_1 \cap e_2 = \varnothing$. In the Maximum Matching Problem, we are required to find a maximum matching in a bipartite graph. Let $U \subseteq V$ be a subset of vertices of a graph. We say that a matching M *covers* U, if for every vertex $v \in U$ there is some edges $e \in M$, such that $v \in e$. We say that a graph $G = (V, E)$ has a *perfect matching* if there is some matching $M^\star \subseteq E$ that covers V.

In the *Maximum Matching* problem, we are required to find a matching of maximum size. For some graph G we denote the size of a maximum matching with $\tau(G)$.

Maximum Matching	
Input:	A graph $G = (V, E)$.
Output:	A matching $M \subseteq E$, of maximum size.

Staying in the spirit of the previous section, we will unravel the connection between the Minimum Vertex Cover and Maximum Matching problems in bipartite graphs. We will see that $\nu(G) = \tau(G)$ in Königs Theorem. Before that, we will explore the connection of matching with flow networks, and prove the Hall's Theorem for perfect matching in bipartite graphs. In both cases, our proofs will be constructive and provide an algorithm for finding a maximum matching and a minimum vertex cover respectively. Our foundation will be Ford's and Fulkerson's Algorithm 4.

[2] We can "color" the nodes of each circle with alternating colors, e.g. red and blue. If we have only even circles, we will not have an edge with its endpoints having the same color. The parts of the bipartite graph, will be the color classes.

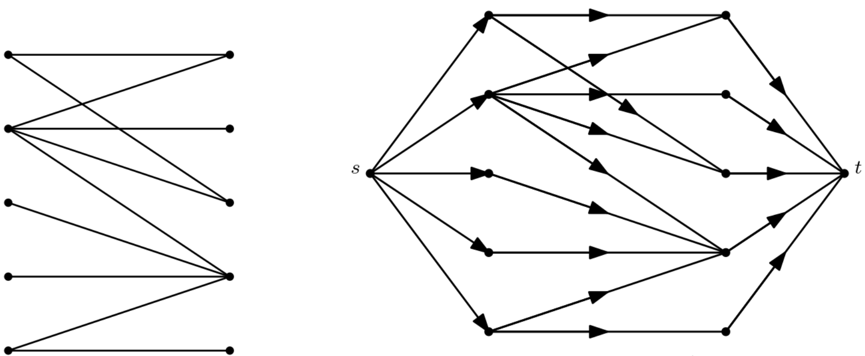
2.2.1 Hall's Theorem

Hall's Marriage Theorem[3] gives a sufficient and necessary condition for the existence of a perfect matching in bipartite graphs. There are a few ways to prove Hall's Theorem, here we follow the presentation of [15] and use the Maximum Flow Minimum Cut Theorem (Theorem 2.7)

Theorem 2.11 (Hall's Marriage Theorem, 1935)**.** Let $G = (X \uplus Y, E)$ be a bipartite graph, with $|X| = |Y|$, then there is a *perfect matching*, *if and only if* for every subset of vertices $A \subseteq X$, $|A| \leq |N(A)|$.

Before we give the proof of Theorem 2.11, we describe a reduction from the problem of Maximum Matching in Bipartite Graphs, to the problem of Maximum Flow in Networks. Starting from a bipartite graph $G = (X \uplus Y, E)$ we create a network $\mathcal{N}$ as depicted in Figure 2.3. Firstly, we orient all the edges of G from X to Y. Next, we add a new vertex s and an arc (s, x) for every vertex $x \in X$. Subsequently, we another new vertex t and an arc (y, t) for each vertex $y \in Y$. Lastly, we set the capacity of each arc of $\mathcal{N}$ equal to 1. From the construction of the network $\mathcal{N}$, the conservation condition 2.1, and the Integral Flow Theorem (see Theorem 2.5) we get the following Lemma.

Lemma 2.12. Let G a bipartite graph and $\mathcal{N}$ its corresponding network, then G has a maximum matching of size k, *if and only if* $\mathcal{N}$ has maximum flow of value k. Moreover, the edges in a maximum matching M are exactly the edges of non zero flow in $\mathcal{N}$.



(a) A bipartite graph G.

(b) The corresponding network $\mathcal{N}$ of the bipartite graph. All arcs have capacity of 1, and are oriented from left to right.

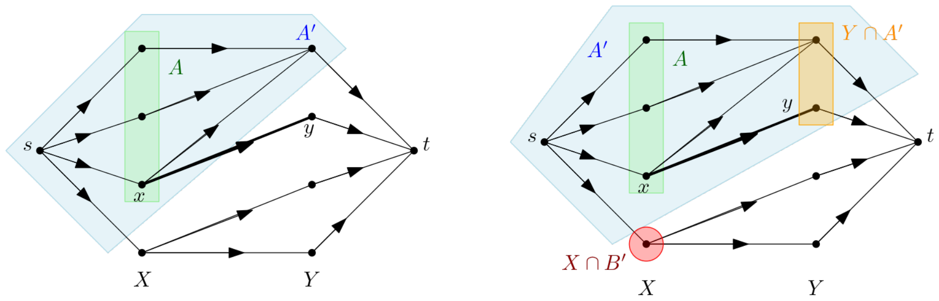
Figure 2.3: The reduction from a bipartite graph to a network.

We now present the proof of Theorem 2.11. Here, we follow the proof in [15].

Proof of Theorem 2.11. [$\Rightarrow$] We assume that there is a perfect matching in G. Then, naturally, for every $A \subseteq X$, we would have $|A| \leq |N(A)|$; otherwise there would always remain some element of A unmatched.

[$\Leftarrow$] For this direction, we assume that for every subset of nodes $A \subseteq X$, has at least $|A|$ neighbors. We prove that there is a perfect matching. From Lemma 2.12 it suffices to show that the corresponding network $\mathcal{N}$ admits maximum flow of value n.

[3] Proved by the English Mathematician Philip Hall in 1935 [17].



(a) A minimum cut. With light blue we denote set A', of the cut (A', B'). With light green we denote the set $A = A' \cap X$.

(b) The state of the cut after we include y in A'. With light orange we denote the set $Y \cap A'$, while with red we denote $X \cap B'$.

Figure 2.4: The network $\mathcal{N}$ corresponding to a bipartite graph $G = (X \uplus Y, E)$ of Theorem 2.11. With light blue we denote the first part of a minimum cut (A', B'). All the arcs have capacity of 1.

For sake of contradiction, we assume that the maximum flow is less than n, and we will find some set A, with $|N(A)| < |A|$.

From Maximum Flow Minimum Cut Theorem (Theorem 2.7) we know that there exist a minimum cut (A', B') with capacity less than n. Note that A' will contain the source s, and it could contain vertices from both X, Y. We set $A = X \cap X'$ and we will show that $|N(A)| < |A|$. We claim that we can modify the cut (A', B') in order to ensure that $N(A) \subseteq A'$. Let $y \in N(A)$ be a vertex that belongs to B' (see Figure 2.4a). We claim that moving y from B' to A' will *not* increase the cut's capacity. When moving y from B' to A' the arc (y, t) will traverse the cut, increasing the capacity by one. On the other hand, previously there would be at least one arc of the form (x, y), with $x \in A$, since $y \in N(A)$. Thus, at least one of the (x, y) arcs no longer traverse the cut. Therefore the total capacity of the cut cannot be increased.

We will examine now the capacity of the minimum cut (A', B'), where $N(A) \subseteq A'$ as depicted in Figure 2.4b. Since all the neighbors of A belong to A', we have that the only arcs leaving A', either are outgoing arcs of the source s, or are ingoing arcs to the sink t. Thus, for the capacity of the cut we have,

$$c(A', B') = |X \cap B'| + |Y \cap A'|. \tag{2.3}$$

Observe that $|X \cap B'| = n - |A|$ and $|Y \cap A'| \geq |N(A)|$. From our assumption that $c(A', B') < n$, we get,

$$n - |A| + |N(A)| \leq |X \cap B'| + |Y \cap A'| = c(A', B') < n. \tag{2.4}$$

Comparing the first and the last term of the above inequality we have that $|N(A)| > |A|$. □

Form Theorem 2.11 and the above discussion we get an algorithm for finding a maximum matching in a bipartite graph G. Firstly, we construct the corresponding network $\mathcal{N}$. We find a maximum flow on $\mathcal{N}$. A maximum matching M will include the edges e of G, that correspond to arcs a of $\mathcal{N}$ with $f(a) = 1$.

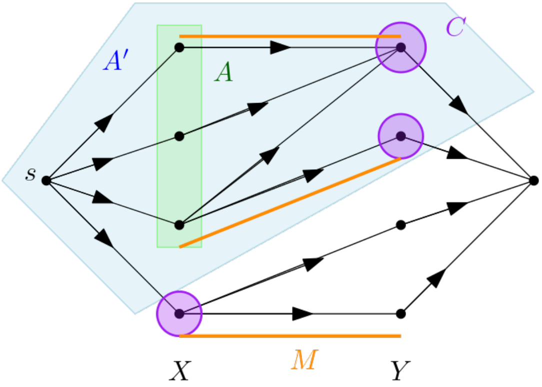


Figure 2.5: The network of König's Theorem 2.13. With dark orange we denote the edges of a maximum matching, while with purple we denote the vertices of a minimum vertex cover.

2.2.2 König's Theorem

In this subsection we will prove König's Theorem. We will build upon our work in the previous subsection, more precisely the proof of König's Theorem expands on the construction of Theorem 2.11, that relates bipartite graphs with flows.

Theorem 2.13 (König, 1931)**.** Let $G = (X \uplus Y, E)$ a bipartite graph. Also let $M \subseteq E$ a *maximum matching* and $C \subseteq V$ a *minimum vertex cover* on G. We have $|M| = |C|$, or

$$\nu(G) = \tau(G) \tag{2.5}$$

Proof. Consider the construction of Theorem 2.11. Let $\mathcal{N} = \langle D, s, t, c\rangle$ be the network corresponding to the to the bipartite graph G, and (A', B') a minimum cut. If $A = A' \cap X$ we assume that $N(A) \subseteq A'$. Let $M \subseteq E$ with,

$$M = \{(x, y) \in A(D[X \cup Y]) \mid f(x, y) = 1\}$$

for some maximum flow f. Note that in M we consider only the arcs that correspond to edges of the bipartite graph. We also define the set,

$$C = (X \cap B') \cup (Y \cap A')$$

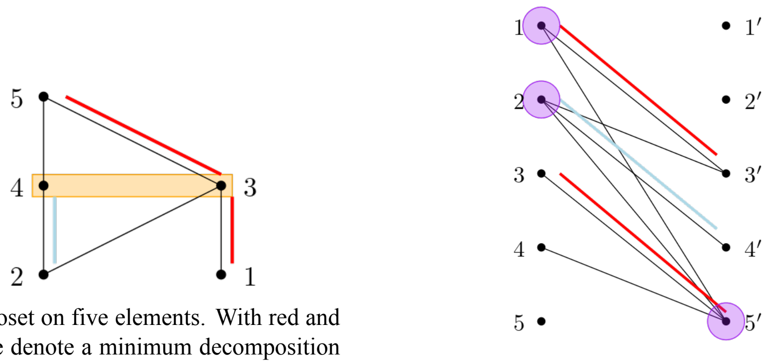
Since $N(A) \subseteq A'$ the minimum cut is only comprised of arcs going from s to $(X \cap B')$ and from $(Y \cap A')$ to t, see Figure 2.5. Hence, following Equation 2.3, we have,

$$|C| = |X \cap B'| + |Y \cap A'| = c(A', B') = |M|. \tag{2.6}$$

Lastly, observe that C is a vertex cover, as any arc that is not incident to vertices from $X \cap B'$ and $Y \cap A'$ must be incident to a pair of vertices from A to $B' \setminus A'$. A contradiction, since $N(A) \subseteq A'$.

□

Again, since we can use Ford's and Fulkerson's algorithm to obtain a maximum matching, with minimal post processing we can compute a minimum vertex cover. In the following section, we use Theorem 2.13 to compute a minimum decomposition of a poset into chains and prove Dilworth's Theorem.



(a) A poset on five elements. With red and blue we denote a minimum decomposition into two chains. With orange, we denote a maximum antichain.

(b) The corresponding bipartite graph of the poset on the left. With red and blue we denote the edges of a a maximum matching. With purple we denote a minimum vertex cover.

Figure 2.6: A poset on five elements and the corresponding bipartite graph.

2.3 Finding a Minimum Chain Decomposition

In this section we will re-prove Dilworth's theorem using Königs Theorem 2.13. Note that, since our prove of König's Theorem of the previous section was constructive, this new proofs of Dilworth's Theorem 1.17 will give us an *algorithm* to compute a minimum decomposition of a posets to chains. Additionally our algorithm will be able to find a maximum antichain. Our main source for this section is Ford's and Fulkerson's textbook [14].

We begin by proving a construction of a bipartite graph, from a given partially ordered set. Let $\langle \mathcal{U}, \preceq \rangle$ be a poset on some universe $\mathcal{U}$, $|\mathcal{U}| = n$. We construct the bipartite graph $G = (\mathcal{U} \uplus \mathcal{U}', E)$, where $\mathcal{U}'$ is a *copy* of our universe $\mathcal{U}$. If $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$ we denote the elements of $\mathcal{U}'$ as $\mathcal{U}' = \{u'_1, u'_2, \ldots, u'_n\}$. For some $u_i \in \mathcal{U}$ and $u'_j \in \mathcal{U}$, with $i \neq j$ we have $\{u_i, u'_j\} \in E$ if $u_i \preceq u'_j$. See Figure 2.6, we will explore the connection of chains. antichains, matching and vertex covers in the sequel.

We proceed to our first Lemma connecting the size of a matching to a decomposition into chains of a poset.

Lemma 2.14. Let $\langle \mathcal{U}, \preceq \rangle$ be a partially ordered set, with $|\mathcal{U}| = n$, and $G = (\mathcal{U} \uplus \mathcal{U}', E)$ be the corresponding bipartite graph. Also, let $M \subseteq E$ a matching of G. Then, there exists a decomposition Δ of $\langle \mathcal{U}, \preceq \rangle$ into chains, such that,

$$|M| + |\Delta| = n. \tag{2.7}$$

Proof. Let M be the matching, where

$$M = \{\{u_{i_1}, u'_{j_1}\}, \{u_{i_2}, u'_{j_2}\}, \ldots, \{u_{i_n}, u'_{j_n}\}\}.$$

Thus,

$$i_1 \preceq j_1, i_2 \preceq j_2, \ldots, i_n \preceq j_n.$$

We gather (”glue”) together the elements of $\{i_1, j_1, i_2, j_2, \ldots, i_n, j_n\}$ into chains greedily[4]. Let Δ be the resulting decomposition. If there is an element $k \in \mathcal{U}$ that does not belong to $\{i_1, j_1, i_2, j_2, \ldots, i_n, j_n\}$, we add it to Δ as a *singleton* chain. We denote the chains of Δ with ℓ_j. We observe that,

$$n = \sum_{j=1}^{|\Delta|} |\ell_j| = \sum_{j=1}^{|\Delta|} (\ell_j - 1) + |\Delta| \overset{(\star)}{=} |M| + |\Delta|$$

Where the equation $(\star)$ holds, because a path of ℓ vertices has $\ell - 1$ edges.

□

We can observe an example of Lemma 2.14 in Figure 2.6. Note that in the poset of Figure 2.6a we have a minimum chain decomposition of size two. On the other hand, in the bipartite graph of Figure 2.6b there is a maximum matching of size three. Lastly, note that in the proof of Lemma 2.14 we would glue together the edges $\{3, 5'\}$ and $\{1, 3'\}$ of Figure 2.6b. We proceed to our next Lemma connecting the size of a maximum antichain and a minimum vertex cover.

Lemma 2.15. Let $\langle \mathcal{U}, \preceq \rangle$ be a partially ordered set, with $|\mathcal{U}| = n$, and $G = (\mathcal{U} \uplus \mathcal{U}', E)$ be the corresponding bipartite graph. Also, let $X \subseteq \mathcal{U} \cup \mathcal{U}'$ be a *proper*[5] vertex cover, and $U \subseteq \mathcal{U}$ be an antichain of $\langle \mathcal{U}, \preceq \rangle$. Then it holds that,

$$|X| + |U| = n. \tag{2.8}$$

Proof. Let $\phi \colon \mathcal{U} \cup \mathcal{U}' \to \mathcal{U}$ be a function that maps the nodes of the bipartite graph $G = (\mathcal{U} \uplus \mathcal{U}, E)$ to the corresponding elements of the poset $\langle \mathcal{U}, \preceq \rangle$. Also, let $X \subseteq \mathcal{U} \cup \mathcal{U}'$ be a proper vertex cover of the bipartite graph G, where

$$X = \{u_1, u_2, \ldots, u_\ell, u_1', u_2', \ldots, u_k'\}$$

We observe that the elements of X correspond to different elements of $\mathcal{U}$. In other words, there are no $x, y \in X$, where $\phi(x) = \phi(y)$. For sake of contradiction we assume that there are such elements $x, y \in X$, with $\phi(x) = \phi(y)$. Since X is a proper vertex cover, there is some $u' \notin X$, such that $\{u', x\} \in E$; similarly there is some $u \notin X$, such that $\{y, u\} \in E$. Because, $\phi(x) = \phi(y)$ and the transitivity property of the poset there is the edge $\{u', u\} \in E$ in the bipartite graph. Moreover, $\{u', u\}$ is not covered by X. A contradiction. Hence, all the elements of X are distinct.

Now, let $U = \mathcal{U} \setminus X$. Since X is a vertex cover, U would be an *independent* set of G. Thus, the elements of U would be *unrelated* in the poset $\langle \mathcal{U}, \preceq \rangle$. This concludes our proof.

□

We now present an alternative proof of Dilworth’s Theorem, based on the work of this chapter.

Theorem 2.16 (Dilworth, 1950)**.** Let $\langle \mathcal{U}, \mathcal{R}_\preceq \rangle$ be a poset. If width$(\mathcal{U}, \mathcal{R}_\preceq) = w$, then there is a partition of *chains* $\mathcal{U} = A_1 \cup A_2 \cup \cdots \cup A_w$.

[4] Note that the decomposition resulting from the ”gluing” of the mathching’s edges is *unique*. We can only ”glue” together edges of the form $\{u_1, u_2'\}, \{u_2', u_3\}$. If we stop the ”gluing” earlier we won’t be having a *decomposition*.

[5] Namely, $\mathcal{U} \setminus X \neq \varnothing$ and $\mathcal{U}' \setminus X \neq \varnothing$. Note that we can always assume a proper vertex cover, without loss of generality.

Proof. We get Dilworth's theorem as an immediate result of Lemmas 2.14, 2.15 and König's Theorem 2.13. Let $M^\star$ be a maximum matching, $X^\star$ be a minimum vertex cover. Also, let $\Delta^\star, U^\star$ be the corresponding sets of the poset $\langle \mathcal{U}, \preceq \rangle$. We would have $|M^\star| = \nu(G)$ and $|X^\star| = \tau(G)$. For (2.7) we have $\nu(G) + |\Delta^\star| = n$, or $|\Delta^\star| = n - \nu(G)$. On the other hand, from (2.8) we have $|U^\star| = n - \tau(G)$. Since, König's Theorem states that $\nu(G) = \tau(G)$, we have the required result.

□

This concludes our presentation of the algorithmic aspect of Dilworth's Theorem on posets. We will be using the algorithm proposed in the proof of Lemma 2.7 heavily in the next chapters as a subroutine for finding a poset's minimum chain decomposition.

CHAPTER 3

WIDTH-BASED MODEL: SORTING ALGORITHMS

In this chapter we introduce the *Width-Based Model*. This model was introduced in a Daskalakis et al. paper [1] in 2011. In this setting we are given an oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \leq, \nsim\}$, of an underlying poset $\langle \mathcal{U}, \preceq \rangle$, and some constant $w \geq \text{width}(\mathcal{U}, \preceq)$ an upper bound to the width of our poset. Our goal is to reconstruct the unknown partial order using the *minimum* number of oracle calls. In the next chapter, we stay in the same setting and consider the k-Selection problem.

Sorting on Poset (Width-Based Model)	
Input:	1. A partially ordered set $\mathcal{U}$. 2. An oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \leq, \nsim\}$, of a partial order relation $\preceq$ on $\mathcal{U}$. 3. A constant $w \in \mathbb{N}$, with width$(\mathcal{U}, \preceq) \preceq w$.
Output:	The partial order relation $\preceq$.

We can derive a lower bound on the query complexity of the above problem using the following result due to Brightwell and Goodfall [18].

Theorem 3.1 (Brightwell and Goodfall [18], 1996)**.** Let $N_w(n)$ be the number of partially ordered sets on n elements and width at most w. We have,

$$\frac{n!}{w!}4^{n(w-1)}n^{-24w(w-1)} \leq N_w(n) \leq n!4^{n(w-1)}n^{-(w-2)(w-1)/2}w^{w(w-1)/2} \tag{3.1}$$

Using the above Theorem 3.1 and Theorem 4.5, which we present in the next chapter, we establish a lower bound on the number of queries required to sort a poset. Note that, as we shall see, Theorem 4.5 states that $\frac{w+1}{2}n - w$ queries are required to find the minimal elements of a poset.

Theorem 3.2. Any algorithm which sorts a poset a poset of width at most w on n elements requires $\Omega(n(\log n + w))$ queries.

Proof. From Theorem 3.1, if $w = o\left(\frac{n}{\log n}\right)$, then $N_w(n) = \Theta(n \log n + wn)$; hence $\Omega(n(\log n + w))$ queries are required *information theoretically* for sorting (see Section 1.4). On the other hand, let $w = \Omega\left(\frac{n}{\log n}\right)$. Theorem 4.5 gives a lower bound of

$\Omega(wn)$ queries for finding minimal elements of a poset. Since sorting is at least as hard as finding the minimal elements, it follows that $\Omega(wn)$ queries are necessary for sorting. In this case, $wn = \Omega(n \log n + wn)$.

□

We organize this chapter as follows. In Section 3.1 we give a data structure representing a poset. This data structure will be the output of our algorithms. In Section 3.2 we present a first algorithm for sorting called *Bin-Insertion Sort* with $O(wn \log n)$ oracle queries, due to Faigle and Túran [19]. In Section 3.3 we modify Bin-Insertion Sort to obtain the Entropy Sort algorithm of asymptotically optimal query complexity $O(n(\log n + w))$. This optimal algorithm is due to Daskalakis et al. [1]. Subsequently, in Section 3.4, we present a time-efficient variant of Merge Sort in poset, with $O(wn \log(n/w))$ query complexity and $O(w^2 n \log(n/w))$ running time. Lastly, in Section 3.5 we provide a summary of this chapter.

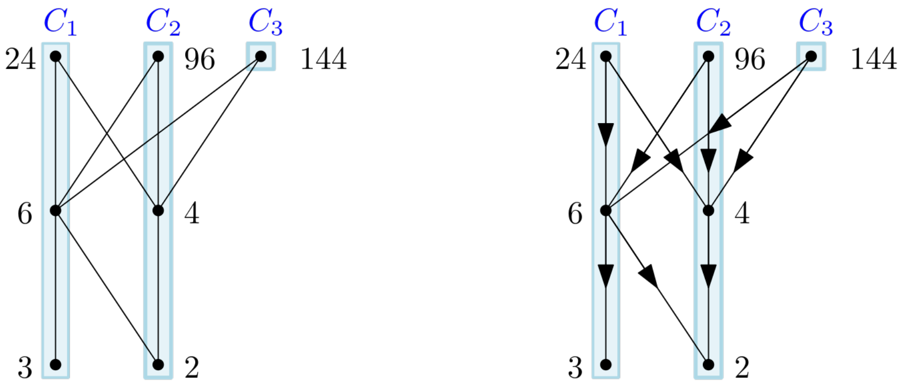
3.1 Representing a poset: The ChainMerge data structure

In this section we will establish the ChainMerge data structure, for encoding a poset. Our algorithms will output this data structure as a result. As we shall see we will be able to obtain all the information of the poset by querying this data structure. Namely, for two elements $a, b \in \mathcal{U}$, we can answer a query $c(a, b)$ in *constant time*. Essentially, ChainMerge will be a directed variation of a Hasse diagram. We built our data structure from a chain decomposition of a poset.

Let $\mathcal{C} = \{C_1, C_2, \ldots, C_w\}$ be a chain decomposition of a poset $\langle \mathcal{U}, \preceq \rangle$. We assume that the decomposition $\mathcal{C}$ is given as a set of *totally ordered* arrays. Let $x_i \in \mathcal{U}$ be an element of the i-th chain C_i. We store in our data structure a record of the form $\langle i \mid x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_w \rangle$. Namely, an *index* to the i-th chain, and $w - 1$ indices to the *largest elements, smaller than* x_i, from each chain. In Figure 3.1, we present the Hasse diagram of the Example 1.18 and the corresponding ChainMerge data structure. The performance of the data structure is characterized by the following lemma.

Lemma 3.3. Given a query oracle for a poset $\langle \mathcal{U}, \preceq \rangle$ *and* a chain decomposition $\mathcal{C}$ into w chains, building the ChainMerge data structure has *query complexity* and *time complexity* $O(wn)$, where $|\mathcal{U}| = n$. Given ChainMerge data structure, the relation of any pair of the elements can be found in *constant time*.

Proof. We will construct the records $\langle i \mid y_1, y_2, \ldots, y_{i-1}, y_{i+1}, \ldots, y_w \rangle$, of all the elements of the i-th chain, by simultaneously scanning the chains C_i and the rest of the chain $C_1, \ldots C_{i-1}, C_{i+1}, \ldots, C_w$. The scan begins with the smallest element of the chain C_i, and the smallest elements of the other chains. Let x_i be the current element of C_i and $y_1, \ldots, y_{i-1}, y_{i+1}, y_w$ the current elements of each of the other chains. At each step, we query whether x_i dominates any of the elements y_j. If so, we store an index to y_j and consider the next element in C_j. Else, we keep the j-th position in the above record empty, i.e. $y_j \leftarrow$ `nul` and stop considering this chain. This way, we make w iterations, at each iteration, we consider (in the worst case) all the elements of the other chains. Hence, for the query and time complexity we have $O\left(w \sum_{j \neq i} |C_j|\right) = O(wn)$.



(a) The Hasse diagram of Example 1.18.

(b) The corresponding ChainMerge data structure.

Figure 3.1: The Hasse diagram of Example 1.18 and the corresponding ChainMerge data structure, represented as a directed graph. The arcs represent an index.

Let $x, y \in \mathcal{U}$, with $x \in C_i$ and $y \in C_j$. The look-up operation works as follows. If $i = j$ we simply do a comparison of the indices of x and y in C_i, as in case of a total order. If $i \neq j$, then we look up the index of the largest element of C_j that is dominated by x; this is greater than (or equal to) the index of y in C_j if and only if $x \succ y$. If $x \not\succ y$, then we look up the index of the largest element of C_i that is dominated by y; this index is greater than (or equal to) the index of x in C_i if and only if $y \succ x$. If neither $x \succ y$ nor $x \prec y$, then $x \not\sim y$.

□

From Lemma 3.3 and the general discussions, our algorithms will be composed of two main steps. Consider a poset $\langle \mathcal{U}, \preceq \rangle$. Firstly, we find a chain decomposition of size w, where $\text{width}(\mathcal{U}, \preceq) \leq w$ the parameter of our model. Then, we construct the ChainMerge data structure as described in Lemma 3.3.

3.2 Bin-Insertion Sort

Before we present the optimal algorithm, with respect to the number of queries, due to Daskalakis et al., in this section we examine a more intuitive approach. A natural idea would be to sequentially insert elements into a subset poset, while maintaining a chain decomposition of the latter into a number of chains that is at most the upper bound w on the width of the poset to be constructed. A straightforward implementation of this idea is to perform a *binary search* on every chain of the decomposition in order to figure out the relationship of the elements being inserted with every element of that chain and, ultimately, with all the elements of the current poset. We call this algorithm Bin-Insertion Sort and was firstly presented by Faigle and Túran in 1985 [19]. We present Bin-Insertion Sort in Algorithm 5.

Observe that in Step 17 of Algorithm 5 we do not need additional queries to the oracle, since we have constructed the whole poset in P'. Also note that in Steps 7, 16 we can find a chain decomposition using flows as we described in Chapter 2. From Theorem 2.4 we can find a minimum chain decomposition in $O(mn)$ time, where $n = |\mathcal{U}|$ and $m = |\mathcal{R}_{\preceq}|$. For the *time complexity* of Algorithm 5, note that we have n

Algorithm 5: Bin-Insertion Sort

```
Input: 1. A set $\mathcal{U}$.
2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
Output: A ChainMerge data structure for $\langle \mathcal{U}, \preceq \rangle$.
Let $e$ be an arbitrary element of $\mathcal{U}$.
$P' \leftarrow \langle \{e\}, \varnothing \rangle$ // The current poset.
$\mathcal{U}' \leftarrow \{e\}, \mathcal{R}'_{\preceq} \leftarrow \varnothing$
$\mathcal{U} \leftarrow \mathcal{U} \setminus e$ // The set of unsorted elements.
while $\mathcal{U} \neq \varnothing$ do
    Choose an element $e \in \mathcal{U}, \mathcal{U} \leftarrow \mathcal{U} \setminus e$.
    Find a decomposition $\mathcal{C} = \{C_1, C_2, \dots, C_q\}$ of $P'$, of width $q \leq w$.
    // Do Binary Serach in each chain.
    for $i \in \{1, 2, \dots, q\}$ do
        Let $C_i = \{e_{i1}, e_{i2}, \dots, e_{i\ell_i}\}$ the $i$-th chain.
        Do binary search, in order to find the *smaller element (if any) that dominates $e$*.
        Do binary search, in order to find the *greater element (if any) that is dominated by $e$*.
    From the above binary searches we deduce the relationships between $e$ and the elements of $\mathcal{C}$.
    Add to $\mathcal{R}'$ all the relationships of $e$ with elements of $\mathcal{U}'$.
    $\mathcal{U}' \leftarrow P' \cup e$
    $P' \leftarrow \langle \mathcal{U}', \mathcal{R}_{\preceq} \rangle$
Find a decomposition $\mathcal{C}$ of $P'$
Construct the ChainMerge data structure from $P'$ and $\mathcal{C}$. // No additional queries.
return ChainMerge$(P', \mathcal{C})$
```

iterations. In each iteration we find a minimum chain decomposition in $O(nm)$ time and perform $O(w)$ binary searches of time $O(\log n)$. Therefore, we have $O(n(nm + w \log n)) = O(n^2m + w \log n)$ time. On the other hand, we can come up with an upper bound to the number of relationships m, if we observe that *at least two* elements are incomparable between two distinct chains in a *minimum* chain decomposition. Hence we have,

$$m = |\mathcal{R}_{\preceq}| = O\left(\binom{n}{2} - \binom{w}{2}\right) = O(n^2 - w^2).$$

Thus, for the time complexity we get,

$$O(n^2m + w \log n) = O(n^4 - n^2w^2 + w \log n) \tag{3.2}$$

Since in $O(n^2)$ time we can reconstruct naively the whole Hasse diagram, a (rightfully) curious reader would wonder why we got ourselves in so much trouble for an inefficient algorithm. This is because the naive algorithm will also make $O(n^2)$ queries. We give the next Lemma for the query complexity of Algorithm 5.

Lemma 3.4 (Faigle and Túran [19], 1985)**.** Algorithm 5 sorts any partial order $\langle \mathcal{U}, \preceq \rangle$ of width at most w on n elements with $O(wn \log n)$ oracle queries.

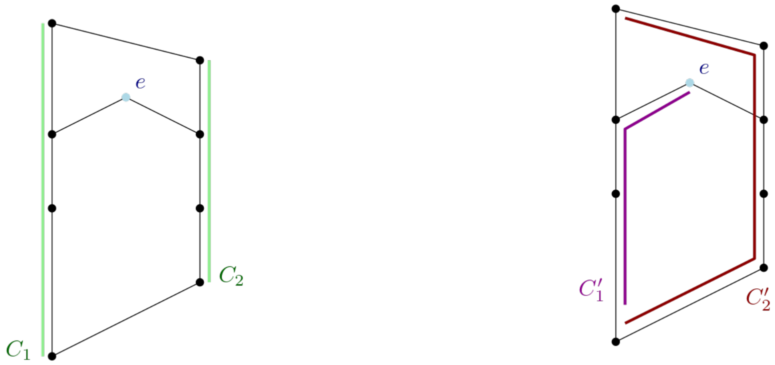
Proof. The correctness of Algorithm 5 should be clear from its description. Also, it is not hard to see that the number of oracle queries includes by the algorithm for inserting each element is $O(w \log n)$. Note that in order to find a minimum chain decomposition we don't make additional queries. Therefore, the total number of queries is $O(nw \log n)$.

□

3.2.1 Greedy Counter-Example

Before we close this section argue about the necessity of the flow algorithm of Chapter 2 in each iteration of Algorithm 5. Why should we re-compute a chain decomposition in each iteration? Couldn't we update the existing decomposition? As it turns out, a naive incremental algorithm for updating an existing chain decomposition wouldn't work. Let $\mathcal{C}$ be an existing decomposition, while e is a new element after we induced all its relationships from Steps 9 to 12. We need to find some chain $C \in \mathcal{C}$, whose all elements are related to e. If there is such chain, we just add e to C. Otherwise, we create a new chain $\{e\}$. In order to prove the correctness of this greedy algorithm, we have to argue that if q the size of the greedy chain decomposition, then $q = \text{width}(\mathcal{U}, \preceq)$. Observe Figure 3.2. Let $\mathcal{C} = \{C_1, C_2\}$ be our existing chain decomposition, as shown in Figure 3.2a. The greedy algorithm would add e in a singleton chain, resulting in a decomposition of size 3. On the other hand, in Figure 3.2b we show the optimal chain decomposition of size 2.

Thus we concluded our discussion about the first sorting algorithm in posets. Also, we noted the trade-off between time and query complexity. From Theorem 3.2 we know that Algorithm 5 is not optimal. In the next section we modify Algorithm 5 in order to show an optimal algorithm, with respect to the number of queries. There the time-query trade-off will become more apparent. Lastly, note our last remark regarding a greedy incremental chain decomposition algorithm. In Section 3.4 we devise an incremental, non-trivial, algorithm for finding a minimum chain decomposition.



(a) An existing chain decomposition.

(b) The minimal chain decomposition of size 2, after we add e optimally.

Figure 3.2: A counter-example to a greedy incremental chain decomposition algorithm. Note that in the left figure a greedy algorithm would create 3 chains. On the right, we show a minimal chain decomposition of size 2.

3.3 Entropy Sort

In this Section we describe the optimal algorithm EntropySort [1], with respect to the number of queries. We begin by discussing the big picture of such an optimal query algorithm. Let's consider the sequence of queries of the optimal algorithm. For any set of oracle queries and responses, there is a corresponding set of posets, which we call *candidates*. These are the posets consistent with the responses to there queries. A natural sorting algorithms would find a sequence of oracle queries such that, for each query the possible responses to the query partition the spaces of candidates into three parts, at least two of which are relatively large. Such an algorithm would achieve the information-theoretic lower bound.

For example, the effectiveness of Quicksort for sorting total orders relies on the fact that most of the queries made[1] by the algorithm partition the space of candidate total orders into two parts, each of relative size at least $1/4$. Indeed, in case of total orders, much more is known; for any subset of possible queries to the oracle, there always exists a query that partitions the spaces of candidate total orders into two parts, each of relative size at least $3/11$ [20, 21].

The Achilles' kneel of Bin-Insertion Sort is in the method of insertion of an element, specifically in the way the binary searches are performed in Steps 9-12. In these sequences of queries, no structural properties of P' are used for deciding which oracle queries are more useful than the others. In some sense, the binary searches give the same "attention" to queries whose answer is guaranteed to greatly decrease the number of remaining possibilities and those whose answer could potentially not be very informative.

The Daskalakis' et al. algorithm tries to resolve this conundrum. The suggested scheme has shares the same structure with Bin-Insertion Sort but exploits the structure of the already constructed P' in order to *amortize* the cost of queries over the insertions.

[1] Half of the queries in average; if we choose the pivot at random. The probability of choosing good pivots can be increased by using a technique similar to the Median of Medians algorithm of Chapter 1. Namely, we partition the input in k parts, e.g. $k = 3$; choose randomly an element of each part; then selecting the median of these elements as pivot.

The amortized query cost matches the lower bound of Theorem 3.2.

3.3.1 Weighted Binary Search

The essence of the optimal EntropySort lies in modifying the binary searches into *weighted binary searches*. The weights assigned to the elements satisfy the following property. *The number of queries it takes to insert an element into a chain is proportional to the (logarithm of the) number of candidate posets that will be eliminated after the insertion of the element.* In other words, we use fewer queries for insertions that are less informative and more queries for insertions that are informative.

To define this notion formally, we introduce some notation. Let $P' = \langle \mathcal{U}', \mathcal{R}'_{\preceq} \rangle$ be the already computed poset, with width w. Also, let $u \in \mathcal{U} \setminus \mathcal{U}'$ a new element we would like to include in P'. We define some set of relations $\mathcal{ER}, \mathcal{PR} \subseteq (\{u\} \times \mathcal{U}') \cup (\mathcal{U}' \times \{u\})$. We call $\mathcal{ER}$ the *enforced relations*, and $\mathcal{PR}$ the *prohibited relations*. We call some poset $P'' = \langle \mathcal{U}' \cup u, \mathcal{R}''_{\preceq} \rangle$ with $(\mathcal{R}'_{\preceq} \cup \mathcal{ER}) \subseteq \mathcal{R}''_{\preceq}$ and $(\mathcal{PR} \cap \mathcal{R}''_{\preceq}) = \varnothing$, *a w-extension of P' conditioned on* $(\mathcal{ER}, \mathcal{PR})$. In other words, we demand P'' to be an extension of P' with the same width, that includes the relations of $\mathcal{ER}$, but prohibits the relations of $\mathcal{PR}$.

For the weighted binary search, we proceed as follows. Consider the interval $[0, 1)$. We will partition $[0, 1)$ into disjoint sub-intervals, each of which corresponds to an element e_{ik}, the k-th element of the i-th chain C_i in an existing chain decomposition $\mathcal{C} = \{C_1, C_2, \ldots, C_w\}$, of the already computed poset P'. Consider the i-th chain $C_i = \{e_{i1}, e_{i2}, \ldots, e_{i\ell_i}\}$, sorted in *increasing* order, i.e. $e_{i1} \preceq e_{i2} \preceq \ldots \preceq e_{i\ell_i}$. With ℓ_i we denote the number of element in the chain C_i. Also, consider the k-th element e_{ik}. Let e be the new element we would like to sort[2]. Since, the underlying poset is unknown is some universe e_{ik} is the *smallest element of C_i that dominates e*. To enforce this scenario we define,

$$\mathcal{ER}_k = \{e \preceq e_{ij} \mid j \in \{k, k+1, \ldots, \ell_i\}\},$$
$$\mathcal{PR}_k = \{e \preceq e_{ij} \mid j \in \{1, 2, \ldots, k-1\}\}.$$

With $\mathcal{ER}_k$ we enforce that all the elements greater (or equal to) e_{ik} *also* dominate e. On the other hand, with the constraints of $\mathcal{PR}_k$ we establish that the elements smaller than e_{ik} *do not* dominate e. With $\mathcal{D}_{ik}$ we denote *the number of w-width extensions conditioned on* $(\mathcal{ER}_k, \mathcal{PR}_k)$. Let $\mathcal{D}_i = \sum_{k=1}^{\ell+1} \mathcal{D}_{ik}$, where in the $(\ell+1)$-th scenario *non element of C_i dominates e*. Now we can define a *mass function* for each $e_{ik} \in C_i$,

$$\texttt{mass}_{i,e}(e_{ik}) = \frac{\mathcal{D}_{ik}}{\mathcal{D}_i}. \tag{3.3}$$

Observe that $\sum_{k=1}^{\ell} \texttt{mass}_{i,e}(e_{ik}) = 1$. Also, our mass function depends only from the chain C_i *and* the element to be sorted e. Using the mass function we can define a bijection between the elements of C_i and interval $[b_k, t_k) \subseteq [0, 1)$ with length corresponding to the mass of e_{ik}. We have $b_1 = 0, b_k = t_{k-1}$ and $t_k = \sum_{k' \leq k} \texttt{mass}_{i,e}(e_{ik})$. Hence, the correspondence $\texttt{interval}_{i,e} \colon C_i \to 2^{[0,1)}$, with $\texttt{interval}_{i,e}(e_{ik}) \mapsto [b_k, t_k)$ is a well defined isomorphism. Note that the bijection $\texttt{interval}_{i,e}(\cdot)$ depends on the chain C_i as well as on the element e to be sorted. With the function $\texttt{interval}_{i,e}(\cdot)$ computed we have all the necessary ingredients to do *weighted binary search* on the elements of C_i to find the *smallest element $e_{ik} \in C_i$ that dominates e*.

[2] Insert e to P', while keeping $P' \cup e$ sorted.

Algorithm 6: Weighted Binary Search

```
   Input: 1. A totally ordered set $C_i$.
   2. An interval function $\texttt{interval}_{i,e} \colon C_i \to 2^{[0,1)}$.
   3. The element to be sorted $e$.
   Output: The smallest element $e_k \in C_i$ that dominates $e$.
1  $x \leftarrow 1/2$
2  $e_{ik} \leftarrow \texttt{interval}_{i,e}^{-1}(x)$
3  while $\neg(e_{ik-1} \preceq e \prec e_{ik})$ do
4      if $e \preceq e_{ik}$ then
5          // look below
6          $x \leftarrow x/2$
7      else
8          // look above
9          $x \leftarrow x + x/2$
10     $e_{ik} \leftarrow \texttt{interval}_{i,e}^{-1}(x)$
11 return $e_{ik}$
```

In Algorithm 6 we present the pseudocode for the Weighted Binary Search. Note that in Steps 2, 10 we mildly violate the notation and denote with $\texttt{interval}_{i,e}^{-1}(\cdot)$ the element e_{ik} that corresponds to an interval $[b_k, t_k)$ such that $x \in [b_k, t_k)$. Also, to ease the notation we emit the "edges" cases where no element of C_i dominates e, or e_{i1} dominates e, but can be easily induced from context. In the following lemma we argue about the *query complexity* of Algorithm 6.

Lemma 3.5. For every $k \in \{1, 2, \ldots, \ell_i + 1\}$ if e_{ik} is the smallest element of chain C_i which dominates element e (with $k = \ell_i + 1$ corresponding to the case where no element of C_i dominates e), then k is found after *at most*

$$2\left(1 + \log \frac{1}{\texttt{mass}_{i,e}(e_{ik})}\right)$$

oracle queries in Algorithm 6.

Proof. Let $\lambda = \texttt{mass}_{i,e}(e_{ik})$ be the length of the interval that corresponds to e_{ik}. We wish to prove that the number of queries needed to find e_{ik} (and determine that it is the smallest element of C_i that dominates e) is at most $2\left(1 + \lfloor \log \frac{1}{\lambda} \rfloor\right)$. From the definition of the Weighted Binary Search, we see that if the interval corresponding to e_{ik} contains a point x of the form $2^{-r} \cdot m$ inits interior, where r, m are integers, then the search completes after *at most r iterations*. Each iteration involves at most two oracle queries. Now, an interval of length λ must include a point of the form $2^{-r} \cdot m$, where $r = 1 + \lfloor \log \frac{1}{\lambda} \rfloor$, which concludes the proof.

□

Note that uniformative insertions would correspond to large mass, namely if some element e_{ik} has large mass, then we would have that e_{ik} is the smallest element of C_i that dominates e in "many universes". Since its quite likely that e_{ik} is "the smallest dominator" of e verifying this fact doesn't tells us much about the structure of the unknown poset. Hence this explains the use of the term *entropy-weighted search*. A symmetric version of Lemma 3.5 holds, naturally, for finding the *largest element of chain C_i dominated by e.*

3.3.2 The Algorithm

With the above discussion regarding the Weighted Binary Search, most of our algorithm is completed. We only need to put the pieces of the puzzle together. Remember we are going to "patch" the Lines 9-12 of Algorithm 5. Let us summarize the work of the previous subsection. We have the already computed poset $P' = \langle \mathcal{U}', \mathcal{R}'_{\preceq} \rangle$. We also have a chain decomposition $\mathcal{C} = \{C_1, C_2, \dots, C_w\}$ of P'. With e we denote the new element we would like to sort. For each chain $C_i = \{e_{i1}, e_{i2}, \dots, e_{i\ell_i}\}$ we proceed as follows. For each $e_{ik},\ k \in [\ell_i]$ we compute a set of constraints $(\mathcal{ER}_k, \mathcal{PR}_k)$. From the set of constraints we compute the mass function $\texttt{mass}_{i,e} \colon C_i \to [0, 1)$. Subsequently, from the mass function we compute the interval correspondence $\texttt{interval}_{i,e} \colon C_i \to 2^{[0,1)}$. Lastly, using the interval correspondence we compute the "smallest dominator" e_{ik} of e, using Algorithm 6. We use a similar process to find the greatest element e_{iq} that e dominates.

We describe this process in a formal way in the Algorithm 7. In the following algorithm, we introduce a different version of the mass function, the interval correspondence and the Weighted binary sort, for the two problems we examine. With $\texttt{mass}^{+}_{i,e}(\cdot)$, $\texttt{interval}^{+}_{\texttt{i,e}}(\cdot)$ and $\texttt{WeightedBinarySort}^{+}(\cdot,\cdot,\cdot)$ we denote the notions for the smallest dominator sub-problem. With $\texttt{mass}^{-}_{i,e}(\cdot)$, $\texttt{interval}^{-}_{\texttt{i,e}}(\cdot)$ and $\texttt{WeightedBinarySort}^{-}(\cdot,\cdot,\cdot)$ we denote the symmetrical notions for finding the greater element of C_i that is dominated by e. Lastly, note that a pair of constraints $(\mathcal{ER}, \mathcal{PR})$. In each iterations of the for-loop we add to $(\mathcal{ER}, \mathcal{PR})$ the constraints $(\mathcal{ER}^{+}_{k^\star}, \mathcal{PR}^{+}_{k^\star})$ and $(\mathcal{ER}^{-}_{p^\star}, \mathcal{PR}^{-}_{p^\star})$ corresponding to the results of $\texttt{WeightedBinarySearch}^{+}$ and $\texttt{WeightedBinarySearch}^{-}$, respectively. Lastly, we return the set $\mathcal{ER}$ as the deduced set of relations.

3.3.3 Analysis

Before we proceed with the query analysis of Algorithm 7, we need to address the elephant in the room; the time complexity of the algorithm. Algorithm 7 is *exponential.* This should be clear since the computation of the mass function involves the computational expensive task of enumerating all possible extensions of P' and filter out these that not respect the constraints $(\mathcal{ER} \cup \mathcal{ER}^{+}_{k}, \mathcal{PR} \cup \mathcal{PR}^{+}_{k})$. At the moment this thesis is being written finding an efficient way to compute the mass function $\texttt{mass}_{i,e}(e_{ik})$ remains an open problem. The authors of [1] suggest the work due to Dyer, Freize, and Kannan for *"approximating the volume of convex bodies"* [22], as a possible route for improving the time complexity of EntropySort.

The following Lemma characterizes the query complexity of a single iteration of EntropySort[3] and is the core result that will help use establish the query complexity of the whole algorithm. Let us introduce some notation. Suppose that $\mathcal{U} = \{e_1, e_2, \dots, e_n\}$ is the universe of our elements, sorted in the order in which the elements of $\mathcal{U}$ are inserted iinto the poset P'. Also, denote with P_d the *restriction* of poset $P = \langle \mathcal{U}, \preceq \rangle$ onto the set of elements $\{e_1, e_2, \dots, e_d\}$. Additionally, with Z_d we denote *the number of w-width extensions of P_d onto $\mathcal{U} \setminus \{e_1, e_2, \dots, e_d\}$.* Clearly , $Z_0 \equiv N_w(n)$ and $Z_n = 1$.

[3] From now on we will use the term *EntropySort* to denote the "patched" version of Algorithm 5, using Algorithm 7.

Algorithm 7: EntropySort

```
Input: 1. An already computed poset $P' = \langle \mathcal{U}', \mathcal{R}'_{\preceq} \rangle$.
2. A chain decomposition of $P'$, $\mathcal{C} = \{C_1, C_2, \dots, C_w\}$.
3. A new element $e \notin \mathcal{U}'$ to be sorted.
Output: The set of relationships between $e$ and the elements of $\mathcal{C}$.
$\mathcal{ER} \leftarrow \varnothing, \mathcal{PR} \leftarrow \varnothing$ // A partial set of relations.
for $i \in [w]$ do
    // For each chain..
    Let $C_i = \{e_{i1}, e_{i2}, \dots, e_{i\ell_i}\}$, with $e_{i1} \preceq e_{i2} \preceq \dots \preceq e_{i\ell_i}$.
    // Finding the smaller element that dominates $e$
    For each $e_{ik}$, compute a pair of restrictions $(\mathcal{ER}^+_k, \mathcal{PR}^+_k)$.
    From the pairs $(\mathcal{ER} \cup \mathcal{ER}^+_k, \mathcal{PR} \cup \mathcal{PR}^+_k), k \in [w]$ of restrictions compute
      the mass function $\texttt{mass}^+_{i,e} \colon C_i \to [0,1)$.
    From the mass function compute the interval correspondence
      $\texttt{interval}^+_{i,e} \colon C_i \to 2^{[0,1)}$.
    $e_{ik^\star} \leftarrow \texttt{WeightedBinarySearch}^+(C_i, \texttt{interval}^+, e)$
    // Where $e_{ik^\star}$ the smaller element that dominates $e$
    $\mathcal{ER} \leftarrow \mathcal{ER} \cup \mathcal{ER}^+_{k^\star}, \mathcal{PR} \leftarrow \mathcal{PR} \cup \mathcal{PR}^+_{k^\star}$
    // Finding the greater element that is dominated by $e$
    For each $e_{ip}$, compute a pair of restrictions $(\mathcal{ER}^-_p, \mathcal{PR}^-_p)$.
    From the pairs $(\mathcal{ER} \cup \mathcal{ER}^-_p, \mathcal{PR} \cup \mathcal{PR}^-_p), p \in [w]$ of restrictions compute
      the mass function $\texttt{mass}^-_{i,e} \colon C_i \to [0,1)$.
    From the mass function compute the interval correspondence
      $\texttt{interval}^-_{i,e} \colon C_i \to 2^{[0,1)}$.
    $e_{ip^\star} \leftarrow \texttt{WeightedBinarySearch}^+(C_i, \texttt{interval}^-, e)$
    // Where $e_{ip^\star}$ the greater element that is dominated by $e$
    $\mathcal{ER} \leftarrow \mathcal{ER} \cup \mathcal{ER}^-_{p^\star}, \mathcal{PR} \leftarrow \mathcal{PR} \cup \mathcal{PR}^-_{p^\star}$
return $\mathcal{ER}$
```

Lemma 3.6. Algorithm 7 needs at most $4w+2\log\frac{Z_k}{Z_{k+1}}$ oracle queries to insert element e_{k+1} into poset P_k in order to obtain P_{k+1}.

Proof. Let $\mathcal{C} = \{C_1, \dots, C_w\}$ be a chain decomposition of P_k. Suppose also that, for all $i \in [w]$, $k_i^\star \in \{1, \dots, \ell_i + 1\}$ and $p_i^\star \in \{0, 1, \dots, \ell_i\}$ are the indices computed by the weighted binary searches. Also , let $\mathcal{D}_i^+, \mathcal{D}_{ij}^+, j \in \{1, \dots, \ell_i + 1\}$ and $\mathcal{D}_i^-, \mathcal{D}_{ij}^-, \in \{0, 1, \dots, \ell_i\}$ the respective quantities, as defined in Subsection 3.3.1. We also stick to the $+/-$ exponents of Subsection 3.3.2 for denoting the two phases of the for-loop of Algorithm 7. The following hold,

$$Z_d = \mathcal{D}_1^+, \tag{3.4}$$

$$\mathcal{D}_{p_w^\star}^- = Z_{d+1}, \tag{3.5}$$

$$\mathcal{D}_{ik_i^\star}^+ = \mathcal{D}_i^- \quad \forall i \in [w], \tag{3.6}$$

$$\mathcal{D}_{ip_i^\star}^- = \mathcal{D}_{i+1}^- \quad \forall i \in [w-1]. \tag{3.7}$$

The validity of the above equations should come naturally, since the numbers Z_d and $\mathcal{D}_i^+$, $\mathcal{D}_{i,j}^+$ measure the number of possible universes. Just, note that the milestones in the variation of Z_d is when we find the smaller element $e_{ik_i^\star}$ that dominates e, and the greater element $e_{ip_i^\star}$ that is dominated by e.

Using Lemma 3.5 it follows that the total number of queries required to construct P_{d+1} from P_d is at most,

$$\sum_{i=1}^{w}\left(2 + 2\log\frac{\mathcal{D}_i^+}{\mathcal{D}_{ik_i^\star}^+} + 2 + 2\log\frac{\mathcal{D}_i^-}{\mathcal{D}_{ip_i^\star}^-}\right) \leq 4w + 2\log\frac{Z_k}{Z_{k+1}}$$

which is what we wanted to show.

□

In the following Theorem we prove the total query complexity of the EntropySort.

Theorem 3.7 (Daskalakis et al. [1], 2011)**.** EntropySort sorts any partial order $P = \langle \mathcal{U}, \preceq \rangle$ of width at most w on n elements using at most $2\log N_w(n) + 4wn = \Theta(n\log n + wn)$ oracle queries. In particular the query complexity of the algorithm is at most $2n\log n + 7wn + 2w\log w$.

Proof. From Lemma 3.6, the query complexity of the EntropySort is,

$$\begin{aligned}
&\sum_{d=0}^{n-1}(\text{\# queries to insert element } e_{d+1})\\
&\quad = \sum_{d=0}^{n-1}\left(4w + 2\log\frac{Z_k}{Z_{k+1}}\right)\\
&\quad = 4w + 2\log\frac{Z_0}{Z_n}\\
&\quad = 4wn + 2\log N_w(n)
\end{aligned}$$

Taking the logarithm of the upper bound from Theorem 3.1, it follows that the number of queries required by the algorithm is $2n\log n + 8wn + 2w\log w$. Thus, we proved what was desired.

□

3.4 Merge Sort

In this section we turn our attention to the time efficiency for the sorting problem. We present the algorithm Merge-Sort which superficially imitates the recursive structure of the classical merge sort on totally ordered sets. The merge step is quite different however; it makes use of the technical Peeling Algorithm in order to efficiently *maintain a small chain decomposition of the poset* throughout the recursion[4]. The Peeling subroutine is a specialization of the classical flow-based bipartite-matching algorithm, introduced in Chapter 2, that is efficient in the comparisons model. We begin by giving a description of the Merge Sort method in the first Subsection 3.4.1. In Subsection 3.4.2 we analyse the Peeling subroutine.

3.4.1 Merge Sort

We begin from the top down by giving the Merge-Sort method. Let $\mathcal{U}$ be our universe, and $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ be an oracle to the poset $\langle \mathcal{U}, \preceq \rangle$. Also, let w be an upper bound to the width of our poset, namely width$(\mathcal{U}, \preceq)$. The Merge Sort algorithm produces a decomposition of $\langle \mathcal{U}, \preceq \rangle$ into w chains and concludes by building a ChainMerge data structure. In order to obtain the chain decomposition the algorithm partitions the elements of $\mathcal{U}$ into two subsets of (as close as possible to) equal size. Then finds a chain decomposition of each subset *recursively*. The recursive call returns a decomposition of each subset into at most w chains, which constitutes a decomposition of the whole set $\mathcal{U}$ into at most $2w$ chains. Then the Peeling algorithm of the next subsection is applied to *reduce the decomposition to a decomposition of w chains*. Given a decomposition of $\mathcal{U}' \subseteq \mathcal{U}$, where $m = |\mathcal{U}'|$, into at most $2w$ chains, the Peeling subroutine returns a decomposition using $4wn$ queries and $O(w^2 m)$ time. The pseudocode of the Merge-Sort is given in Algorithms 8 and 9, while the performance is characterized by Theorem 3.8.

Algorithm 8: Merge Sort

```
Input: 1. A set $\mathcal{U}$.
2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
Output: A ChainMerge data structure for $\langle \mathcal{U}, \preceq \rangle$.
$\Delta \leftarrow$ MergeSort-recursive$(\mathcal{U}, c, w)$ // A decomposition of $\langle \mathcal{U}, \preceq \rangle$
    into $w$ chains.
return ChainMerge$(\Delta)$
```

Theorem 3.8 (Daskalakis et al. [1], 2011)**.** *The algorithm Merge Sort sorts an poset $\langle \mathcal{U}, \preceq \rangle$ of width at most w on n elements using at most $4wn \log\left(\frac{n}{w}\right)$ queries and total complexity $O\left(w^2 n \log\left(\frac{n}{w}\right)\right)$.*

Proof. The correctness of Algorithm 8 is founded upon the correctness of the peeling method, the latter is established in Theorem 3.9. Let $T(m)$ and $Q(m)$ be the worst-case time and query complexity respectively, for a poset on m elements and width at most

[4] Note our comments for that matter at Subsection 3.2.1. Maintaining a chain decomposition, while adding a new element is not a trivial problem.

Algorithm 9: Merge Sort

```
Input: 1. A set $\mathcal{U}'$.
2. An oracle $c\colon \mathcal{U}' \times \mathcal{U}' \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
Output: A decomposition of $\mathcal{U}'$ into $w$ chains.
if $|\mathcal{U}'| \leq w$ then
    return $\{\mathcal{U}'\}$ // the trivial decomposition of length 1.
Partition $\mathcal{U}'$ into $\mathcal{U}'_1$ and $\mathcal{U}'_2$.
$\Delta_1 \leftarrow$ MergeSort-recursive($\mathcal{U}'_1, c, w$)
$\Delta_2 \leftarrow$ MergeSort-recursive($\mathcal{U}'_2, c, w$)
$q \leftarrow |\Delta_1| + |\Delta_2|$
if $q > w$ then
    $\Delta \leftarrow$ Peeling($\mathcal{U}', c, w, \Delta_1 \cup \Delta_2$)
else
    $\Delta \leftarrow \Delta_1 \cup \Delta_2$
return $\Delta$
```

w. When $m \leq w$, $T(m) = O(w)$ and $Q(m) = 0$. Otherwise, we have,

$$T(m) = 2T\left(\frac{m}{2}\right) + O(w^2 n),$$
$$Q(m) = 2Q\left(\frac{m}{2}\right) + 2wm,$$

where the last terms of each equation correspond to the time and query complexity of Peeling. Solving the recursions we have $T(n) = O\left(w^2 n \log\left(\frac{n}{w}\right)\right)$ and $Q(n) \leq 4wn \log\left(\frac{n}{w}\right)$. Lastly, note that the cost of constructing the ChainMerge data structure is negligible.

□

3.4.2 Peeling Algorithm

We describe an algorithm that efficiently reduces the size of a given decomposition of a poset. It can been seen as an adaptation of the classic flow-based bipartite-matching algorithm, we presented in Chapter 2, adapted to be efficient in the *width-based oracle model*. The algorithm is optimized for reducing the size of a given decomposition rather than constructing a minimum chain decomposition from scratch[5].

The Peeling algorithm is given an oracle $c(\cdot, \cdot)$ for a poset $\langle \mathcal{U}, \preceq \rangle$, where $n = |\mathcal{U}|$, and a decomposition Δ into $q \leq 2w$ chains. At first, it builds a ChainMerge data structure using at most $2qn$ queries and $O(qn)$ time. Every subsequent query the algorithm makes after that is actually *a look-up* in the data structure and therefore takes *constant times* and *no oracle call*. Our algorithm proceeds in a number of *peeling iterations*. Each iteration produces a decomposition of $\langle \mathcal{U}, \preceq \rangle$ with *one less chain*. After *at most* w peeling iterations, we will have a decomposition of size at most w.

In order to comprehend the Peeling algorithm we need to understand the essence of the greedy counter-example of Subsection 3.2.1. There the optimal chain decomposi-

[5] For an efficient algorithm constructing a minimum chain decomposition see [23], due to Y. Chen (2007). This algorithm is unsuited for our model.

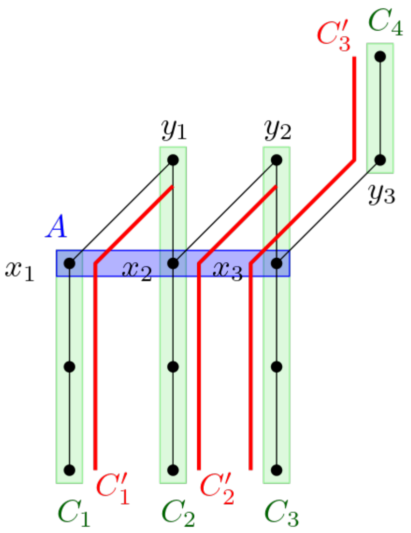


Figure 3.3: Peeling Example. We reduce the chain decomposition from $\{C_1, C_2, C_3, C_4\}$ to $\{C'_1, C'_2, C'_3\}$. With A we denote a maximum chain. The pairs $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ form a dislodgement sequence.

tion would break a chain into two, insert the new element, and then append the one part of the first chain to the second. A more sophisticated example is presented in Figure 3.3. Note that in Figure 3.3 we are initially given a chain decomposition of size four. Then, we partition each of the chains C_2, C_3, C_4 into two parts above y_1, y_2, y_3 and below respectively, including the $y_i, i \in [3]$. With $C[y:]$ we will denote the elements of a chain *above* (greater than or equal to) y, whereas with $C[:x]$ we denote the elements of the chain C *below* (less than or equal to) We concatenate the subchain $C_{i+1}[y_i:]$ with the subchain $C_i[:x_i]$. Thus, obtaining a new chain C'_i. We call the sequence $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ a *dislodgement* sequence, for intuitively x_i will *dislodge* y_i. Note that this way, we reduce the number of chains, *if and only if* the last element of the sequence y_ℓ is a *minimum* element in its chain. In Algorithm 10 we give the pseudocode of the Peeling algorithm. Theorem 3.9 characterizes the query and time complexity.

Theorem 3.9 (Daskalakis et al. [1], 2011)**.** Given an oracle for the poset $\langle \mathcal{U}, \preceq \rangle$, where $n = |\mathcal{U}|$, and a decomposition into at most $2w$ chains, the Peeling Algorithm 10 returns a decomposition of $\mathcal{U}$ into w chains. It has *query* complexity at most $O(wn)$ and *time* complexity of $O(w^2 n)$.

Proof. To prove the correctness of one peeling iteration, we observe first that it is always possible to find a pair (x, y) of top elements such that $y \succ x$, as specified in Step 8, since the size of any antichain is at most w. We now argue that it is possible to find a subsequence of dislodgements as specified by Step 10. Let y_t be the element whose deletion broke the while-loop. Since y_t was dislodged by x_t, x_t was the top element of some list *when that happened*. In order for x_t to be a top element, it was either top from the beginning, or its parent y_{i-1} must have been dislodged by some element x_{t-1}, *and so on*.

We claim that, given a decomposition into q chains, one peeling iteration produces a decomposition of $\mathcal{U}$ into $q-1$ chains. Recall that $y_1 \succ x_1$ and moreover, for every i, $2 \leq i \leq t$, $y_i \succ x_i$ and $y_{i-1} \succ x_i$. Observe that, after Step 11 of the peeling iteration,

Algorithm 10: Peeling

```
Input: 1. A set $\mathcal{U}'$.
2. An oracle $c\colon \mathcal{U}' \times \mathcal{U}' \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
4. A chain decomposition $\Delta$ of $\mathcal{U}'$.
Output: A decomposition of $\mathcal{U}'$ into $w$ chains.
$q \leftarrow |\Delta|$
for $i \in [q]$ do
    Build a linked linked list $C_i = e_{i\ell_i} \to e_{i\ell_i-1} \to \cdots \to e_{i1}$, where
        $e_{i\ell_i} \succ e_{i\ell_i-1} \succ \ldots \succ e_{i1}$. // The lists are sorted in
        decreasing order.
while $q > w$ do
    // Perform a peeling iteration.
    for $i \in [q]$ do
        $C'_i \leftarrow C_i$
    while $\nexists C'_i \neq \varnothing$ do
        // the largest elements of each $C'_i$ is a top element.
        Find a pair $(x, y)$ s.t.: (a) $x \in C'_i$ $y \in C'_j$; (b) top elements; and (c)
          $x \prec y$.
        $C'_j \leftarrow C'_j \setminus y$
    In the sequence of dislodgements, find a subsequence
      $(x_1, y_1), \ldots, (x_t, y_t)$, s.t.: (a) $y_t$ is the element whose deletion created an
      empty chain; (b) for $i \geq 2$ $y_{i-1}$ is parent of $x_i$ in its original chain; (c) $x_1$
      is the top element of one of the original chains.
    // Modify the original chains $C_1, \ldots, C_q$
    for $i = 2, \ldots, t$ do
        Delete the pointer going from $y_{i-1}$ to $x_1$
        Replace it with a pointer going from $y_i$ to $x_i$
    Add a pointer from $y_1$ to $x_1$
    $q \leftarrow q - 1$
    Adjust the indices of the original chains.
return the current chain decomposition, containing $w$ chains.
```

the total number of pointers has increased by 1. Therefore, if the link structure remains a union of disconnected chains, the number of chains must have decreased by 1, since one extra pointer implies 1 less chain. It can been seen that the switches performed by Step 11 of the algorithm maintain the invariant that the *in-degree and out-degree of every vertex is bounded by 1.* Moreover, no circles are introduced, since every pointer that is added corresponds to a valid relation. Therefore, the link structure is indeed a union of disconnected chains.

The query complexity of the Peeling algorithm is exactly the query complexity of ChainMerge, which is at most $O(wn)$. We show next that one peeling iteration can be implemented in time $O(qn)$, which implies the claim.

In order to implement one peeling iteration in $O(qn)$ time, a little book-keeping is needed, in particular, for Step 10. We maintain during the peeling iteration a list L of potentially comparable pairs of elements. At any time, if a pair (x, y) is in L, then x and y are top elements. At the beginning of the iteration L consists of all pairs (x, y), where x and y are top elements. Any time an element x that was not a top element becomes a top element, we add to L the set of pairs (x, y) that y is currently a top element. Whenever a top element x is dislodged, we remove from L all pairs contain x. When Step 8 requires us to find a pair of comparable top elements, we take an arbitrary pair (x, y) out of L and check whether x and y are comparable. If they are not comparable, we remove (x, y) from L and try the next pair. Thus, *we never compare a pair of top elements more than once*. Since each element of $\mathcal{U}$ is responsible for inserting *at most* q pairs into L (when it becomes top element), it follows that a peeling iteration can be implemented in $O(qn)$ time.

□

3.4.3 Lifting the Known Width Hypothesis

We end this section with a remark regarding the *"known width hypothesis"*. Recall from the beginning of the chapter that $N_w(n)$ is the number of posets of width at most w on n elements. In the following Proposition we lift our hypothesis that an upper bound on the width is known. We will modify the Entropy Sort, of Algorithm 7, and the Merge Sort, of Algorithm 8, to work without the w parameter. It turns out that this is quite simple; let $n = |\mathcal{U}|$, for $i \in [n]$ we try the width values of the form $w = 2^i$.

Proposition 3.10. Given a set $\mathcal{U}$ of n elements and access to an oracle $c \colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \leq, \nsucc\}$ of a poset $\langle \mathcal{U}, \preceq \rangle$ of unknown width, there is an algorithm that sorts $\mathcal{U}$ using at most $\log w(2 \log N_{2w}(n) + 8wn) = \Theta(n \log w(\log n + w))$ queries. Additionally, there is an efficient algorithm that sorts $\mathcal{U}$ using at most $8nw \log w \cdot \log(n/(2w))$ queries and time complexity $O(nw^2 \log w \cdot \log(n/w))$.

Proof. We modify Entropy Sort to return `FAIL` if it cannot insert an element (while maintaining a decomposition of the given width). Similarly, we modify Merge Sort to return `FAIL` if the Peeling algorithm cannot reduce the size of the decomposition to the given width. For the first query complexity of the proposition, for $i \in [n]$ we run the modified version of Entropy Sort on the input set $\mathcal{U}$, with width upped bound $w = 2^i$ until the algorithm return without failing. For the second claim we proceed analogously but use the modified Merge Sort. The complexities follow from Theorems 3.7 and 3.8 and from the fact that we reach an upper bound of at most $2w$ on the width of $\mathcal{U}$ in $\log w$ rounds.

□

	Query Complexity	**Time Complexity**	**Lower Bound**
Bin-Insertion Sort	$O(wn \log n)$	$O(n^4 - n^2w^2 + w \log n)$	$\Omega(n(\log n + w))$
Entropy Sort	$O(n \log n + wn)$	Exponential	
Merge Sort	$O(wn \log(n/w))$	$O(w^2 n \log(n/w))$	

Table 3.1: The Algorithms of Chapter 3 and their respective *query* and *time* complexity. w is an upperbound to the width of the unknown poset. In the forth column, we show the lower bound of Theorem 3.2. Note that the query lower bound is also lower bound for the time complexity.

3.5 Conclusions

With the discussion on the *"known width hypothesis"*, we concluded our overview of the Daskalakis et al. results on Poset Sorting in the Width-Based Model [1]. In Section 3.2 we introduced the Bin-Insertion Sort method in Algorithm 5. This was our first attempt for sorting a partially ordered set, where we achieved a query complexity of $O(wn \log n)$ and $O(n^4 - n^2w^2 + w \log n)$ time complexity. Modifying Algorithm 5 we we obtain the Entropy Sort method, presented in the Algorithm 7, of Section 3.3. Entropy Sort achieves *the optimal query complexity* of $O(n \log n + wn)$. Unfortunately the optimality of the query complexity comes at the cost of an *exponential time complexity* in Entropy Sort. An algorithm that balances the query and time complexity is the Merge Sort, presented in the Algorithm 8. of Section 3.4. Merge Sort, achieves a query complexity of $O(wn \log(n/w))$ and $O(w^2 n \log(n/w))$ time complexity. We summarize the above results in Table 3.1. The authors in [1] note that while Entropy Sort is optimal with respect to the queries, the problem if this can be achieved in polynomial time *remains an open problem*.

CHAPTER 4

WIDTH-BASED MODEL: SELECTION ALGORITHMS

An other problem we will consider is the k-Selection problem, where we are required to find *the k-smallest elements* of the poset, as discussed in Definitions 1.14 and 1.15. Remember that for an element $u \in \mathcal{U}$, of a poset $\langle \mathcal{U}, \preceq \rangle$ with height(u) we denote the size of a *maximal* chain C, where for each $c \in C$, $u \succ c$. If S_k the set containing the k-smallest elements, then for each $u \in S_k$, we have height$(u) = k - 1$. Therefore, the minimal elements of S_1 will have height zero.

k-Selection on Poset (Width-Based Model)	
Input:	1. A partially ordered set $\mathcal{U}$. 2. An oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \leq, \not\sim\}$, of a partial order relation $\preceq$ on $\mathcal{U}$. 3. A constant $w \in \mathbb{N}$, with width$(\mathcal{U}, \preceq) \preceq w$.
Output:	The k-smallest elements of the poset.

Selection on Poset (Width-Based Model)	
Input:	1. A partially ordered set $\mathcal{U}$. 2. An oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \leq, \not\sim\}$, of a partial order relation $\preceq$ on $\mathcal{U}$. 3. A constant $w \in \mathbb{N}$, with width$(\mathcal{U}, \preceq) \preceq w$.
Output:	The *minimal* elements of the poset.

In this chapter we will present efficient algorithms, with respect to the query complexity, for the Selection and k-Selection problems. We will also show some *adversarial lower bounds* for these problems. In the adversarial setting, we construct an algorithm that an adversary can follow in order to force an agent to make ”many” queries. We will also break our tradition of presenting only *deterministic* algorithm, and we will show two randomized methods for the problems at hand. The randomized algorithms are quite intuitive, that’s why we chose to include them in this presentation. Similarly to the previous chapter all algorithms and results come from [1]. Thus we remain in the Width-Based Model, we assume that an upper bound to the poset’s width w is provided. In Section 4.1 we present our algorithms for Selection and k-Selection, hence establishing *upper bounds* for these problems. In Section 4.2 we present some *lower bounds* based on the adversarial setting. Lastly, in Section 4.3 we summarize our discussion.

4.1 Upper Bounds

In this section we provide deterministic and randomized upper bounds for k-selection, which are *asymptotically tight for* $k = 1$. The basic idea for the k-selection algorithms is to *iteratively use the sorting algorithms* introduced in Chapter 3 to *update a set of candidates* that the algorithm maintains. We begin with the 1-Selection problem, namely the problem of finding the minimal elements.

4.1.1 Selection Problem

We present our first method in Algorithm 11. Theorem 4.1 characterises the query and time complexity of the algorithm.

```
Algorithm 11: Deterministic Selection
   Input: 1. A set $\mathcal{U}$.
   2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
   3. An upper bound $w$ on the width of the poset.
   Output: The minimal elements of $\langle \mathcal{U}, \preceq \rangle$.
1  $T \leftarrow \varnothing$ // The current set of candidates
2  Let $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$
3  for $t \in [n]$ do
4      Compare $x_t$ to all elements in $T_{i-1}$
5      if $\exists a \in T$ s.t. $a \succ x_t$ then
6          $D \leftarrow \{a \in T \mid a \succ x_t\}$
7          $T \leftarrow T \setminus D$
8          $T \leftarrow T \cup x_t$
9  return $T$
```

We now argue about the correctness and complexity of the Algorithm 11. The algorithm updates a set of *incomparable* elements T. Observe that $|T| \leq w$, since T forms an antichain in $\langle \mathcal{U}, \preceq \rangle$. At the termination of the algorithm T will contain all the elements of height 0. Therefore, we have wn queries and $O(wn)$ time complexity. Hence, we proved the following theorem.

Theorem 4.1. The minimal elements of a poset can be found *deterministically* with at most wn queries in $O(wn)$ time.

Now we proceed with a randomized algorithm for the Selection problem. Algorithm 12 is similar to Algorithm 11, with some modifications to avoid (in expectation) worst-case behaviour. In the following algorithm we denote with $x \overset{\mathcal{R}}{\leftarrow} S$ the *uniformly random* selection of an elements x from the set S.

Let T_t denote the instance of T in the t-th iteration. Also let, $\mathcal{U}_t = \{x_1, x_2, \ldots, x_t\}$ contains the first t elements of $\mathcal{U}$, as selected in Step 5. Then, T_t will contain all the

Algorithm 12: Randomized Selection

```
   Input: 1. A set $\mathcal{U}$.
   2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
   3. An upper bound $w$ on the width of the poset.
   Output: The minimal elements of $\langle \mathcal{U}, \preceq \rangle$.
1  $x \xleftarrow{\mathcal{R}} \mathcal{U}, n \leftarrow |\mathcal{U}|$
2  $T \leftarrow \{x\}$ // The current set of candidates
3  for $t \in [n-1]$ do
4      $x \xleftarrow{\mathcal{R}} \mathcal{U}, \mathcal{U} \leftarrow \mathcal{U} \setminus x$
5      $r \leftarrow |T|$
6      for $j \in [r]$ do
7          $y \xleftarrow{\mathcal{R}} T$
8          if $y \succ x$ then
9              $T \leftarrow T \setminus y$
10         else
11             GOTO 3
12     $T \leftarrow T \cup x$
13 return $T$
```

minimal elements of $\mathcal{U}_t$, hence $|T_t| \leq w$. Note, furthermore, that at step t,

$$\mathbb{P}[x_t \text{ is minimal for } \mathcal{U}_t] \leq \frac{w}{t},$$

since there are *at most* w minimal elements. If x_t is not minimal for $\mathcal{U}_t$, then the expected number of queries needed until x_t is compared to an element $a \in \mathcal{U}_t$ that dominates x_t is clearly at most $\frac{w+1}{2}$. We thus conclude that the expected running time of the algorithm is bounded by,

$$\sum_{i=1}^{w}(t-1) + \sum_{t=w+1}^{n}\left(\frac{w}{t}w + \frac{(t-w)}{t}\frac{(w+1)}{2}\right) = \binom{w}{2} + \sum_{t=w+1}^{n}\frac{1}{2t}(w^2 - w + tw + t)$$
$$\leq \frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w).$$

Hence, we proved the following theorem.

Theorem 4.2 (Daskalakis et al. [1], 2011)**.** There exists a randomized algorithm that finds the minimal elements in an expected number of queries that is upper bounded by $\frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w)$.

4.1.2 k-Selection Problem

We now proceed to examine the case where $k \geq 1$. As in the previous subsection we begin with a deterministic algorithm and then we proceed with a randomized method. The basic idea, behind the deterministic algorithm, is to use the sorting algorithm presented in the previous chapter in order to update a set of candidates for the k-Selection problem.

Algorithm 13: Deterministic k-Selection

```
Input: 1. A set $\mathcal{U}$.
2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
4. A parameter $k$.
Output: The smaller $k$ elements of $\langle \mathcal{U}, \preceq \rangle$.
$C \leftarrow \varnothing$
$t \leftarrow 1$
while $(t-1)wk + 1 \leq n$ do
    $D \leftarrow C \cup \{x_{(t-1)wk+1}, \ldots, x_{\min(twk,n)}\}$
    sort($D$)
    Update $C$ to be the solution of $k$-Selection in $D$
    $t \leftarrow t + 1$
return $C$
```

Clearly, at the end of the execution C will contain the solution to the k-Selection problem. Note that at each iteration, we would have $|D| \leq 2wk$. In the proof of Theorem 3.7 we showed that the Entropy Sort has a query complexity of $2n\log n + 8wn + 2w\log w$. Hence, after "plugging" to the later formula the size of D, we obtain a query complexity of $4wk\log(2wk) + 16w^2k + 2w\log w$ for each iteration. Thus, the total complexity for the $\frac{n}{wk}$ iterations would be,

$$\frac{n}{wk}\left(4wk\log(2wk) + 16w^2k + 2w\log w\right) = 4n\log(2wk) + 16wk + \frac{2n}{k}\log w.$$

On the other hand, if the time complexity is more important to us, we may use Marge Sort of Algorithm 8 to sort D. This will result in $8w^2k\log(2k)$ query complexity, and $O(nw^2\log(2k))$ time complexity. From this discussion we proved the following theorem.

Theorem 4.3 (Daskalakis et al. [1], 2011)**.** The query complexity of the k-Selection problem is at most $16wn + 4n\log(2k) + 6n\log w$. Moreover, there exist an efficient k-Selection algorithm with query complexity at most $8wn\log(2k)$ and time complexity $O(w^2n\log(2k))$.

Next we outline a randomized algorithm achieving a better coefficient of the main term wn. Again we use $x \overset{\mathcal{R}}{\leftarrow} S$ to denote the *uniformly random* selection of an elements x from the set S.

It is clear that C contains the solution to the k-Selection problem. To analyze the query complexity of the algorithm, recall from Theorem 3.8 that $s(w,k) = 8w^2k\log(2k)$ is an upper bound on the number of queries used by the efficient Merge Sort algorithm to sort $2wk$ elements in a poset of width w.

There are two types of contributions to the number of queries made by the algorithm:

1. Comparing x to the set of maximal elements $\texttt{maximal}(C)$.

2. Sorting C and $C \cup D$.

To bound the expected number of queries of the first type, we note that for $t \geq kw + 1$, since $|C \cup D| \leq 2kw$ and the elements are in random order, the probability

Algorithm 14: Randomised k-Selection

```
Input: 1. A set $\mathcal{U}$.
2. An oracle $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ for a poset $\langle \mathcal{U}, \preceq \rangle$.
3. An upper bound $w$ on the width of the poset.
4. A parameter $k$.
Output: The smaller $k$ elements of $\langle \mathcal{U}, \preceq \rangle$.
// Initialization
$C \leftarrow \varnothing$
for $i \in [wk]$ do
    $x \overset{\mathcal{R}}{\leftarrow} \mathcal{U}, \mathcal{U} \leftarrow \mathcal{U} \setminus x$
    $C \leftarrow C \cup x$
sort$(C)$
Remove any element $a$ from $C$ wight height$(a) > k - 1$.
// Begin the main while-loop.
$D \leftarrow \varnothing, t \leftarrow wk + 1$
while $t \leq n$ do
    Let maximal$(C)$ be the set of maximal elements of $C$.
    $r \leftarrow |$maximal$(C)|$
    $x \overset{\mathcal{R}}{\leftarrow} \mathcal{U}, \mathcal{U} \leftarrow \mathcal{U} \setminus x$
    // Compare $x$ with the maximal elements of $C$.
    for $j \in [r]$ do
        $m \overset{\mathcal{R}}{\leftarrow}$ maximal$(C)$, maximal$(C) \leftarrow$ maximal$(C) \setminus m$
        if (height$(m) = k - 1$ and $x \succ m$) or (height$(m) < k - 1$ and
          $x \succ m$) then
            $D \leftarrow D \cup x$
            break
    if for all $a \in C$, $x \not\sim a$ then
        $D \leftarrow D \cup x$
        break
    if $|D| = k$ or $t = n$ then
        sort$(C \cup D)$
        Set $C$ to be the elements of height at most $k - 1$ in $C \cup D$
        $D \leftarrow \varnothing$
return $C$
```

that x ends up in D is at most $\min\left(1, \frac{2kw}{t}\right)$. If x is not going to be in D, then the number of queries needed to verify this is bounded by w. Overall, the expected number of queries needed for comparisons to maximal elements is bounded by wn.

To calculate the expected number of queries of the second type, we bound the expected number of elements that needed to be sorted as follows,

$$\sum_{t=kw+1}^{n} \min\left(1, \frac{2kw}{t}\right) \leq 2kw(\log n - 1)$$

Therefore the total query complexity is bounded above by $wn + 2s(w, k) \log n$, and thus we have proved the following theorem.

Theorem 4.4 (Daskalakis et al. [1], 2011)**.** The k-Selection problem has a randomized query complexity of at most $wn + 16kw^2 \log(2k)$ and time complexity $O(wn + \text{poly}(k, w) \log n)$.

4.2 Lower Bounds

In this Section we will present some lower bounds for the Selection and k-Selection problems. We present these lower bounds in the *adversarial setting*. Until now we were trying to help an agent sort efficiently a poset with access only to an oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$. In this section we will traverse to the other side, and take the role of an *adversary* serving the agent's queries. We assume she wants to force the agent to make as many queries as possible. The adversary is allowed to choose her response to a query *after* receiving it. A response is *legal* if there exists a partial order of width at most w with which this response and all previous responses are consistent.

Since the responses of the adversary are valid, namely consistent with her previous answers and correspond to a valid poset, we can always argue that such a bad instance could appear as an input to our algorithms. In other words, instead of providing a worse case instance $\langle \mathcal{U}, c(\cdot, \cdot), w\rangle$; we produce it on-line depending on the agent's queries. For every, $\mathcal{U}, w$ we will give a method to compute the responses of the oracle $c(\cdot, \cdot)$ with respect to the agents queries.

We will present two key results. In Theorem 4.5 we give a lower bound for the Selection problem. The pseudocode of the adversarial algorithm, for this theorem, is presented on Algorithm 15. On the other hand, the lower bound for the k-Selection problem is presented in Theorem 4.6. For the latter we constrain ourselves to a simple sketch of the proof, omitting most of the details. For an extensive presentation of the proof of Theorem 4.6 we refer to [1].

4.2.1 Selection Problem

The adversarial Algorithm 15, of Theorem 4.5, outputs query responses that correspond to a poset $\langle \mathcal{U}, \preceq\rangle$ of w *disjoint chains*. Along with outputting a response to a query, the algorithm may also announce for a queried element to *which chain it belongs*. In any proof that an element x is *not* a smallest element, it must be shown to dominate at least on other element. The algorithm is designed to so that in order for such a response to be given, x must be queried against at least $w - 1$ other elements with which is incomparable.

The algorithm outputs query responses that correspond to a poset $\langle \mathcal{U}, \preceq\rangle$ of w disjoint chains. Given a query $c(x, y)$, the algorithm outputs a response to the query, and in some cases, it may also announce for one of x, y, or both, to which chain the element belongs. Note this extra information makes the situation *only easier* for the agent doing the queries and trying to determine the k smallest elements. During the course of the algorithm, the adversary will maintain a graph $G = (\mathcal{U}, E)$. Whenever the adversary responds that $x \not\sim b$, she adds an edge (x, y) to E.

We assume that the queries are passed to the adversarial algorithm as *stream*, or a sequence of queries $\mathcal{Q}$, of which the adversary is only able to access one at a time. If some query $c(x, y) \in \mathcal{Q}$ is accessed the adversary needs to respond, *before* she may access the next query. Let $c_t(x)$ be *the number of queries that involve the element* a, out of the first t. Also, let $\texttt{chain}(x)$ be the *chain assignment* that the adversary has announced to the agent for element a. Initially, we have $\texttt{chain}(x) = \texttt{nul}$ for every element $x \in \mathcal{U}$.

Algorithm 15: Selection Adversarial Protocol

```
Input: A sequence of queries $\mathcal{Q}$. A query $q \in \mathcal{Q}$ is $q = c(x, y)$, $x, y \in \mathcal{U}$.
Output: A sequence of responses $\mathcal{R}$. A response $r \in \mathcal{R}$ can be either
        $r = \langle x, \texttt{chain}(x) \rangle$, with $x \in \mathcal{U}$ and $\texttt{chain}(x)$; or $r = \langle x, y, \square \rangle$, with
        $x, y \in \mathcal{U}$ and $\square \in \{\preceq, \succ, \not\sim\}$
// We keep an "incomparability" graph.
$G \leftarrow (\mathcal{U}, \varnothing)$
while $\mathcal{Q} \neq \varnothing$ do
    // Fetch the next query.
    $c(x, y) \leftarrow \texttt{pop}(\mathcal{Q})$
    if $c_t(x) \leq w - 1$ or $c_t(y) \leq w - 1$ then
        $c_t(x) \leftarrow c_t(x) + 1$, $c_t(y) \leftarrow c_t(y) + 1$
        if $c_t(x) = w - 1$ then
            Choose a chain $a \in [w]$, s.t. for every neighbor $v \in N(x)$
              $\texttt{chain}(v) \neq a$.
            $\texttt{chain}(x) \leftarrow a$
            $\mathcal{R} \leftarrow \mathcal{R} \cup \langle x, \texttt{chain}(x) \rangle$
        if $c_t(y) = w - 1$ then
            Choose a chain $a \in [w]$, s.t. for every neighbor $v \in N(y)$
              $\texttt{chain}(v) \neq a$.
            $\texttt{chain}(y) \leftarrow a$
            $\mathcal{R} \leftarrow \mathcal{R} \cup \langle y, \texttt{chain}(y) \rangle$
    if $\texttt{chain}(x) \neq \texttt{chain}(y)$ then
        $\mathcal{R} \leftarrow \mathcal{R} \cup \langle x, y, \not\sim \rangle$
    else
        Let $i, j$ be the indices of $x, y$ respectively in their mutual chain.
        if $i > j$ then
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{\succ\}$
        else
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{\preceq\}$
return $\mathcal{R}$
```

We can easily observe that the output of the algorithm is consistent with a poset

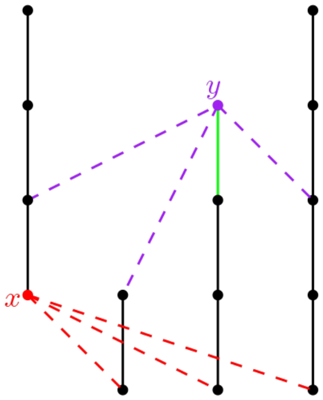


Figure 4.1: An Example for Theorem 4.5. In order to prove that y is not a minimal element the adversary need to inform as of the green edge. In order to to that, we need to query all the dashed purple edges, where the adversary will respond that the elements are incomparable. In order to prove that x is a minimal element we need to query all the red dashed edges.

consisting of w chains that are pairwise incomparable. Also, without loss of generality we can assume that *each chain is non-empty*. The following theorem shows the lower bound on queries achieved from Algorithm 15. Namely, the minimum number of queries the agent will always be forced to make by the adversary.

Theorem 4.5 (Daskalakis et al. [1], 2011)**.** In the adversarial model, at least $\frac{w+1}{2}n - w$ queries are needed in order to find the minimal elements.

Proof. We will prove a lower bound on the number of queries that are required to find a *proof* that the minimal elements are indeed the minimal elements.

In any proof that m is *not* a smallest element, it must be shown that m dominates at least another element. In order to get such a response from the adversary, m must be queried against *at least* $w - 1$ *elements* with which is incomparable. To prove that a minimal element of one chain is indeed minimal, it must be queried *at least* against the minimal elements of the other chains, to rule oput the possibility it dominates one of them. We show both of those checks in Figure 4.1.

From the above discussion, we observe that each element must be compared to at least $w - 1$ elements that are incomparable to it. Hence, the total number of queries of the form $c(x, y)$, where $x \not\sim y$ is at least $\frac{w-1}{2}n$[1]. Thus, we proved our claim.

□

4.2.2 k-Selection Problem

We now proceed to Theorem 4.6, which establishes a lower bound to the number of queries in k-Selection problem. Theorem 4.6 utilizes Theorem 1.22 of Chapter 1. Remember that Theorem 1.22 states that we need at least $n - k + \log\left(\binom{n}{k-1}/k\right)$ queries to solve the k-Selection problem in *total orders*. The algorithm of Theorem 4.6 is based on a similar idea to Algorithm 15 but uses a more specific rule for assigning queried

[1] To see that consider a graph $G = (V, E)$, where $\{x, y\} \in E$, if and only if x is compared with y and $x \not\sim y$. The handshake lemma gives us the required quantity.

elements to chains. The responses are designed to achieve a trade-off between the case when few chains are short, when Theorem 1.22 implies that the number of queries required must be large; and the case when many chains are short, when the algorithm must ensure that the number of pairs declared incomparable is large.

Theorem 4.6 (Daskalakis et al. [1], 2011)**.** Let $r = \frac{n}{2w-1}$. If $k \leq r$. then the number of of queries to solve the k-Selection problem is at least,

$$\frac{(w+1)n}{2} - wk - \frac{w^3}{8} + \min\left\{(w-2)\log\left(\binom{r}{k-1}/k\right) + \log\binom{rw}{k-1},\right.$$
$$\left.\frac{n(w-1)(r-k)}{2r} - \log\left(\binom{r}{k-1}/k\right) + \log\binom{n-(w-1)k}{k-1}\right\}. \tag{4.1}$$

The proof of Theorem 4.6 is quite extensive and exceeds the scope of this thesis. We just present a sketch of the proof here.

A sketch of the proof of Theorem 4.6. The adversarial algorithm outputs query responses exactly a Algorithm 15, except in the case of the t-th query is $c(x, y)$ and $c_t(x) = w-1$ or $c_t(y) = w - 1$. In that case it uses a more specific rule for the assignment of one or both of these elements to chains.

In addition to assigning the elements to chains, the process must also select the k smallest elements in each chain, and Theorem 1.22 gives a lower bound, in terms of lengths of the chains, on the number of queries required to do so.

We think of the assignment of elements to chains as a coloring of the elements with w colors. The specific color assignment rule is designed to ensure that if, at the end, the number of elements e with color κ is small, then there must have been many queries in which the element being colored could not receive color κ because it had already been declared incomparable to an element with color κ.

□

We also give a lower bound on the number used by *randomized* k-Selection algorithms. The authors of [1] conjecture that the randomized algorithm for finding minimal elements, Algorithm 12, essentially achieves the lower bound, though the lower bound we present here is different from that upper bound by a factor of 2. We give the lower bound without a proof. The proof of the theorem *does not* follow the adversarial model.

Theorem 4.7 (Daskalakis et al. [1]. 2011)**.** The expected query complexity of any algorithm solving the k-Selection problem is at least,

$$\frac{w+3}{4}n - wk + w\left(1 - \exp\left(-\frac{n}{8w}\right)\right)\left[\log\left(\binom{n/(2w)}{k-1}/k\right)\right]. \tag{4.2}$$

An immediate result from Theorem 4.7 is the following corollary for the Selection problem, i.e. $k = 1$.

Corollary 4.8. The expected query complexity of any algorithm solving the Selection problem is at least $\frac{w+3}{4}n - w$.

4.3 Conclusions

With the brief discussion on the lower bound for the deterministic k-Selection we conclude this presentation of the results of [1] on the Selection and k-Selection Problems.

	Query Complexity	**Time Complexity**	**Lower Bound**
Det. Selection	$O(wn)$	$O(wn)$	$\Omega\left(\frac{w+1}{2}n - w\right)$
Random. Selection	$\frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w)$	$\frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w)$	$\Omega\left(\frac{w+3}{4}n - w\right)$
Det. k-Selection (Entropy Sort)	$O(16wn + 4n\log(2k) + 6n\log w)$	Exponential	4.1
Det. k-Selection (Merge Sort)	$O(8wn\log(2k))$	$O(w^2n\log(2k))$	4.1
Random. k-Selection	$O(wn + 16kw^2\log(2k))$	$O(wn + \text{poly}(k,w)\log n)$	4.2

Table 4.1: The Algorithms of Chapter 4 and their respective *query* and *time* complexity. w is an upperbound to the width of the unknown poset. In the forth column, we show the query complexity lower bounds. Note that query lower bounds are also lower bound for the time complexity.

In Section 4.1 we presented some *upper bounds* for the two problems at hand. Namely, in Subsection 4.1.1 we examined two algorithms for the Selection problem, of finding the minimal elements of a poset. The first Algorithm 11 is deterministic and achieves $O(wn)$ query and time complexity. The second Algorithm 12 is randomized and achieves $\frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w)$ query and time complexity. In Subsection 4.1.2 we described two more algorithms for the k-Selection problem. The first algorithm we examined for the k-Selection problem is presented in Algorithm 13. The algorithm is deterministic, while the its time and query complexity *depends* on the method we use for sorting. If we want to achieve optimality with respect to the queries, we will use the Entropy Sort of Algorithm 7. This will result in $O(16wn + 4n\log(2k) + 6n\log w)$ query complexity but *exponential* time complexity. On the other hand, if time efficiency is our main concern we may use the Merge Sort, of Algorithm 8. The resulting algorithm will have $O(8wn\log(2k))$ query complexity, and $O(w^2n\log(2k))$ time complexity. The second algorithm we examined for k-Selection is presented in Algorithm 14. The algorithm is randomized and utilizes the time-efficient Merge Sort of Algorithm 8. The algorithms achieves a $O(wn + 16kw^2\log(2k))$ query complexity and $O(wn + \text{poly}(k,w)\log n)$ time complexity, where $\text{poly}(k,w)$ is the polynomial expression that gives the time complexity of Merge Sort.

In Section 4.2 we discussed some lower bounds for the two problems at hand. We discussed lower bounds on the number of queries for deterministic algorithm, and for the *expected* number of queries in the randomized setting. The first two of these lower bounds are presented in the adversarial setting. There we describe an algorithm for an adversary that simulates the oracle function. The adversary will force the agent doing the selection to make many queries. In Subsection 4.2.1 we discussed a lower bound for the deterministic Selection problem. The lower bound we derive to is $\Omega\left(\frac{w+1}{2}n - w\right)$ for the number of queries. In Subsection 4.2.2 we briefly discussed a lower bound for the deterministic k-Selection problem. The lower bound is quite elaborate, and is given in Equation 4.1. In the same Subsection we also gave a lower bound for the randomized k-Selection (see Equation 4.2). From Equation 4.2 we can derive a lower bound for the Selection problem, $k = 1$. We presented the latter lower bound in Corollary 4.8, and show that the expected number of queries is at least $\frac{w+3}{4}n - w$. We summarize these results in Table 4.1.

CHAPTER 5

FORBIDDEN COMPARISONS MODEL

In this chapter we examine the Sorting Problem of Chapter 3, under a different model. We present the *Forbidden Comparisons Model*, which was introduced by Indranil Banerjee and Dana Richards in 2016 [2]. In this model we are given a graph $G = (\mathcal{U}, E)$ which essentially describes a set of restriction; we can compere two elements $u, v \in \mathcal{U}$, only if $\{u, v\} \in E$. We call G *comparison graph.* Again, we assume that we can compare two elements *only by calling a comparison oracle*. On the other hand, in contrast to Chapters 3 and 4, our oracle function will be defined as, $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, where $c(u, v) =\perp$ if the elements are not connected with an edge in G. Therefore, since we assume that the comparison graph is given, we don't need to make an oracle call to ensure that two elements are comparable. An important thing to note is that, the comparison $c(u, v) =\perp$ may be *undefined* for two elements, but we could have that the elements *are comparable*, $u \sim v$. Consider a scenario where $c(u, a) =\preceq$, $c(a, v) =\preceq$, but $c(u, v) =\perp$; from transitivity we can induce that $u \preceq v$ in the underlying, unknown poset. Also, observe that if the comparison graph is a complete graph (clique) $G \approx K_n$ then the underlying poset collapses to a total order. In this setting, our parameter q would be *the number of missing edges* from G, i.e. $q = \binom{n}{2} - |E|$. Note that the parameter q, *quantifies* the distance of G from the complete graph K_n. We give a formal definition for this problem. In the subsequent section, we explore the connection between the comparison and comparability graph in greater detail.

Sorting (Forbidden Comparisons Model)	
Input:	1. A finite set $\mathcal{U}$. 2. A *comparison graph* $G = (\mathcal{U}, E)$. 3. An oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, for an underlying poset $\langle \mathcal{U}, \preceq \rangle$, where $c(a, b) =\perp$ iff $\{a, b\} \notin G$. 4. A parameter $q = \binom{n}{2} - |E|$.
Output:	The partial order $\preceq$.

In [2] Banerjee and Richards present an algorithm with query complexity of $O((q + n) \log n)$, the first non-trivial sorting algorithm in the Forbidden Comparisons Model. Based on this work Biswas et al. [3] propose a refinement of Banerjee's and Richards's method with $O(q + n) \log(n^2/q)$ query complexity. For $q = \Theta(n^2)$ the latter method achieves better query-efficiency than Banerjee's and Richard's algorithm. In [3] the authors, also present a lower bound to the problem along with a study of some special cases. In this chapter, we will follow the presentation of [3], but we will also discuss briefly the key results of [2]. We organise this chapter as follows. In Section 5.1 we introduce some new concepts regarding directed graphs, while we explore further their connection to posets in our model. In Section 5.2 we discuss the connection between the comparability and comparison graph. There, we also present a characterization of the comparability graphs due to Gallai. In Section 5.3 we present the sorting algorithm of [2], along with the improvements of [3]. In that Section we also present some lower bound to our problem. In Section 5.4 we present some results on the query-efficiency of our sorting, for special cases of the input comparison graph. Finally, in Section 5.5 we summarise our discussion on the sorting problem in the Forbidden Comparisons Model.

5.1 Graph Orientations & Tournaments

In this section we introduce some new notion regarding directed graphs. We explore their relation with undirected graphs and their connection with partial orders. We define the notion of *orientation* of an undirected graph, while we focus on orientations that respect the transitivity. We define a special class of directed graphs, the tournaments, that are closely connected to total orders. Since any partial order, contains a total order, e.g. its maximal chain, we will use tournaments to sort a partial order in the sequel.

We begin with some definitions. We call a directed graph, with no directed cycles; a *directed acyclic graph* or DAG. Directed acyclic graphs will be of great importance in our presentations as they will be useful in the study of partial orders. An important notion in directed acyclic graphs is the *topological order*. We give the following definition.

Definition 5.1 (Topological Order)**.** Let $D = (V, A)$ be a *directed acyclic graph*. We call a *topological order* of D a sequence of its nodes $v_1, v_2, \dots, v_n$, such that if $(v_i, v_j) \in A$, then $i < j$.

In this chapter we will examine a problem where we are given a undirected graph and we will be required to find a suitable *orientation* of its edges, for the resulting directed graph to correspond to a valid partial order. We give the following definition.

Definition 5.2 (Graph Orientation)**.** Let $G = (V, E)$ an *undirected* graph. Also, let $\phi\colon E \to V \times V$ be a *1-1 correspondence*. We call the resulting directed graph $D = (V, \phi(E))$ an *orientation* of G.

A *transitive orientation* is an orientation that respects the transitivity. Namely, let $G = (V, E)$ be a undirected graph, and $D = (V, A)$ an orientation of G. We say that *D respects the transitivity* if for every $(a, b), (b, c) \in A$, then $(a, c) \in A$. An important notion, for our analysis will be this of an orientation of a clique K_n.

Definition 5.3 (Tournament)**.** Let K_n be a *clique* on n nodes. Let D be an orientation of K_n, we will call D a *tournament*. Note that, essentially in a tournament D every two nodes are connected with *a single* arc.

Observe that any transitive tournament is acyclic. Consider a path $v_1, \dots v_k$ in a tournament $D = (V, A)$. Now, since we have a tournament either (v_1, v_k) or (v_k, v_1) is

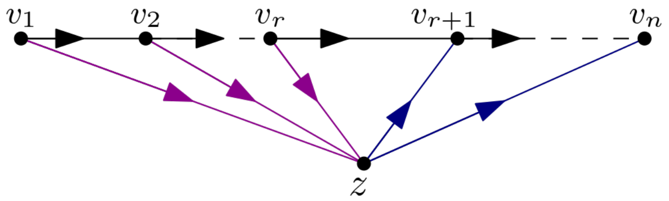


Figure 5.1: The figure for Rédei's Theorem 5.5 proof.

an arc. If our tournament is transitive then $(v_1, v_k) \in A$, hence we don't have a cycle. On the other hand, the only way for us to not have a cycle is to not include (v_k, v_1) as an arc. Thus, we proved the following characterization of a tournaments acyclicity.

Proposition 5.4. Let $D = (V, A)$ be a tournament. Then D is acyclic, *if and only if* D is transitive.

A *directed Hamiltonian path* is a path in a directed graph D that visits each node of the graph only once. We present the following key property of tournaments due to Rédei.

Theorem 5.5 (Rédei, 1934)**.** Any tournament $D = (V, A)$ on n vertices contains a *Hamiltonian path*, i.e., a directed path on all n vertices.

Proof. Suppose we have a directed path $v_1, \ldots, v_k$ which does not contain all the vertices of the graph. Let z be any vertex not on this path. If (z, v_1) is an arc, i.e. $(z, v_1) \in A$ we can insert z at the beginning of the sequence. If $(z, v_1) \notin A$, then (v_1, z) is an arc, since we have a tournament. In this case, if (z, v_2) is an arc, then we can insert z between v_1, v_2 to get a directed path from v_1 to v_k which includes z.

If (z, v_2) is not an arc, since we have a tournament, then (z, v_2) is an arc; and we let r be the greatest integer for which $(v_1, z), (v_2, z), \ldots, (v_r, z)$ are arcs (see Figure 5.1). If $r < k$ we can insert z between v_r and v_{r+1} as we did in the previous cases and complete our v_1, v_k-path. On the other hand, if $r = k$, we just add z at the end.

Therefore, if we have a directed path which does not contain all the vertices, we can always insert another vertex into this directed path. Since the graph is finite, we will eventually get a directed path that contains all of the vertices.

□

Note that the proof of Theorem 5.5 is constructive. It also gives us an algorithm to compute the Hamiltonian path in $O(n \log n)$ time. Observe that in order to achieve this time-complexity, we need to do *binary search* to compute r in $O(\log n)$ time. From Theorem 5.5 we get an interesting corollary; for transitive tournaments not only there always exists a Hamiltonian path in a tournament, but it is also unique.

Corollary 5.6. Let $D = (V, A)$ a *transitive* tournament on n vertices. Then, there is a *unique* Hamiltonian path $P = v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n$.

Proof. Let P' be another Hamiltonian path, where v_i, v_j appear in P' in reverse order than this of P. Consider the subpath of P', from v_j to v_i. From transitivity, we have that $(v_j, v_i) \in A$. A contradiction, since D is a tournament.

□

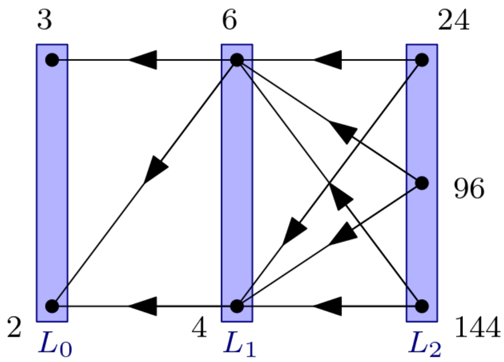


Figure 5.2: The *layer decomposition* of the directed orientation, of the Hasse diagram of Example 1.12.

On the other hand, observe that in a tournament a topological order, will always be a Hamiltonian path. Indeed, let $v_1, v_2, \ldots, v_n$ be a topological order. Also, let v_i, v_{i+1} be adjacent in this sequence. From since we have a tournament either (v_i, v_{i+1}), or (v_{i+1}, v_i) is an arc. Since, v_i, v_{i+1} are adjacent in the topological order we can only have that (v_i, v_{i+1}) is an arc. But, from Corollary 5.6 we can only have a single Hamiltonian path in a transitive tournament. Therefore, we have a unique topological order for a transitive tournament.

Corollary 5.7. Let $D = (V, A)$ be a *transitive* tournament on n vertices. Then, there is a *unique* topological order $v_1, \ldots, v_n$.

We can rewrite Corollary 5.7, to state that directed acyclic graphs have a unique topological order. It is sometimes more intuitive to underline the acyclicity of a graph than its transitivity. In Section 5.4 we will use Corollary 5.7 in this latter form.

5.1.1 Layer Decomposition & Linear Extension

Before we close this section, we make some observations regarding the number of topological orders of a given directed acyclic graph. The case which we are interested in is, naturally when a given DAG represents a poset. We give the following definition, that is valid in any directed acyclic graph, regardless of it being transitive of not.

Definition 5.8 (Layer Decomposition of a DAG)**.** Let $D = (V, A)$ being a directed acyclic graph. We consider a decomposition $\mathcal{L} = \{L_i \mid i \in [k]\}$, for some $k \in \mathbb{N}$, of the vertices of D, such that,

$$\left.\begin{array}{ll} L_0 = & \{v \in V \mid d_{\text{out}}(v) = 0\} \\ L_{i+1} = & \{v \in V \mid \text{ if there is } (v, u) \in A, u \in L_j, j \leq i\} \end{array}\right\}. \tag{5.1}$$

We call the decomposition $\mathcal{L}$ a *layer decomposition* of D.

In Figure 5.2 we show the layer decomposition of our running example, Example 1.12. An important thing to note is that, if we demand each node to be included to the set L_i, with the smallest index i, such that the Equation 5.1 holds, then the layer decomposition is *unique*. In the sequel, when we are referring to the layer decomposition of a DAG, we will be assuming that the latter uniqueness condition holds. We give the following definition, regarding a sup-relation that includes the relations of a given poset, but constitutes a total order.

Definition 5.9 (Linear Extension)**.** Let $\langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$ be a poset on a finite set $\mathcal{U}$. We call a relation $\mathcal{R}'_{\preceq}$ a *linear extension* of $\langle \mathcal{U}, \mathcal{R}_{\preceq} \rangle$, if $\mathcal{R}'_{\preceq} \supseteq \mathcal{R}_{\preceq}$ and $\mathcal{R}'_{\preceq}$ is a total order.

From the above discussion, we hope to come naturally the following. Let ℓ_0 be an ordering of the L_0 layer of a layer decomposition, ℓ_i be an ordering of the layer L_i, etc. Then, $\ell_0 \ell_1 \ell_2 \cdots \ell_k$ be a *concatenation* of the layers' orderings. Then, $\ell_0 \ell_1 \ell_2 \cdots \ell_k$ will be a *topological sort* of the given directed acyclic graph. On the other hand, if the DAG is transitive and represents a poset, then *every topological sort is a linear extension*. A simple combinatorial result is the following, regarding the number of linear extensions of a poset.

Proposition 5.10. Let $\langle \mathcal{U}, \preceq \rangle$ be a poset in a finite set $\mathcal{U}$. Also, let $D = (V, A)$ be a directed acyclic graph, that represents the relations of the poset. Lastly, assume $\mathcal{L} = \{L_0, L_1, \ldots, L_k\}$ be the unique layer decomposition of D. Then, the number of linear extensions of $\langle \mathcal{U}, \preceq \rangle$ is ν, where,

$$\nu = \prod_{i=0}^{k} |L_i|!$$

Remember that earlier in this section we argued that a *transitive* tournament has a *single* topological order (and a unique Hamiltonian path). Hence, a transitive tournament has a *unique* linear extension. Therefore, the underlying poset of a transitive tournament, is actually a total order.

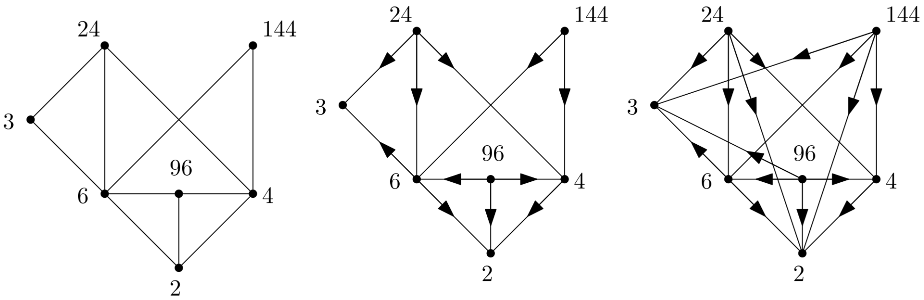
5.2 Comparability & Comparison Graphs

In this section we formalize the notions discussed in the introduction of this chapter and explore further the connection between the comparison and comparability graphs. We begin with a formal definition of the comparability graphs.

Definition 5.11 (Comparability Graph)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset. Let $\widetilde{G} = (\mathcal{U}, \widetilde{E})$ be the graph, where for two nodes $x, y \in \mathcal{U}$ there is an edge $\{x, y\} \in \widetilde{E}$, *if and only if* the two corresponding elements of the poset are related, i.e $x \sim y$. We call $\widetilde{G}$ the *comparability graph* of $\langle \mathcal{U}, \preceq \rangle$.

Note that a comparability graph of a poset results from the directed graph of the poset, if we remove the direction from the edges. On the other hand, we can imagine the comparability graph as the Hasse diagram of the poset, if we add the transitive edges. An arbitrary graph, in general, cannot be the the comparability graph of a poset. In other words, there are graphs where we cannot find a proper *orientation of its edges*, in order for the induced binary relation to be a partial order. Hence, we also call these graphs *transitivity orientable*. Later, we will give a characterization of the comparability graphs in Gallai's theorem, which we present without a proof. Let K_ℓ be the ℓ-clique and $K_\ell \subseteq \widetilde{G}$, for some comparability graph $\widetilde{G}$. Then the *subposet* induced on the nodes of K_ℓ will be a *total* order. On the other hand, note that we *cannot* have an odd cycle ($n > 3$) as an induced subgraph in any comparability graph. We present this intuitively obvious fact in the theorem bellow, without a proof.

Theorem 5.12 (Ghouila-Houri, 1962)**.** Let $G = (V, E)$ be a graph. Then G is a comparability graph *if and only if* there is no sequence $x_1, x_2, x_3, \ldots, x_{2n+1}$ of (not necessarily distinct) vertices from V with $n \geq 2$ such that $\{x_i, x_{(i+1) \mod 2n+1}\} \in E$, but $\{x_i, x_{(i+2) \mod 2n+1}\} \notin E$.



(a) Comparison graph.

(b) Assigning orientation to the comparison graph.

(c) Computing the transitive edges.

Figure 5.3: The comparison graph (5.3a), the orientation of the comparison graph (5.3b), and the transitive closure (5.3c) of the poset of Example 1.12. Note that the comparison graph in Figure 5.3a is *isomorphic* to the Hasse diagram of the poset. In contrast to a Hasse diagram, we have no constriction on the way we draw the graph on the plane, i.e. we don't need to draw a higher than b if $a \succ b$. A *comparability* graph would be identical to the graph of Figure 5.3c, *if we remove the orientation*.

As we established in the introduction, a similar notion to comparability graphs, are the *comparison graphs* introduced by I. Banerjee and D. Richards [2]. A comparison graph, is a comparability graph, where we are allowed to omit some edges. Observe, that we can omit all the edges that can be induced from the transitive closure, without harming the underlying partial order. Therefore, the Hasse diagram of a poset can be regarded as a *comparison graph*, whereas it cannot, in general, be considered a *comparability graph*, due to the lack of the transitively induced edges. We give a formal definition.

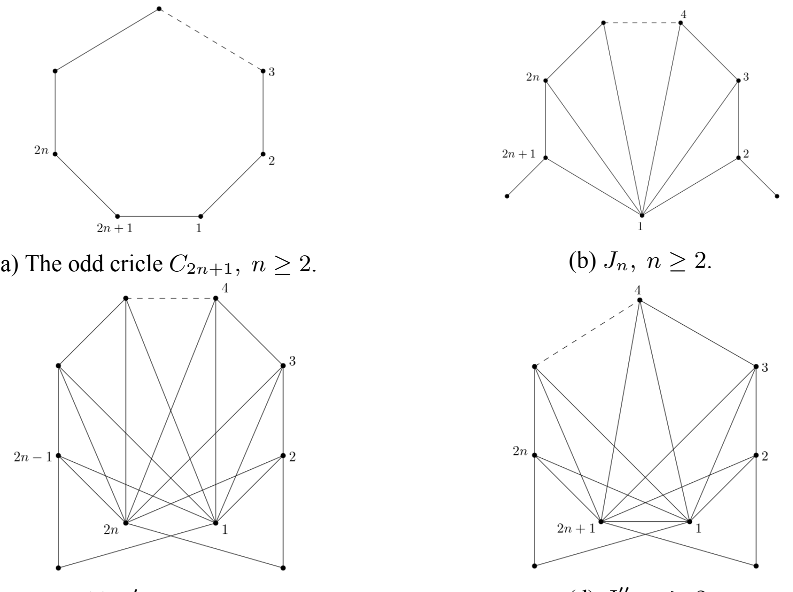
Definition 5.13 (Comparison Graph)**.** Let $\langle \mathcal{U}, \preceq \rangle$ be a poset. We call $G = (\mathcal{U}, E)$ a *comparison graph*, if for every $x, y \in \mathcal{U}$ that are connected with an edge $\{x, y\}$, then $x \sim y$ in $\langle \mathcal{U}, \preceq \rangle$. On the other hand, we allow to elements $x, y \in \mathcal{U}$ to be related $x \sim y$, but not to be connected in G with an edge $\{x, y\} \notin E$.

Observe that from Definitions 5.11 and 5.13, we would have $G \subseteq \widetilde{G}$. Also, note that since a comparison graph is essentially a comparability graph, where we have omitted some edges, and because omitting edges cannot introduce cycles; the comparison graphs have no odd cycles. Hence, a comparison graph is also bipartite.

The notion of a comparison graph raises some interesting algorithmic questions. The problem that we will analyse in this chapter is to determine a valid partial order from a comparison graph. This, essentially, takes two steps; firstly, to determine a valid orientation of the edges; then, to deduce the transitive closure of this graph. In Figure 5.3 we show the subsequent steps of this process, with some remarks on the graph theoretic notions we introduced.

5.2.1 Gallai's Theorem

Perhaps one the most important results in partially order theory, regarding the comparability graphs is the theorem due to Gallai (1967). We follow the elegant presentation of [8]. In some cases, when a class of discrete structures is *closed under a notion of*



(a) The odd cricle $C_{2n+1},\ n \geq 2$.

(b) $J_n,\ n \geq 2$.

(c) $J'_n,\ n \geq 2$.

(d) $J''_n,\ n \geq 2$.

Figure 5.4: Part 1 of the Gallai's forbiden graphs $\mathcal{C}$. Each subfigure depicts an infinite family of graphs. These graphs cannot be a subgraph of a comparability graph.

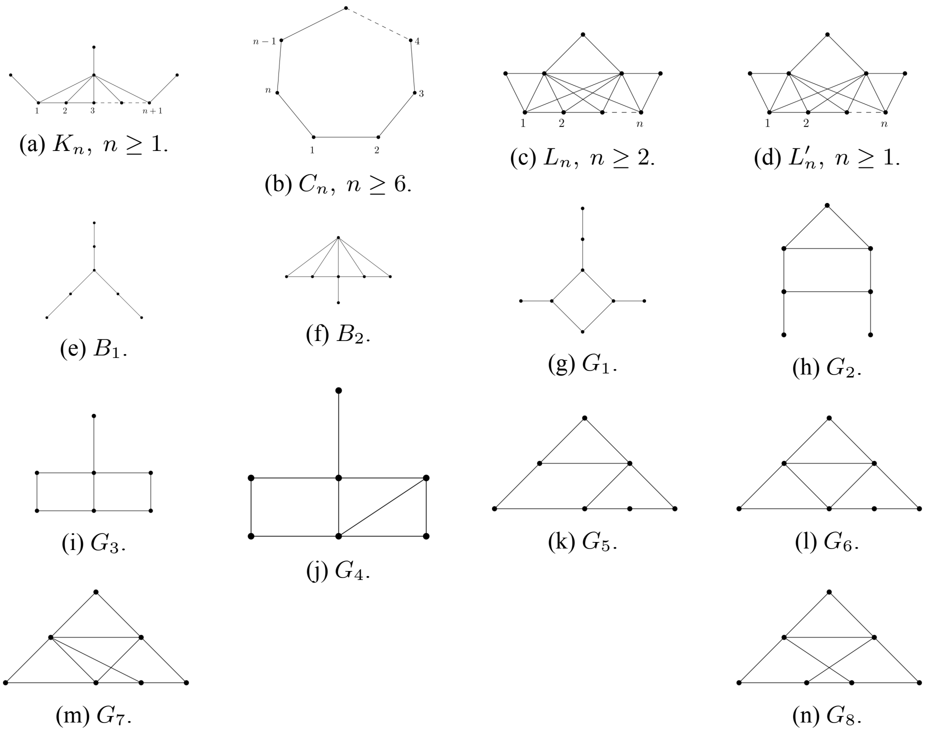
substructure, it can be characterized by a *minimum list of forbidden substructures*. Perhaps the best known example of this type is Kuratowski's theorem which asserts that a graph is planar if and only if it does not contain a subgraph which is isomorphic to a subdivision K_5 or $K_{3,3}$. Note that from its definition the notion of a transitive orientable graphs is *closed under the notion of subgraph*. Hence, Gallai's theorem gives exactly the aforementioned minimum list of the forbidden subgraphs. We denote this list with $\mathcal{C}$. Following Gallai, we divide the family $\mathcal{C}$ of forbidden graphs into parts, see Figures 5.4, 5.5. The graphs in Figure 5.4 belong in $\mathcal{C}$, while the *complements* of the graphs in Figure 5.5 belong in $\mathcal{C}$. Note that each graph in $\mathcal{C}$ is not a comparability graph, *but every proper nonempty induced subgraph of a graph in $\mathcal{C}$ is a comparability graph.*

Theorem 5.14 (Gallai)**.** A graph $G = (V, E)$ is a *comparability graph* if and only if G does not contain an induced subgraph isomorphic to a graph in the collection $\mathcal{C}$ described in Figures 5.4, 5.5.

We do not include here a proof of Theorem 5.14, or even an outline of the proof, since it is difficult, time consuming, and out of the scope of this thesis. In the sequel we focus our attention on *comparison graphs*, rather than comparability graphs. We discuss here briefly some important results of this area, as the intuition behind the two notions is similar. For the reader interested in this area, we refer to [8], and Kelly's survey paper [24].

5.3 Sorting Under Forbidden Comparisons

In this section we will present a sorting algorithm for the Forbidden Comparisons Model. We will follow the presentation in [3].



(a) $K_n,\ n \geq 1$.

(b) $C_n,\ n \geq 6$.

(c) $L_n,\ n \geq 2$.

(d) $L'_n,\ n \geq 1$.

(e) B_1.

(f) B_2.

(g) G_1.

(h) G_2.

(i) G_3.

(j) G_4.

(k) G_5.

(l) G_6.

(m) G_7.

(n) G_8.

Figure 5.5: Part 2 of the Gallai's forbiden graphs $\mathcal{C}$. The first four subfigures depict an infinite family of graphs. The *complements* of these graphs cannot be a subgraph of a comparability graph.

5.3.1 Upper Bounds

We begin by reviewing the algorithm in [2]. The central claim in the algorithm in [2] is the following lemma, which is not stated explicitly, but is presented as a collection of lemmas culminating in the result in Section 2.2 of [2]. We omit a the proof of Lemma 5.15.

Lemma 5.15. Let $G = (V, E)$ be a comparison graph with $n = |V|$, and $q = \binom{n}{2} - |E|$. If $q < n^2/320$ [1] there is an approximate *median vertex* m, that is *greater* than at least $n/40$ elements and less than $n/40$ elements. Furthermore m cannot be compared with at most $O(q/n)$ elements, and it can be found using $O(q + n)$ queries.

The intuition behind Lemma 5.15 is that we can always find a sufficiently big clique as a subgraph of the comparison graph. Assume that $q \leq cn$, where n is the number of vertices and $c \in \mathbb{N}$ a constant. Also, let $\overline{G} = (V, \overline{E})$ be the *complementary graph*, of the comparison graph G. From our hypothesis we would have $|\overline{E}| = q$. We denote with R the vertices of the comparison graph G with many missing edges. Note, that these vertices would have many edges in the complementary graph, namely $R = \{v \in V \mid d_{\overline{G}}(v) > c_1\}$, for some constant $c_1 \in \mathbb{N}$. Applying the *Handshake Lemma* on the complementary graph we would have $\sum_{v \in V} d_{\overline{G}}(v) = 2q$. Thus, for the size of R we have, $|R| \leq \frac{2q}{c_1} = \frac{2cn}{c_1}$. Also, let $S = V \setminus R$ the set with few missing edges

[1] The constant used in [2] is 200, instead of 320, used by the authors in [3].

in the comparison graph G. Observe that we have $S = \{v \in V \mid d_{\overline{G}}(v) \leq c_1\}$ or $S = \{v \in V \mid d_G(v) \geq n - 1 - c_1\}$. Again, from the Handshake Lemma on $\overline{G}$, we have $|S| \geq \frac{2q}{c_1} = \frac{2cn}{c_1}$. Now, we set the constant c_1 to be $c_1 = 4c$. Therefore, $|S| \geq \frac{n}{2}$. From the above discussion we established that there is a sufficiently large set S, which contains vertices with sufficiently large degree. We will use this fact to show that we can always find a large enough clique as a subgraph of the comparison graph.

Proposition 5.16. Let $G = (V, E)$ be a comparison graph with $n = |V|$, and $q = \binom{n}{2} - |E|$. Also, assume $q \leq cn$ for some constant $c \in \mathbb{N}$, and $S = \{v \in V \mid d_G(v) \geq n - 1 - 4c\}$ be the set with vertices of large degree. Then, there exists some $X \subseteq S$, such that $G[X]$ is a *clique*, and,

$$|X| \geq \frac{n}{2(4c + 1)}$$

Proof. We give an algorithm for computing X. The algorithm is quite simple and follows a greedy strategy. We choose an arbitrary vertex $v \in S$. We also initialise the active neighborhood to be $N = N_G(v) \cap S$. In each iteration, we pick a vertex $u \in N$, and update N to be $N' = N \cap N(u)$. We repeat this process until the active neighborhood is depleted, i.e. $N = \varnothing$.

Observe that since $d(v) \geq n - 1 - 4c$, then $d(v) \geq n/2 - 4c$, or $d(v) \geq |S| - 4c$. Also note that in each iteration we loose some of the active neighbors in N. More precisely we have loose the neighbors of the new vertex u, *that are not* already in N. In the worst case, these would be $4c + 1$ nodes. Therefore, $|X| \geq \frac{|S|}{4c+1} = \frac{n}{2(4c+1)}$. □

Note that, in order to compute X we do not make any oracle calls. On the other hand, since $G[X]$ is a clique, the subposet $\langle X, \preceq \rangle$ would be a *total order*. Using Algorithm 1 of Section 1.4, we can find a median in X in *linear* $O(|X|)$ queries.

We are now ready to present the algorithm due to Biswas et al. [3]. The algorithms in [3] and [2] are essentially the same, with the key difference being that the recursion in [3] stops earlier, which improves the bounds. The output of the algorithm is an orientation of some of the edges of the comparison graph. The orientation of the remaining edges can be deduced using transitivity. We present the pseudocode in Algorithm 16.

Theorem 5.17 (Biswass et al. [3], 2017)**.** Sorting a comparison graph with q forbidden edges can be done in,

$$\min\left\{|E|, O\left((q + n)\log\left(\frac{n^2}{q}\right)\right)\right\}$$

queries.

Proof. Algorithm 16 is called with depth $d = 0$. The algorithm checks if the value of q or the depth d of the recursion is less than a threshold and then it breaks the problem into two *disjoint* subproblems using $O(q + n)$ queries by Lemma 5.15. Suppose at level ℓ of the recursion, the sizes of the subproblems are $n_1, n_2, \ldots, n_t$ and the number of missing edges in these subproblems is $q_1, q_2, \ldots, q_t$ respectively. The algorithm either breaks them into smaller subproblems or queries every edge in that subproblem (with $q_i \geq n_i^2/320$ missing edges) in which case the algorithm performs at most $160q_i$ queries. The query cost incurred by the incomparable elements at any internal node of the recursion tree is also $O\left(\frac{q_i}{n_i} n_i\right) = O(q_i)$. In either case, the total number of queries

```
Algorithm 16: Sort
  Input: 1. A finite set $\mathcal{U}$.
  2. A comparison graph $G = (\mathcal{U}, E)$.
  3. An oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, for an underlying poset $\langle \mathcal{U}, \preceq \rangle$,
  where $c(a,b) = \perp$ iff $\{a,b\} \notin G$.
  4. A parameter $q = \binom{n}{2} - |E|$.
  5. A parameter $d$, for the depth of the recursion.
  Output: An orientation of $G$ as a directed graph $D = (V, A)$.
  if $q \geq \frac{n^2}{320}$ or $d = \log\left(\frac{n^2}{q}\right) / \log\left(\frac{40}{39}\right)$ then
      Query every edge in $E$ and output their orientations.
  else
      Find an approximate median vertex $m$ using Lemma 5.15.
      Compare $m$ with all its neighbors in $V$ and output their orientations.
      $V_L \leftarrow \{v \in V \mid v \preceq m\}$, and $V_H \leftarrow \{v \in V \mid v \succ m\}$
      $V_{\text{incomp}} \leftarrow \{v \in V \mid \{v, m\} \notin E\}$
      Compare every vertex in $V_{\text{incomp}}$ with all its neighbors in $V$ and output
        their orientations.
      Let $q_L$ be the number of missing edges in $G[V_L]$ and $q_H$ be the number of
        missing edges in $G[V_H]$
      Sort($V_L, G[V_L], c(\cdot,\cdot), q_L, d+1$)
      Sort($V_H, G[V_H], c(\cdot,\cdot), q_H, d+1$)
```

done at this level is at most $\sum_{i=1}^{l}(q_i + n_i) = O(q+n)$. Thus, at any level of the recursion tree, the algorithm makes at most $O(q+n)$ queries.

The algorithm is essentially the same as that of [2], except that it is forced to stop the recursion, when the depth of the recursion i is $d = \log\left(\frac{n^2}{q}\right) / \log\left(\frac{40}{39}\right)$. At this point the number of subproblems would be at most $O(n^2/q)$ and the size of each subproblem would be at most $(39/40)^i n = q/n$, since each subproblem has at most 39/40 fraction of the vertices of its parent subproblem by Lemma 5.15.

Even if all these subproblems were complete graphs, the total number of edges in all subproblems would be at most $O(n^2/q \cdot q^2/(2n^2)) = O(q)$. At this point we just ask all the edge queries without recursing any further using $O(q)$ edge queries. The algorithm creates a recursion tree which has $O\left(\log \frac{n^2}{q}\right)$ levels and queries $O(q+n)$ edges at each level. Thus the total edge queries made by the algorithm is $O\left((q+n)\log\left(\frac{n^2}{q}\right)\right)$. If $|E| < (q+n)\log\left(\frac{n^2}{q}\right)$, then the algorithm just asks all the edge queries without optimizing in any way.

□

In the above theorem, we have just analysed the query complexity and ignored the time it takes to find the queries to make. One can easily see that the rest of the running time remains as $O(n^2 + \sqrt{q}^{\omega})$ as shown in Theorem 6 of [2]. This running time is essentially required to compute the transitive closure of the directed graph, in order to deduce the missing relations.

5.3.2 Lower Bounds

We now exhibit lower bounds on the number of edge queries needed to sort a graph $G = (V, E)$ in terms of $|V| = n$ and $q = \binom{n}{2} - |E|$, the number of missing edges. When q is large, we have the following lower bound.

Lemma 5.18. There exists a graph with $q \geq n^2/4$ and an orientation such that, $\Omega(|E|)$ edge queries are needed to sort the graph.

Proof. The graph which we construct is a *complete bipartite* graph $G(A \uplus B, E)$, with $|A| = |B|$. We force the oracle function to orient the edges from A to B. Here the number of missing edges, as well as the number of edges present is, roughly $n^2/4$. The edges are oriented from A to B forcing the algorithm to query every edge, as the algorithm cannot deduce any of the edges using transitivity. If the algorithm fails to query an edge, it will have a choice of flipping its direction.

□

For $q < n^2/4$, we have the following bound.

Theorem 5.19 (Biswas et al. [3], 2017)**.** When $q < n^2/4$, there exists a graph and an orientation of the edges such that any algorithm hat to make $\Omega(q + n \log n)$ oracle calls to sort the graph.

Proof. In this case, the graph we construct consists first of a complete bipartite graph $B = (X \uplus Y, E_B)$, where X and Y have size roughly $\sqrt{q}$ each such that it has q edges and has q edges missing. Then we construct a clique K on the remaining $n - 2\sqrt{q}$ vertices and we maintain a total order among those vertices. And we add all the edges between K and B. If a query comes between a vertex $b \in B$ and a vertex $k \in K$, the oracle directs this edge from b to k. If the edge query is between two vertices inside the complete graph, the oracle function will answer consistently with the total order we chose. If the edge query is between two elements inside the bipartite graph, the oracle always directs the edges from X to Y.

The numbers of edge queries required to sort the complete graph K would be $\Omega(n \log n)$ and the number of edge queries required to sort the bipartite graph B is at least $\Omega(q)$ from Lemma 5.18, which gives a lower bound of $\Omega(q + n \log n)$ edge queries.

□

5.4 Special Cases

In general, the number of edge queries needed to sort a poset can depend on both its size n and the number of forbidden pairs q. However, when the comparison graph accompanying the poset has some additional structure, the number of edge queries needed can be at most $O(n \log n)$, and more importantly *independent* of q. In this section we show that when the comparison graph is *chordal* or *transitively orientable*, the graph can be sorted by making $O(n \log n)$ edge queries. In fact, both algorithms presented below output linear extensions (i.e. topological orderings) of the input posets.

Topologically sorting a general directed acyclic graph $D = (V, A)$ needs $\Omega(|V| + |A|)$ running time, while the algorithms in this section make $O(n \log n)$ queries. This is due to the fact that the algorithms exploit the additional information about the input poset which the comparison graph provides and more importantly, we only care about the query complexity, and not bother about finding the orientation of every edge. One can *deduce* the edge directions from the topological sort (or the layer decomposition) of the vertices *using transitivity*, which have no effect on the query complexity. Therefore, in both of the following cases, the main computational step will be to find the topological sort and a layer decomposition.

The algorithms in the next two subsections have the following general outline.

1. Pick an appropriate (constant-size) subposet of the input and *topologically* sort it.

2. Iteratively extend the topological ordering by inserting one element at a time using a binary search type procedure among its neighbors.

3. Use transitivity to compute the orientation of the remaining edges.

We use the same ideas as in Theorem 5.5. In order to insert a new vertex v we do a binary search in the topological order $v_1, v_2, \dots, v_t$. This way we find if $v \prec v_1$, or $v_t \prec v$, or if there are some adjacent vertices v_i, v_{i+1}, such that $v_i \prec v \prec v_{i+1}$. If the comparison graph is a complete graph, then from Corollary 5.6 any any directed acyclic orientation of its edges has a *unique directed Hamiltonian path*. Hence, from Corollary 5.7 has a *unique topological ordering*. In this case, the simple insertion by binary search is suffices to sort our comparison graph. We will, also show that the binary insertion works even if the comparison graph is not a complete graph. In this direction, we make the following observation, it comes as an immediate result from Corollary 5.7.

Lemma 5.20. Let $D = (V, A)$ be a *directed acyclic graph*, and let $S \subseteq V$ be a subset of vertices, such that $D[S]$ is a *tournament* on the vertices of S. Let $v_1, \dots, v_s$ be the unique topological order of vertices of S. In any topological order of the vertices in V, the elements of S appear in the unique topological order within S.

We will exploit Lemma 5.20 in the first special case, regarding chordal graphs. There, we will see that we will only need to find the "local" order of a node v we would like to insert. In other words, we will only need to sort the new node s with respect to its neighbors, which will form a *total order*. Hence, we can sort the inserted vertex, with respect to its neighbors in $O(n \log n)$ time, using for example Merge Sort. We note that this is a special characteristic of chordal graphs, as we shall see in more detail in the sequel. In general graphs, we have no guarantee the that neighborhood of a vertex is a clique, i.e., the restriction of the poset in $N(v) \cup v$ is a total order. Let v, u be two nodes with disjoint neighborhoods. We could deduce the remaining edges between $N(v) \cup v$ and $N(u) \cup u$ using transitivity.

On the other hand, despite we use a similar strategy in comparability graphs, the correctness of the method will be based on a completely different fact. In comparability comparison graphs, we have the advantage that two nodes u, v are related in the underlying poset, *if and only if* $\{u, v\}$ is an edge. Assume we would like to sort the node v. Also, consider the layer decomposition $\mathcal{L} = \{L_0, \dots, L_k\}$. We only need to assign v to the correct layer L_i. From Definition 5.8 we only need to query the neighbors of v, in the worst case. Now, for every other $u \in L_j$, with $i < j$ we would have $v \prec u$ *if and only if* the edge $\{v, u\}$ exists. Therefore, we can deduce the orientation of such edges, *without* querying the oracle. In the following sections we explore the details of the process we discussed.

5.4.1 Chordal Graphs

An undirected graph is *chordal* if every circle of length greater than three has a chord, i.e., an edge connecting two non-adjacent nodes. Chordal graphs form a well-studied class of graphs as they can be recognized in *linear time* [25]. Moreover, several problems that are hard on other classes of graphs, such as graph coloring, can be solved in polynomial time for chordal graphs [26].

In a graph G, a vertex v is called *simplicial* if and only if the subgraph of G induced by $N(v) \cup v$ form a clique. A graph G on n vertices is said to have a *perfect elimination ordering* (PEO) if and only if there is an ordering $v_1, \ldots, v_n$ of its vertices such that v_i is simplicial in the subgraph induced by $v_1, v_2, \ldots, v_i$. We use the following key result regarding perfect elimination ordering in chordal graphs.

Theorem 5.21 ([26, 25])**.** Every chordal graph has a perfect elimination ordering which can be found in $O(|V|+|E|)$ time. Additionally, we make no edge queries while finding a perfect elimination ordering.

Now we apply the idea outlined in the beginning of this section on the elimination ordering. Suppose $v_1, \ldots, v_n$ is a perfect elimination ordering (PEO) for the graph. We obtain a topological sort of vertices of the graph, in the inductive order of the PEO. Let G_i be the graph induced on the first i vertices of the PEO. Suppose that we know all the orientations of the edges in G_i, and we have a topological sort of the orientation of G. Now we insert v_{i+1} using binary search, since the neighborhood of v_{i+1} in G_i formes a total order.

In particular, suppose we have vertices $v_p, v_q \in G_i$ such that $v_p \prec v_{i+1} \prec v_q$, where v_p, v_q are consecutive vertices in the topological order on the neighbors of v_{i+1}. Then we simply insert v_{i+1} after v_p in the topological order. We claim that the resulting order is a valid topological order.

Proposition 5.22. The resulting order from the above procedure is a valid topological order of the directed acyclic orientation of G_{i+1}.

Proof. We only need to worry about edges incident on v_{i+1} as for every other pair of edges, their relative order has not been changed by the insertion and hence the topological order property is satisfied by induction. By Lemma 5.20, v_{i+1} has been inserted properly in the unique topological order on the induced subgraph of v_{i+1} and its neighbors, as they form a clique (becaus of the simplician ordering property). □

Thus we have the following result.

Theorem 5.23 (Biswas et al. [3], 2017)**.** If the comparison graph is chordal, then the undelying poset of the graph can be sorted using $O(n \log n)$ edge queries.

5.4.2 Comparability Graphs

Remember form Section 5.2 that a comparability graph is an undirected graph that admits a transitive orientation. Observe that that the class of comparability graphs, is not related to chordal graphs, i.e., there are chordal graphs that are not comparability and vice versa (see Figure 5.6). In particular, in comparability graphs, we have *no guarantee* about the existence of a simplicial ordering. However, we argue that we can incrementally insert new vertices and get the orientation of its edges by doing a variation of the binary search, outlined in Rédei's Theorem.

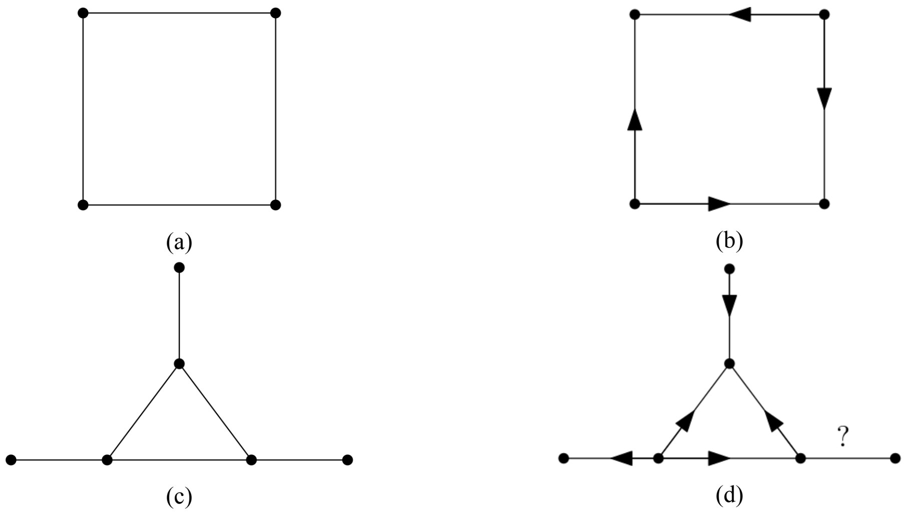


Figure 5.6: In Figure 5.6a we show a comparability, non-chordal graph that admits a valid transitive orientation, as depicted in Figure 5.6b. In Figure 5.6c we show a chordal, non-comparability graph. As shown in Figure 5.6d there is no valid transitive orientation for this graph.

The idea is the same as we did for chordal graphs. However, here we don't have a simplicial ordering. We will simply start with an arbitrary ordering of the vertices and maintain the topological sort of the subgraph induced by the initial set of vertices and incrementally insert the new vertex. Let G_i be the induced subgraph on the first i vertices in our arbitrary order. Suppose that we know all the orientations of the edges in G_i, and we also have a topological sort of the orientation of G_i. Now we insert v_{i+1} using binary search as before. In particular, suppose we have vertices $v_p, v_q \in G_i$ such that $v_p \prec v_{i+1} \prec v_q$, where v_p and v_q are consecutive in the topological order of G_i among the neighbors of v_{i+1}. Then, we simply insert v_{i+1} after v_p in the topological order. We claim the following. Note that, in contrast to the previous case, here we use the term "binary search" somewhat loosely, since we have *no guarantee* that the neighbors of v_{i+1} form a clique, i.e., a total order. Let $\mathcal{L} = \{L_0, \dots L_k\}$ be a layer decomposition. We consider the sequence $v_0, \dots, v_k$ of the neighbors of v_{i+1}, each of the belonging to a single layer, i.e. v_j belongs to layer L_j, and no other vertex of the sequence belongs to L_j (see Figure 5.7). Now, the elements of the sequence form a total order and we can proceed as the previous case. From Lemma 5.20 and following a similar strategy with the proof of Proposition 5.22, we have the following.

Proposition 5.24. The resulting order is a valid topological order of the directed acyclic orientation of G_i.

Thus we have proved the following.

Theorem 5.25 (Biswas et al. [3], 2017)**.** If the comparison graph is a comparability graph, and the oracle function respects a valid comparability orientation, then we can sort the underlying poset using $O(n \log n)$ edge queries.

Note that our proof crucially used the fact that the oracle function answers the queries according to a comparability orientation, and this is not an artifact of the proof.

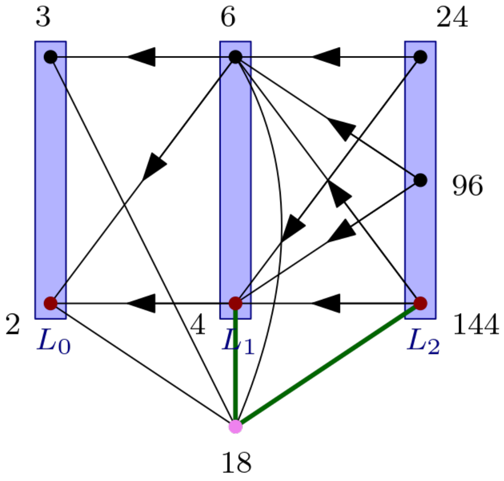


Figure 5.7: Adding the vertex $v = 18$ to our running Example 1.12. Remember that here our partial order is the ”divides” relation ”|”. Hence, 18 will be connected to $3, 2, 6, 4, 144$. We only consider one representative from each layer, i.e. $2, 4, 144$, which form a total order. Doing binary search we only need to query the edges $\{4, 18\}$ and $\{18, 144\}$. We would have $v_p = 4$ and $v_q = 144$. Note, that after the sort, we would end up with a new singleton layer $L' = \{18\}$ between L_1 and L_2.

Otherwise we may to sort using $O(n \log n)$ queries. For example, a complete bipartite graph is a comparability graph, but if we are told that the orientation the oracle uses is a comparability orientation, then *only two orientations are possible* (see Figure 5.8), and we can sort in just one query. However we have shown an $\Omega(n^2)$ lower bound in Lemma 5.18 if the oracle is free to choose any directed acyclic orientation.

5.5 Conclusions

With the discussion on the special cases of chordal and comparison graphs we conclude our discussion of the Forbidden Comparisons Model. In this model we examined primarily the results in [2] and [3]. More precisely, in Section 5.1 we expanded

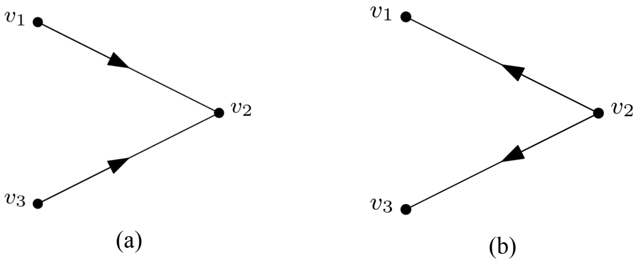


Figure 5.8: The two possible orientations of a comparability bipartite graph. Note that if we had an orientation of the form $(v_1, v_2), (v_2, v_3)$ then we would have the the transitive edge (v_1, v_3). A contradiction, since our graph is a comparability graph and we cannot have odd circles.

	Query Complexity	**Time Complexity**	**Lower Bound**
Sorting [2]	$O((q+n)\log n)$	$O(n^2+q^{\omega/2})$	$\Omega(q+n\log n)$
Sorting [3]	$O((q{+}n)\log(n^2/q))$	$O(n^2+q^{\omega/2})$	$\Omega(q+n\log n)$
Sorting (Chordal Graphs)	$O(n\log n)$	$O(n^\omega)$	$O(n\log n)$
Sorting (Comparability Graphs)	$O(n\log n)$	$O(n^\omega)$	$O(n\log n)$
Random Sort	$\widetilde{O}(n^2\sqrt{q+n}{+}n\sqrt{q})$	$O(n^\omega)$	Open Question

Table 5.1: The Algorithms of Chapter 5 and their respective *query* and *time* complexity. q is the number of missing edges. ω is the exponent of the matrix multiplication. In the forth column, we show the query complexity lower bounds. Note that query lower bounds are also lower bounds on the time complexity.

our presentation on directed graphs. We introduced a special class of directed graphs, the tournaments, while we examined the concepts of topological sort and the orientation of a graph. In Section 5.2 we presented the class of comparability graphs and explored its connection with comparison graphs. The comparison graphs are at the heart of our model. In Section 5.3 we discussed an improvement of the algorithm in [2], presented in [3]. The algorithm presented in [2] achieves a query-complexity of $O((q+n)\log n)$ and a time-complexity of $O(n^2+q^{\omega/2})$, where ω is the exponent of matrix multiplication. On the other hand, in [3] the authors present an algorithm of $O((q+n)\log(n^2/q))$ query-complexity and the same time-complexity. Additionally, in this section we present a lower bound of $\Omega(|E|)$, for $q \geq n^2/4$ and $\Omega(q+n\log n)$ for $q < n^2/4$, on the query-complexity. Lastly, in Section 5.4 we examine some special cases for chordal and comparability graphs. For both of these cases we showed an algorithm with $O(n\log n)$ query and $O(n^\omega)$ time complexity. Another interesting result we left out of our presentation is the randomised analysis in [2]. There the authors present a randomised algorithm with $\widetilde{O}(n^2\sqrt{q+n}+n\sqrt{q})^2$ queries in $O(n^\omega)$ time. In Table 5.1 we summarize the algorithms and lower bounds we examined in this chapter.

The authors in [3] present some interesting avenues for future work. Finding the largest class of comparison graphs that can be sorted in $O(n\log n)$ time remains an open question. Moreover, determining the query complexity when we know that the underlying order relation is a total order, remains an open question. For example, in that case, the complete bipartite graph cannot be oriented, as shown in Lemma 5.18, and hence we do not know of any lower bound other than $\Omega(n\log n)$ regardless of the missing edges. In particular, sorting a complete bipartite graph is the famous *nuts and bolts* problem and can be sorted using $O(n\log n)$ queries [27]. The authors in [3] conjure that sorting in any undirected graph whose vertices constitute a total order, can be done in $O(n\log n)$ queries. In other words, if we know that the directed acyclic orientation of the comparison graph has a directed Hamiltonian path, we conjecture that we can find that path in $O(n\log n)$ queries. An important thing to note is that for total orders, in the forbidden pairs model, we do not even know how to find the k-smallest element in $O(n)$ time, for arbitrary k, while this is possible for $k=1$, or $k=n$.

[2] In the $\widetilde{O}()$ notation we ignore the logarithmic terms.

CHAPTER 6

CONCLUSION

In this thesis we surveyed some results from the literature regarding sorting in the partial order setting. In general, we assumed that we are given access to an oracle function $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \not\sim\}$ that informs us for the relation of two elements of our universe $\mathcal{U}$. We also, assumed that the oracle answers our queries with respect to an underlying, unknown poset $\langle \mathcal{U}, \preceq \rangle$. Our goal was to retrieve all the information of the unknown partial order $\preceq$, by making as few queries to the oracle as possible. In this direction we considered two slightly different models. In Chapters 3 and 4 we discussed the *Width-Based Model*, introduced by Daskalakis et al. in [1]. In this model, we are given access to an oracle $c(\cdot, \cdot)$, along with a parameter w, an upper bound to the underlying poset's width. In Chapter 5 we present the Forbidden Comparisons Model, due to Banerjee and Richards [2]. In this model, our oracle function is defined slightly differently. Namely, we have $c\colon \mathcal{U} \times \mathcal{U} \to \{\preceq, \succ, \perp\}$, where $\perp$ denotes a forbidden comparison. Note, that when $c(a, b) = \perp$, we can know the relation of a and b, or whether they are related at all. We would have to deduce such relations through transitivity. Also, we are given a comparison graph $G = (V, E)$, where $\{u, v\}$ is an edge, if and only if $c(u, v) \neq \perp$. There, our parameter is q, denoting the number of missing edges of G, i.e. $q = \binom{|V|}{2} - |E|$. We organized our presentation as follows.

In Chapter 3 we considered the sorting problem in the Width-Based Model. In Theorem 3.2 we showed an information theoretic lower bound for the sorting problem in the Width-Based Model. In Algorithm 7, we presented the Entropy Sort, a query-optimal algorithm, in the Width-Based Model. Entropy Sort, despite achieving optimality with respect to the number of queries, lack in time efficiency. In particular, the authors in [1] used an elaborate method to estimate the cost of each query, by computing a mass function, described in Section 3.3. This computation assigns large mass to uniformative queries and a small mass to informative queries. We perform a weighted binary search, with respect to the inverse value of the mass function. Unfortunately, in order to compute the mass of each query we consider all the extensions of the currently computed poset, which are exponential in number. Hence, the time-inefficiency of the algorithm. On the other hand, an algorithm that tries to bridge the gap between time and query efficiency is presented in Merge Sort of Algorithm 8. We summarise the results of Chapter 3 in Table 3.1. The open problems in this area are the following. Find a query-optimal algorithm, that achieves polynomial time. The authors suggest the work in [22] as a possible route for improving the time complexity.

In Chapter 4 we discussed the Selection and k-Selection problem, in the Width-Based Model. Note that in general we want to find the k-smallest elements of the poset, following the Definition 1.14. Now, when $k = 1$, our problem would be to find the minimal elements. Our goal is to find the k-smallest elements efficiently, with respect to queries, without sorting the whole poset, if possible. In Section 4.1 we show some upper bounds for the problems at hand, by presenting some deterministic and randomised algorithms. Both the randomised and deterministic algorithms do not deviate much from the philosophy of the constructive Definition 1.14. On the other hand, in Section 4.2 we present some lower bounds for the same problem. The lower bounds are presented in the adversary setting. In this setting we let the queries to the oracle be done by an arbitrary agent, while we traverse on the other side and take the role of an adversary which simulates the query function. The adversary is free to answer the queries as she sees fit, but must be consistent with an underlying poset. The purpose of the adversary is to force the agent to make as many queries as possible. The most sophisticated and general lower bound is presented in Theorem 4.6, regarding the k-Selection problem. We also presented a lower bound for randomised algorithms for the k-Selection problem in Theorem 4.7. We summarized all the results of Chapter 4 in Table 4.1. The open problems in this area are the following. Close the envelop between the upper and lower bounds for the k-selection problems.

Lastly, in Chapter 5 we examined the sorting problem under the Forbidden Comparisons Model. Despite this model had been introduced by [2], we followed the presentation in [3]. Observe that in the Forbidden Comparisons Model we have a piece of information that was unavailable to us in the Width-Based Model, in particular, if $\{a, b\}$ is an edge, we know that a, b are related, without querying the oracle. Moreover, if K_ℓ is a subgraph of the given comparison graph G, we know that the elements in K_ℓ are totally ordered. We used such subgraphs, to sort the whole poset. Note that, essentially the strategy we followed throughout the chapter is as follows. Firstly, we find the orientation of the edges of the comparison graph G; then, we compute the transitive remaining edges. The core result in this chapter is Theorem 5.17, due to Biswas et al., which is an improvement over the related theorem of Banerjee and Richards. Additionally, in that chapter we explored some results regarding special cases, i.e. chordal and comparability graphs, due to Biswas et al. We summarized all these results in Table 5.1. The open problems in this area are the following. Finding the query complexity when we can assume that the comparison graph has a Hamiltonian path. Another open question is to find the query complexity for finding the median, using the latter assumption.

Sorting problems have been at the core of the Computer Science since its conception at the middle of last century. Traditionally, we are given a totally ordered set, with no information about the relations of its elements and we are required to retrieve this information, as fast as possible. In this thesis, we followed a different direction. We assumed a generalised model, where the elements of the given set can be only partially ordered. Our way to determine the elements' relation is restricted to querying an unknown oracle function which we treat as a black box. Additionally, we focused our attention on the number of oracle calls, as a measure of efficiency. In some sense, we ask an information-theoretic question. We are given some partial information (the upper bound w, or the comparison graph G) about a partial order relation; we want to know how much additional information is required to determine the relations of the elements. This problem proved to be much more intricate from its traditional counterpart. We saw that in many cases, query-efficiency requires time-inefficiency and vice versa. In this area we discussed two different models, or variants of this broad topic; the Width-Based Model and the Forbidden Comparisons Model, while we left other equally interesting topics unexamined. For example in [11] Jean Cardinal et al. discuss the problem of Sorting Under Partial Information[1]. In this model, we are given a totally-ordered set and some of the relations of its members, our goal is to retrieve the total order. While this problem shares some similarities with the Forbidden Comparison Model, its key difference is the we are given a directed graph, as a comparison graph, while we also know that the elements are totally ordered. An interesting overview of some earlier work in the poset sorting problem can be found in the Cardinal's and Fiorini's survey paper [10]. Some additional interesting texts, for a more elaborate presentation of the topics we covered in this thesis can be found in Banerjee's Phd dissertation [29] and Jayapaul's Phd dissertation [30].

[1] Another interesting survey text regarding Sorting Under Partial Information is this [28]

BIBLIOGRAPHY

[1] Constantinos Daskalakis, Richard M. Karp, Elchanan Mossel, Samantha J. Riesenfeld, and Elad Verbin. Sorting and selection in posets. *SIAM J. Comput.*, 40(3):597–622, 2011.

[2] Indranil Banerjee and Dana S. Richards. Sorting under forbidden comparisons. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, volume 53 of *LIPIcs*, pages 22:1–22:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[3] Arindam Biswas, Varunkumar Jayapaul, and Venkatesh Raman. Improved bounds for poset sorting in the forbidden-comparison regime. In Daya Ram Gaur and N. S. Narayanaswamy, editors, *Algorithms and Discrete Applied Mathematics - Third International Conference, CALDAM 2017, Sancoale, Goa, India, February 16-18, 2017, Proceedings*, volume 10156 of *Lecture Notes in Computer Science*, pages 50–59. Springer, 2017.

[4] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

[5] Merkouris Papamichail. Introduction to matroid theory. Bachelor's thesis, National and Kapodistrian University of Athens, 2020. `https://pergamos.lib.uoa.gr/uoa/dl/object/2925849`.

[6] L. Mirsky. A dual of Dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971.

[7] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[8] W.T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins Studies in Nineteenth C Architecture Series. Johns Hopkins University Press, 1992.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[10] Jean Cardinal and Samuel Fiorini. On generalized comparison-based sorting problems. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[11] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). *Comb.*, 33(6):655–697, 2013.

[12] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[13] Frank Fussenegger and Harold N. Gabow. A counting approach to lower bounds for selection problems. *J. ACM*, 26(2):227–238, 1979.

[14] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, USA, 1962.

[15] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[16] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2012.

[17] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

[18] Graham R. Brightwell and Sarah Jane Goodall. The number of partial orders of fixed width. *Order*, 20:333–345, 2003.

[19] Ulrich Faigle and György Turán. Sorting and recognition problems for ordered sets. In Kurt Mehlhorn, editor, *STACS 85, 2nd Symposium of Theoretical Aspects of Computer Science, Saarbrücken, Germany, January 3-5, 1985, Proceedings*, volume 182 of *Lecture Notes in Computer Science*, pages 109–118. Springer, 1985.

[20] Jeff Kahn and Michael E. Saks. Balancing poset extensions. *Order*, 1:113–126, 1984.

[21] Graham R. Brightwell, Stefan Felsner, and William T. Trotter. Balancing pairs and the cross product conjecture. *Order*, 12:327–349, 1995.

[22] Martin E. Dyer, Alan M. Frieze, and Ravi Kannan. A random polynomial time algorithm for approximating the volume of convex bodies. In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*, pages 375–381. ACM, 1989.

[23] Yangjun Chen. Decomposing dags into disjoint chains. In Roland R. Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 of *Lecture Notes in Computer Science*, pages 243–253. Springer, 2007.

[24] David Kelly. *Comparability Graphs*, pages 3–40. Springer Netherlands, Dordrecht, 1985.

[25] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[26] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[27] János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in o(n log n) time. *SIAM J. Discret. Math.*, 11(3):347–372, 1998.

[28] Preetum Nakkiran and Matthew Francis-Landau. Graph entropy , posets , and their applications in sorting under partial information, 2015. `https://preetum.nakkiran.org/pdf/entropy_posets_supi.pdf`.

[29] Indranil Banerjee. *Problems on Sorting, Sets and Graphs*. PhD thesis, George Mason University, 2018. `https://www.cct.lsu.edu/~ibanerjee/files/Publications/PhD_Dissertation.pdf`.

[30] Varunkumar Jayapaul. *Sorting and Selection in Restriction Model*. PhD thesis, Chennai Mathematical Institute, 2017. `https://libarchive.cmi.ac.in/theses/varunkumarj_cs2017.pdf`.

INDEX

k-Selection Problem, 13

adversarial lower bound, 47
adversarial setting, 52
algorithm
    median of medians, 13
antichain, 6
    maximal, 6
arc
    backward, 18
    forward, 18
arcs, 5
augmenting path, 19

chain, 6
    maximal, 6
ChainMerge, 30
circle, 5
circuit, 5
clique, 5
cut, 20
    problem, 20

DAG, 58
diagram
    Hasse, 7

edges, 5
element
    k-smallest, 8
    height, 8
    maximal, 6
    maximum, 6
    minimal, 6
    minimum, 6

flow, 18
    network, 17
    problem, 18
    value, 18

graph, 5
    bipartite, 21
    chordal, 69
    comparability, 61
    comparison, 57
    complementary, 5
    directed, 5
    directed acyclic, 58
    induced, 5
    isomorphic, 5
    orientation, 58
    sub-, 5
    tournament, 58
    transitive orientable, 61
    transitive orientation, 58

Handshake Lemma, 5

layer decomposition, 60
linear extension, 61

mass function, 35
matching, 22
    perfect, 22
merge sort, 12

network

residual, 18
node
    degree, 5
    in-degree, 5
    in-neighborhood, 6
    neighborhood, 5
    out-degree, 5
    out-neighborhood, 6
    simplicial, 69
nodes, 5

oracle function, 11, 12

partial order, 4
partially ordered set, 6
path, 5
    Hamiltonian, 59
PEO, 69
perfect elimination ordering, 69
poset, 6
    conditioned, 35
    extension, 35
    height, 7
    width, 7

query complexity, 12

relation, 3
    antisymmetric, 3
    binary, 3
    enforced, 35
    equivalence, 4
    prohibited, 35
    reflexive, 3
    restriction, 8
    symmetric, 3
    transitive, 4

Selection Problem, 13
set
    independent, 22
Sorting Problem, 12
subposet, 8

topological order, 58
total order, 4
transitive closure, 4
    operator, 4
transitive orientation, 61

vertex, 5
    degree, 5
    neighborhood, 5
vertex cover, 22

weighted binary search, 35