PROBLEMS ON SORTING, SETS AND GRAPHS

by

Avah Banerjee
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Problems On Sorting, Sets and Graphs

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Avah Banerjee
Master of Science
George Mason University, 2015
Bachelor of Technology
National Institute Of Technology, Durgapur, India, 2009

Director: Dr. Dana Richards, Professor
Department of Computer Science

Summer 2018
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my mother.

Acknowledgments

This thesis would not have been made possible without my advisor Dana Richards' mentorship and encouragement. I consider myself very fortunate to have had him as my advisor. I would like to acknowledge the members of my committee Walter Morris, Fei Li and Zoran Duric. They help me improve the presentation of this thesis as well as verifying some of the technical results. I would like to thank the anonymous reviewers for their insightful critiques of the papers arising out of this thesis. In particular I would like to thank Igor Shinkar for some of the results on sorting numbers. I would like to thank my previous advisor Kenneth De Jong for giving me the opportunity to explore different research paths during the early years of my PhD.

Last but certainly not least I would like to thank my family for their continued support over the course of all these years.

Table of Contents

Page

List of Tables viii

List of Figures ix

Abstract xiii

I Part I: Problems On Order **1**

1 Comparison Models and Problems 2

1.1 Introduction 2

1.2 Basic Definitions 3

1.2.1 Posets 4

1.2.2 Graphs 5

1.3 Abstract Comparison Model (ACS) 6

1.3.1 Methods for Lower Bounds 7

1.4 Machine Model 9

1.4.1 Pointer Machines and CBPs 11

1.5 CBPs with Additional Inputs 11

1.5.1 Non-uniform Comparison Costs 12

2 Restricted Comparison Model 13

2.1 Restricted Cost Model and Sorting 15

2.2 Background 17

2.3 Preliminaries 19

2.4 A Deterministic Algorithm When $q = O(n)$ 21

2.4.1 Basic Idea 21

2.4.2 A Restricted Case 22

2.4.3 Initial Sorting 22

2.4.4 Partition Step 23

2.4.5 Merge Step 26

3 Restricted Sorting Continued 28

3.1 Constructing the Set K 28

3.1.1 Computing An Approximate Median Of X 29
3.1.2 A Divide-and-Conquer Approach 31
3.1.3 Merge Step 32
3.1.4 Comparison Complexity 32
4 Restricted Comparison Model Under Randomization 35
4.1 Restricted Sorting With Randomization 36
4.2 A Randomized Algorithm 37
4.3 When G Is A Random Graph 40
5 Set Maxima 41
5.0.1 Background 41
5.1 Previous and Related Work 43
5.1.1 Our Result 47
5.2 A Generic Formulation 47
5.3 Convex Set-System 49
6 Concluding Remarks 54
6.1 Restricted Sorting 54
6.2 Set-Maxima 56
6.2.1 Local Sorting 57

II Part II: Problems On Graph Reconfigurations 59

7 Reconfiguration Problems 60
7.1 General Framework 60
7.2 Examples of Reconfiguration Problems 61
7.2.1 Permutation Routing 62
7.2.2 Acquaintance Time 62
7.2.3 Sorting Permutations 63
7.2.4 Visiting Time 64
7.3 Sequential Model 65
8 Hardness Of Permutation Routing 67
8.1 Prior Results 68
8.2 Computational Results 70
8.3 A $O(n^{2.5})$ Time Algorithm for Deciding when $rt(G,\pi) \leq 2$ 71
8.4 Determining $rt(G,\pi) \leq k$ is Hard for Any $k \geq 3$ 73
8.5 Connected Colored Partition Problem (`CCPP`) 77

8.6 Routing As Best You Can 80
9 Structural Results On Permutation Routing 83
9.1 Graph Connectivity 83
9.2 Structural Results 85
9.2.1 An Upper Bound For h-connected Graphs 86
9.2.2 Relation Between Clique Number and Routing Number 90
10 Sorting Permutations and Sorting Number 91
10.1 Some Additional Results On Routing 95
10.1.1 Routing on subgraphs of G 96
10.2 General Upper Bounds on $\mathfrak{st}(G)$ 98
10.3 Bounds on Concrete Graph Families 102
11 Sorting Network On Trees 106
12 Sorting Network On A Pyramid 115
Bibliography 120

List of Tables

Table Page

10.1 Known bounds on the sorting numbers of various graphs 94

List of Figures

Figure Page

1.1 An example of a poset with 5 elements, $X = \{a, b, c, d, e\}$. Here $R = \{(a, b), (a, e), (a, d), (e, d), (c, d), (a, a), (b, b), (c, c), (d, d), (e, e)\}$. Longest chain (a, e, d) and anti-chain (b, e, c). 4

1.2 Set of partial orders in $\mathscr{T}$ for $n = 4$ when $\mathtt{P}$ is the problem of finding the maximum. 6

2.1 Visualizing the steps. At the bottom of T the shaded boxes represents the F_1-nodes and the blue rectangles the F_2-nodes. The outer dashed triangle represents the full tree $\hat{T}$. The tree $\hat{T}$ is created during the partitioning step and in the merge step we start from the deepest leaves of $\hat{T}$ and move upwards. 25

5.1 Polygonal chains. 51

5.2 A case where P and Q are not covers of R. 51

6.1 For the Peterson's graph (left) there are 10 different Hamiltonian paths (20 total orders). However the starfish graph has none, as it is not Hamiltonian. 56

7.1 The stars of stars has the highest value of acquaintance time $O(n^{3/2})$. The outer and inner stars all have degree $\theta(\sqrt{n})$. We can acquaint a pebble with others by placing it on the center of various stars. 63

7.2 Routing the permutation $\pi = (12345678)$ on a Cube. 66

8.1 A star on the left and a Caterpillar to the right. A tree has γ-type star decomposition if every subtree has $\leq \gamma n$ vertices. In a (α, β)-caterpillar decomposition, every T_j connected to one of the end vertices has size at most βn and sum total of all vertices in the trees connected to the central spine (like T_i) is $\leq \alpha n$. Every tree has a $\frac{1}{3}$-star or a $(\frac{1}{3}, \frac{1}{3})$-caterpillar decomposition. 69

8.2 The two cycles are shown as concentric circles. The direction of rotation for the outer circle is clockwise and the inner circle is counter-clockwise. Once, we choose $(\pi_{i,s}, \pi_{j,t})$ as the first matched pair, the rest of the matching is forced. Solid arrows indicate matched vertices during the first round. Note that if the cycles are unequal then the crossed vertices in the figure will not be routed. 72

8.3 Atomic Gadgets, pairs (a, b) need to swap their pebbles. The unmarked red circles have pebbles that are fixed. 74

8.4 Variable graph of X. (a) is a special case for $m_X = 2$, (b) is the general case. 75

8.5 The graph G_ϕ. 77

8.6 An example of a `CCPP` instance. 78

8.7 A modified clause gadget (from the proof of Theorem 7.1) for the clause $C = x \vee y \vee \neg z$. 79

9.1 A situation where C is a cutset of vertices and sets A, B partitions $V \setminus C$. . 85

9.2 Although there are 3 edge disjoint paths between A and B all this paths share the vertex u hence can only be activated (matched) one at a time. . . 85

9.3 G is decomposed into 4 connected blocks, which are connected to each other via G_h. 88

9.4 The clique H has been contracted into a super-vertex v. 89

10.1 Figure (b) corresponds to a sorting network (not necessarily of optimal depth) for the tree given in (a). Here π is the identity permutation. Every comparator, given by vertical segments (all directed upwards) joining two wires, always conform to the edges of the tree. 94

10.2 In the tree above, rooted at vertex labeled 1, the dotted line traces a contour for the tree. The contour represented by a path consists of the following sequence of vertices from T and their duplicates (indicated by crosses): $(1_1, a_1, 6_1, 2_1, 7_1, 2_2, 8_1, 2_3, a_2, 3_1, 9_1, a_3, a_4, 1_2, 10_1, 4_1, 1_3, 5_1, 11_1, 5_2, 12_1, a_5, a_6)$, Where i_j is the j^{th} copy of vertex labeled i and a_k's are additional vertices indicated by small red dots. 102

10.3 The Cartesian product graph $G = G_1 \Box G_2$. The rows highlighted by blue regions represents copies of G_2. 105

11.1 A balanced decomposition of a tree. 107

11.2 Pair of subtrees joined at the root r. The e/o labels indicate parity. 109

12.1 A pyramid $\triangle_{3,2}$ in 3-dimension 115
12.2 The graph $\triangle'_{3,2}$ after stripping way edges from $\triangle_{3,2}$ 116

Notations

- Following general guidelines are used regarding mathematical notations with few exceptions.
- All variables, symbols are defined using lower case letters. For example a vertex u of some graph.
- Sets and structures are written using upper case capital letters. For example, a graph G.
- As a general rule, any set or tuple whose elements are themselves structures are written using the $\mathfrak{Fraktur}$ font. For example, $\mathfrak{P}(S, R)$ is a partial order, with element set S and relation set R.
- As a general rule, constants are defined using letters from the beginning of the English alphabet ($a, b, c, \ldots$ etc.).
- As a general rule, we use lowercase Greek letters for parameters ($\alpha, \beta, \ldots$ etc.).
- Problem names are written using the `typewriter` font.
- As a general rule, functions whose domains are not over integers or reals, have names that are written using the $\mathcal{Calligraphic}$ font.
- Algorithms or methods are written using Sans Serif font.
- 2^S is the power-set of S.
- $S^{\{2\}}$ is the unordered pairs from S and $S^{(2)}$ denotes the set of all ordered pairs of S.
- All logarithms are of base 2 unless specified otherwise.
- We use $\widetilde{O}(f(n))$ to denote $O(f(n) \log^c f(n))$ for some constant c.
- $\omega \in [2, 2.38]$ is the exponent in the complexity of matrix multiplication.
- We use $|X|$ to denote the cardinality of X.
- The notation $[n]$ used to denote the set $\{1, 2, \ldots, n\}$.

Abstract

PROBLEMS ON SORTING, SETS AND GRAPHS

Avah Banerjee, PhD

George Mason University, 2018

Dissertation Director: Dr. Dana Richards

This dissertation sets out to explore the complexity of some fundamental combinatorial problems in both deterministic and randomized settings. We divide the work into two parts: 1) Problems on order and 2) Problems on graph reconfiguration. In the former setting, we first study generalizations of the standard sorting problem followed by a more abstract problem known as the set-maxima problem. Additionally we look at some special cases of these problems. In the second part we study some specific versions of the broadly defined graph reconfiguration problem. Using a matching model we give both structural and computational complexity results for routing and sorting on graphs. Some of these results have already been extended by other authors and several interesting problems have been discovered that remains open. We believe these would be of interest to a broader audience in the future.

The first part of the dissertation focuses on a structured cost model. This is motivated by the fact that the uniform cost model does not provide an accurate estimate of the runtime of an algorithm in many real world scenarios. However, an arbitrary cost model may not be that useful. It has been proven that even for very simple problems it is not possible to find a cost-optimal algorithm if the comparison costs are arbitrarily chosen by an adversary. With this in mind we look at a restricted cost model; we assume that comparison costs are either 1 or ∞. In this setting, given two objects from our universe we can either compare them and determine their order, at a cost of one unit, or the pair is incomparable (comparing them costs ∞) and cannot be compared directly. In our problem description, along with the set of elements, we are also given an undirected graph. There is an edge between two elements if and only if the pair can be compared. We also look at the case when this graph is a random graph. In both of these settings we have made progress with regard to the problem of sorting when some pairs of elements are incomparable. For example, in the structured cost model we give the first nontrivial deterministic algorithm for the problem, which is also optimal for certain special cases. Another comparison-based problem is the well known set-maxima problem, which is to determine the maximum element of every subset from a collection of subsets of a given set. The general case has remained unsolved for about 40 years. Some special cases have been considered in the past. Following this tradition we give an optimal deterministic algorithm based on a geometric formulation of this problem.

In the second part of this work, we focus our attention to graph reconfiguration. We propose a unifying model for graph reconfiguration and discuss several known problems as special cases of this model. Notably, we study the permutation routing and the oblivious sorting problem on graphs. For these problems we give several complexity and structural results. These include proving hardness results of routing and some of its variants. We give bounds on various routing and sorting parameters, for different family of graphs. Additionally, we also give explicit sorting networks for trees and the pyramid graph, which are the first of their kind.

Part I

Part I: Problems On Order

Chapter 1: Comparison Models and Problems

1.1 Introduction

Informally, an order is a ranking on a set of objects. Many natural problems are solved via comparisons between objects whose outcomes are determined by an ordering. Indeed most human decisions are evidently the result of choosing one outcome against another. So it is of no surprise that many archetypal problems in Computer Science are problems that can be solved by means of comparisons. We begin with one of the most basic problems: given a set of numbers find their maximum. Informally, we want to determine:

1. How many comparisons are necessary between the elements to decide that a chosen element is a maximum?

2. Can we *find* a maximum element with the necessary number of comparisons?

The first one is a *lower-bound problem* and the latter is an *upper-bound* problem. If we are looking at a set of n elements, then we know the answer to both of these question, when the problem is to find a maximum element. In fact both these bounds are exactly $n-1$ comparisons. Even though all these facts seems trivial, even for the simple case of finding the maximum, things can be subtle. For example, why do we always need $n-1$ comparisons? We shall discuss this soon.

Sorting is another natural problem that can be solved using comparisons. Any adjective to describe the importance of sorting will not be adequate. We know that at least $\lceil \log n! \rceil$ comparisons are necessary. However, we do not yet know how to sort with exactly this many comparisons, except when the set of objects is small (< 20) [1]. Fortunately, for most practical situations, the cost of a comparison (say comparing numbers on a computer) is cheap. So as long as we do not do too many extra comparisons, it is good enough. What if comparisons are expensive? For example, suppose you are in the real-estate business. You have 10 houses to sell. You want to determine their price based on their popularity. However, you can directly compare a pair of houses only if you have the same client go to both of them. Clearly, this is a case where comparisons are very expensive and you would want to make as few as you can.

1.2 Basic Definitions

We first define some basic terms related to order on finite sets and graphs. They are needed for our description of comparison-based models.

Let X be a set of n elements. A pair is *comparable* if we know their relative values. Otherwise they are said to be *incomparable*. A comparison operator takes a pair $\{u, v\}$ of elements and returns an ordered pair (u, v) or (v, u) based on whether $u \geq v$ or $u < v$ respectively if the pair is comparable. If the pair is incomparable the comparator returns $u \perp v$. We can think of a comparison operation as an operator on pairs of vertices of a graph G with vertex set X. Each comparison adds a directed edge from the larger element (lower rank) to the smaller (higher rank) one, if the two elements can be compared. (Edges that can be inferred by transitivity are assumed; see below) We say X can be totally ordered if every pair of elements are comparable. Formally:

Definition 1.1. (Total-Order) If there is a bijection $\sigma : X \to [n]$ such that for every pair $x, y \in X$, $x \geq y$ iff $\sigma(x) \geq \sigma(y)$.

We call σ the *underlying order* of X. We say X is partially ordered if there exists a *injective* σ is. Every total order is also a partial order. A partial order on X is specified in terms of a relation set $R \subset X \times X$. We use $\mathfrak{P}(X, R)$ to denote the *partially ordered set*, *poset*, of X. We simply use $\mathfrak{P}$ instead, whenever the intended meaning is clear. The relations in R must satisfy the following properties: 1) reflexivity, 2)anti-symmetry and 3) transitivity:

(i) The pair $(u, u) \in R$, $\forall u \in X$ (reflexivity)

(ii) If (u,v) and (v,u) are both in R then $u=v$. (Anti-symmetry)

(iii) If $(u,v),(v,w) \in R$ then (u,w) in R. (Transitivity)

1.2.1 Posets

Definition 1.2. (Anti-chain) A set of mutually incomparable elements is called an *anti-chain* of $\mathfrak{P}$. The size of the largest anti-chain is called the *width* of the poset.

Definition 1.3. (Chain) A set of mutually comparable elements are called a *chain* of $\mathfrak{P}$. The size of the longest chain is called the *height* of the poset.

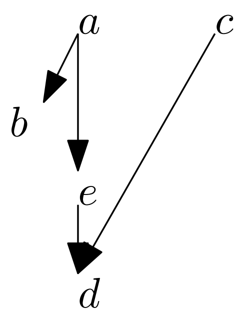


Figure 1.1: An example of a poset with 5 elements, $X=\{a,b,c,d,e\}$. Here $R=\{(a,b),(a,e),(a,d),(e,d),(c,d),(a,a),(b,b),(c,c),(d,d),(e,e)\}$. Longest chain (a,e,d) and anti-chain (b,e,c).

Definition 1.4. (Transitive-closure) Given an anti-symmetric relation set R of X the *transitive closure* of R is the relation set,

$$\mathcal{TC}(R)=\{(u,v)\in X\times X \mid \exists\, w_1,w_2,\ldots,w_k \in X \text{ and } (u,w_1),(w_1,w_2),\ldots,(w_k,v)\in R\} \bigcup R$$

If $(u,v)\in R$ we say u *dominates* v. Additionally, if there does not exist any $w \neq u,v$ such that $(w,v),(u,w)\in R$ then u is also a *cover* of v. The set of covers of u, denoted by $C(u)$, is the *cover set* of u. If $C(u)$ is empty then u is a *maximal element.* On the other hand if u is not in the cover set of some element then u is a *minimal element.*

Definition 1.5. (Linear-extension) A linear extension of $\mathfrak{P}$ is a bijection $\sigma : X \to [n]$ of X such that if $(u, v) \in R$ then $\sigma(u) \geq \sigma(v)$.

We use $\mathscr{E}(\mathfrak{P})$ to denote the set of all linear extension of $\mathfrak{P}$ and $e(\mathfrak{P})$ denote the size of this set. For a pair u, v let $E(u, v)$ and $e(u, v)$ be the set and number of linear extensions respectively in which $u \geq v$.

Definition 1.6. (Balancing-pair) A pair u, v is called δ-balancing if

$$\frac{\min(e(u,v), e(v,u))}{\max(e(u,v), e(v,u))} \geq \delta$$

for some constant $\delta > 0$.

The best known value of δ of an arbitrary partial order is 3/11 from the celebrated results of Kahn and Saks[2]. However it is conjectured that $\delta = 1/3$. We know that δ cannot be less than 1/3. This follows from the fact that the poset with three elements and only one relation has 3 linear extensions and the remaining comparisons can only reduce the number of linear extensions by at most 1/3.

1.2.2 Graphs

This section reviews some useful notions about graphs. Unless otherwise stated all graphs should be assumed simple. Let $G(V, E)$ be such a graph, where V is its set of vertices and E the set of edges. A graph is called *directed* if some of its edges have directions. A *partial orientation* of a graph is a partial coloring of the edges with two colors. The colors l, r indicates the direction of an edge and an uncolored edge remains unoriented. Note that we deviate from the more common definition by allowing some edges to remain unoriented. An *orientation* is a special case where all edges have a direction. Further, a partial orientation is *acyclic* if it does not induce any monochromatic cycles in G.

We use $d(v)$ to denote the degree (number of edges) of any vertex and $d_+(v)$ and $d_-(v)$ to denote the out-degree and in-degree respectively. The *transitive closure* of a directed graph is another graph where we add a directed edge between every pair of vertices (u, v) from u to v if in the original graph there was a directed path from u to v.

1.3 Abstract Comparison Model (ACS)

In this section we introduce the comparison model we will use throughout the first Part. Our input is the set X. Associated with X is a set of permissible partial orders $\mathbb{Q}$ that the elements of X can form. That is, the unknown (partial) ordering of X from which we compute relations comes from this set $\mathbb{Q}$. Unlike X the set $\mathbb{Q}$ is not given to us explicitly as an input rather implied from the definition of the problem. For example in the sorting problem $\mathbb{Q}$ consists of all $n!$ total orderings of X. Comparison-based-problems (CBPs) can be thought of as extracting certain relational information about pairs in X such that the set of relations satisfy some partial order from a set a target orders $\mathcal{T}$. Formally, a CBP is a function $\mathtt{P} : \mathbb{Q} \to \mathcal{T}$. Like the set $\mathbb{Q}$, the target set $\mathcal{T}$ is property of the problem $\mathtt{P}$ and not explicitly supplied as the input. The notation $\mathtt{P}_n(X)$ denotes the problem $\mathtt{P}$ which takes the set X as input and the size of its input is n. We use the subscript n to indicate the output is given in terms of the input size. When the meaning is clear we shall leave out the input or the size or both.

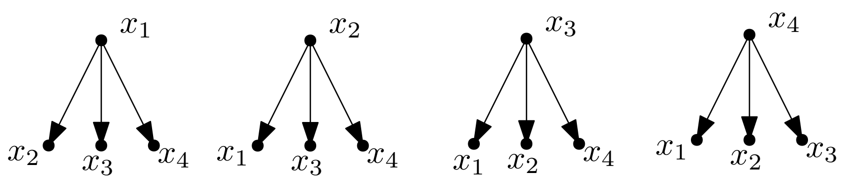


Figure 1.2: Set of partial orders in $\mathcal{T}$ for $n = 4$ when $\mathtt{P}$ is the problem of finding the maximum.

Let $\mathtt{FindMax}_n$ be the problem of finding the maximum element from a set X of n elements. In the Figure 1.2 the posets in the target set are shown for $n = 4$. The set $\mathbb{Q}$ in this case is all possible orderings of X. The target consists of exactly those posets which can be *inferred* from the solution of the problem without computing additional relations. For the case of $\texttt{FindMax}_n$ the output or the solution is an element with the maximum value. Given only this information we can only determine the set of relations where the maximum element dominates (or equal to) the other elements of the set.

A comparison-based algorithm (CBA) solves $\mathtt{P}_n(X)$ if for every ordering in $\mathbb{Q}$ it outputs a relation set R_X (function of the input X) such that its transitive closure $\mathscr{TC}(R)$ is *consistent* with some poset in the collection $\mathscr{T}$. A relation set R is consistent if there is a poset in $\mathscr{T}$ whose relation set is a subset of $\mathscr{TC}(R)$. An optimal comparison-based-algorithm is one which uses the minimum number of comparisons to solve $\mathtt{P}_n$. The number of comparisons necessary for an optimal algorithm to solve $\mathtt{P}_n$ is the *comparison complexity* of $\mathscr{CC}(\mathtt{P}_n)$ of $\mathtt{P}_n$.

Let us revisit the problem of finding the maximum. We mentioned that it requires at least $n-1$ comparisons and we know that we can use $n-1$ comparisons to find the maximum. So we have $\mathscr{CC}(\texttt{FindMax}_n) = n-1$. There are very few natural CBPs where we have an exact result like this.

1.3.1 Methods for Lower Bounds

There are mainly two prevailing methods of determining a lower bound for the comparison complexity. One is using a "comparison tree" (also known as the information theoretic bound) and the other is an adversarial method. In this section we discuss these methods.

Comparison Trees

A deterministic CBA is a function $\mathscr{A} : 2^{X^{(2)}} \to X^{\{2\}}$. Here, $2^{X^{(2)}}$ is the power set of the set of all ordered pairs from X and $X^{\{2\}}$ is set of all pairs from X. Given an input instance and a sequence of comparisons, the algorithm chooses the next pair to compare and keeps going until $\mathscr{TC}(R)$ is consistent with $\mathscr{T}$. These choices could be thought of as descending a binary tree from the root, where a node represents a comparison and the result of the comparison decides the child to descend to. The leaves correspond to the output of the algorithm (from the resulting set of relations $\mathscr{TC}(R)$). Thus computation for a specific input is a path on this tree and the length of this path is the number of comparisons made. Since an algorithm must give the correct output, for each input, the number of nodes of this tree will be at least the number of different possible outputs. The depth (longest path from root to a leaf) gives the minimum number of comparisons necessary to solve the problem for any comparison based algorithm. We can lower bound this depth by noting that it is minimum if the tree is perfectly balanced and in that case the depth is $\log|\text{number of leaves}|$.

The classic example is the sorting problem. For a given input (the set X) there are $n!$ different possible orderings. Hence, the minimum depth of any comparison tree for sorting is $\geq \lceil \log n! \rceil$. As it turns out we also have comparison based algorithm for sorting which are only off by some constant factor. So we can say that, $\lceil \log n! \rceil \leq \mathscr{CC}(\texttt{Sorting}_n) \leq c \log n!$.

However, this method fails to give correct lower bounds for CBPs where the number of output types are relatively small. For example, there are only n possible answer to which element is the maximum of X. This gives a lower bound of $\lceil \log n \rceil$. However this bound is loose (too small) which we will show to be $n-1$ using the adversarial model.

Adversarial Model

In this model we assume to have two agents, Alice and Bob. Alice is our algorithm which solves some CBP. Bob picks the input for which Alice has to solve the problem. Further, Bob knows in advance the sequence of comparisons that Alice will perform given that input. Bob, being an adversary can choose an input that will cause Alice to fail if she does not use sufficiently many comparisons. We want to determine the minimum number of comparisons needed by Alice so that not matter what input is chosen by Bob, Alice always produces the correct output. The common strategy for computing a lower bound using this model is as follows: Suppose we want to show that $\mathscr{CC}(\texttt{P}_n) \geq f(n)$, for some function $f(n)$ over the integers. Then we assume Alice uses at most $f(n)-1$ comparisons and show that there is an input X for which Alice produces an incorrect answer.

We saw earlier that the comparison tree model is too weak for finding the maximum. However we can easily get the optimal bound using the adversarial model. Suppose $\mathscr{C}\mathscr{C}(\texttt{FindMax}_n) < n-1$. Suppose we perform less than $n-1$ comparisons and let u be the maximum computed by Alice. Since there are n elements, there must have been some element v that was not compared to u. Bob can make v the maximum element.

Although this model is quite powerful, it is not always easy to guess the function $f(n)$ and then find the *obstructing* input.

1.4 Machine Model

For comparison based algorithms we are primarily interested in counting the number of comparisons. This leads us to use a machine model that conceptually separates comparisons from other operations like memory access. The model we will use is known as *Pointer Machines.* First we give some background about them. The origin of pointer machines goes all the way back to the middle of twentieth century. These early models, generally describes a pointer machine as a graph where computation is understood as a graph traversal. We are also allowed to modify the structure of the graph by adding directed edges, adding or deleting nodes etc. Typical examples are Knuth's *linking automaton* and Schonhage's *storage modification machine*[3]. Due to its age the literature on pointer machine is vast. However, we only focus on a specific class that is relevant for comparison-based problems. Specifically, we look at the pointer machine model introduced by Tarjan[4].

Pointer machines have an expandable memory and a finite set of *registers.* A register can be of two types: pointer type and data type. A record is a finite collection of fields. Each field can be of two types : data field and pointer field. All records have the same set of fields. A pointer specifies (points to) a particular record or is *null.* We call a field or a register an *element.* An element is atomic. An element either stores a single pointer or an unit of data depending on its type. Machine operations consist of manipulating these pointers, we discuss them next.

Pointer Machine Operations

The notations used here are mostly taken from [4]. Let r denote a pointer register, use s to denote a data register and t to denote a register of either type. We use $*r$ to denote the record pointed by r and $*r.f$ to denote the field labled f of that record. We use $\leftarrow$ to denote assignments between elements of same types. All data write operations are destructive.

1. $r \leftarrow \emptyset$: (place a null pointer in register r)

2. $t_1 \leftarrow t_2$: (copy from t_2 and overwrite it in t_1)

3. $t \leftarrow *r.f$: (copy contents of field f from the record $*r$ to register t)

4. $*r.f \leftarrow t$: (copy from register t to the field f of the record $*r$)

5. $s_3 \leftarrow \texttt{op}(s_1, s_2)$: (apply operation `op` on data registers s_1 and s_2 and put the result in s_3)

6. $\texttt{create}(r)$: (create a new record $*r$ not pointed to by any existing pointers and make r point to it)

7. `halt`: Stop execution.

8. `if` *condition* `then go to` i

9. `true`: Always true condition.

10. $t_1 = t_2$: `true` iff t_1 and t_2 are same

11. $\texttt{p}(s_1, s_2)$: `true` iff s_1 and s_2 satisfy the predicate $\texttt{p}(,)$.

The main limitation of pointer machines are that they are not allowed to perform arithmetic operations on pointer types. That is, operations like,

$$t_3 \leftarrow \texttt{op}(t_1, t_2)$$

where t_1, t_2, t_3 can be of different register types, are not allowed. Further, in the setting of comparison-based algorithms, we restrict operations on data elements to only comparisons. These limitations makes it a less powerful machine model than the more commonly used *Random Access Machines* (RAMs), where we do not have such limitations as the name suggests. However, this also makes RAMs harder to study in terms of determining lower bounds for problems.

1.4.1 Pointer Machines and CBPs

For CBPs the only type of predicate we use is a comparison between elements (data registers or data fields). We do not have operations like $\mathtt{op}(s_1, s_2)$ on the data elements. Thus the complexity of CBPs on pointer machines can be decomposed into two parts: 1) number of comparison predicates, 2) all other operations such as assignment, create, equality testing, transitive closure etc., during the execution of the machine. Transitive closures can be computed by traversing from one element to another using a sequence of pointers where we work with an adjacency list representation of a graph. Further, we can assume all these operations take some constant time per operation. We call them *memory-type* operations. So the total complexity is the sum of these two parts, which for a CBP $\mathtt{P}_n$ is denoted by $T(\mathtt{P}_n)$. Obviously, comparison complexity $\mathscr{C}\mathscr{C}(\mathtt{P}_n) \leq T(\mathtt{P}_n)$. For many CBPs the number of memory-type operations per comparison is generally bounded by a small constant (independent of the size of the problem). In this case it is reasonable to upper-bound $T(\mathtt{P}_n)$ by $\mathscr{C}\mathscr{C}(\mathtt{P}_n)$.

1.5 CBPs with Additional Inputs

There are many natural CBPs where the relation set R we need to compute on X depends on some additional structures. For example, consider the *minimum spanning forest* problem ($\mathtt{MSF}_{n,m}$). A vertex disjoint set of trees is known as a forest. If the forest contains all the vertices of G then it is also a spanning forest. Given an edge-weighted undirected graph G

the task is to find a spanning tree for each connected component of G with the minimum total edge weight. Here, the set X is the set of edges of G. For this problem $\mathbb{Q}$ is again the set of all possible total ordering of X. Solving MSF requires one to compute a partial order on X, although the precise properties of this poset remain unknown. However, the structure of G is needed to filter this poset to get the required output. In Chapter 5 we discuss one such problem called Set-Maxima.

1.5.1 Non-uniform Comparison Costs

One generalization of the comparison paradigm we discussed so far is to make the cost associated with comparing a pair to be a function of the pair. One such model is the monotone cost model [5], where the cost of comparing depends on the rank of the pair in the total ordering of X. An extreme case is when the cost is either 1 or ∞. This can be specified as a graph G (the additional input) with vertex set X. A pair of elements that are adjacent in G can be compared, whereas non-adjacent pairs are too cost-prohibitive to be compared. This will be our main model in Chapters 2-4.

Chapter 2: Restricted Comparison Model

We can extend the basic comparison model where comparison costs always are the same to a one where the costs depends on which pair is being compared. Let us motivate this with an example similar to what was provided in [6]. Suppose we want to rank different companies based on certain financial data. However, not all companies have these data available for free. Comparing financial prospect (say which has a better short-term outlook with respect to stock price) of two companies requires one to purchase these internal data. However, for different companies the cost of accruing the data may very. Hence we have non-uniform costs of comparisons.

So for this setting we have two inputs. One is the set X as before. Another is a cost (of comparisons) function $\mathscr{C} : X^{\{2\}} \to \mathbb{R}$. These inputs can be represented as a edge-weighted graph G, with vertex set X and weights on an edge between a pair of vertices representing the cost of comparing them. Henceforth we shall assume the input for this non-uniform comparison cost model is the pair (X, G).

Competitive Ratio: In the non-uniform cost setting complexity of a problem is not simply the number of comparisons required to solve it. For example, let us go back to our running example $\texttt{FindMax}_n$. Suppose the costs are arbitrary. We know that $n-1$ comparisons are required. However, many different choices exists for choosing this set of $n-1$ comparisons. Ideally we want to choose the set of comparisons that minimizes the total cost. However, without knowing the maximum this may seem like a difficult task if the costs are arbitrary. This leads to a notion of competitive ratio. Suppose omniscient observers have seen the maximum, however they have to convince you that the element is indeed the maximum. For this they can decide to give the results of a collection of comparisons that convinces you that their assertion is correct. This set of comparisons serves as a proof and its cost is the total of all the comparison costs. A proof with the minimum cost is an optimal proof. The competitive ratio of an algorithm is computed based on this cost (numerator) and the cost associated with using the algorithm (denominator). Formally, let C_A and C^* be the set of comparisons made by an algorithm `A` and an optimal algorithm respectively. Then the competetive ratio of `A` is

$$= \frac{\sum_{(u,v)\in C_A} \mathscr{C}(u,v)}{\sum_{(u,v)\in C^*} \mathscr{C}(u,v)}$$

.

Arbitrary non-uniform cost models can make trivial problems become non-trivial, like finding the minimum [6, 7]. It is known that the lower bound for the competitive ratio in this case is $n-2$. By contrast, in the uniform cost setting the competitive ratio is 1, since any set of $n-1$ comparisons cost the same. There is an $(n-1)$-competitive algorithm for this problem. The basic idea is to maintain a set of possible maximums T. As long as the set has more than one element find the cheapest comparison, not yet been made, between some element of the set with some other element not necessarily from the set T. It can be shown that the total comparison cost in each round is no more than the cost of the optimal proof certificate. Since there are at most $n-1$ rounds we get the above completive ratio.

From the preceding discussion we see that restricting ourselves to a more structured cost model might lead to more useful results. For example, a common cost model is the monotone cost model. In this model the cost of comparing a pair is a monotone function of the values of the pair. For example, we can define a monotonic cost function as follows: $\mathscr{C}(u,v) = \max(u,v)$, so the cost of comparing u, v is $\max(u,v)$. Here we abuse the notation to use u, v as both vertices of the graph G and the elements of the set X. *Monotonicity* comes from the fact that increasing the value of one element in the pair will not decrease the comparison cost. Many other semigroup operations satisfy monotonicity. It was shown in [7] that the best one can do, without knowledge of the minimum, is to get an algorithm that is within a logarithmic factor of a cost optimal algorithm. This competitive ratio is computed as follows.

Let us look at a cost model that is not monotonic. Consider first a *bi-colored* cost model. In this model edges of G are colored red or green. G is called the comparison graph. Cost associated with comparing a red edge is r and that for comparing a green edge is g. This model is not monotonic for the following reason. Let $r > g$. For some triple $\{u, v, w\}$ of elements in X with $u > v > w$, let $\mathscr{C}(u, v) = g$ and $\mathscr{C}(u, w) = r$.

If $g = 0$ and $r = 1$ we get the problem of sorting a set of elements where some relation between the pairs are known advance. If the set of green pairs define a poset $\mathfrak{P}$ with $e(\mathfrak{P})$ number of linear extensions then we know there exists a sequence of at most $\log e(\mathfrak{P})$ (red) comparisons that sorts X[2]. However, we do not yet know how to do this algorithmically.

2.1 Restricted Cost Model and Sorting

Inspired by the previous example, we look at a different bi-colored cost model in this thesis. This model has comparison cost of 1 or ∞. A pair with cost ∞ is considered a "forbidden pair". Let G have m edges and q missing edges (forbidden edges). The edge set and the forbidden edge set are denoted by E and $\bar{E}$ respectively.

There are two version of this problem. In the first setting, one is guaranteed that the graph G has enough edges that we can always determine a total order for X. We denote it as $\texttt{FSortingWT}_{n,q}(X, G)$ (here "WT" stands for with total order guarantee). In the other version there is no such guarantee, which is denoted by $\texttt{FSorting}_{n,q}(X, G)$. We do not yet have a non-trivial bound for $\mathscr{C}\mathscr{C}(\texttt{FSortingWT})$. We believe that the comparison tree model is too weak for this purpose. For example, given a *comparison graph* G the number of different acyclic orientations of G gives an upper bound on the number of possible answers as each correspond to a unique total order, if we have the total order guarantee. Since, G has $\leq \prod_{v \in V}(d_v + 1) \leq n^n$ different acyclic orientations[8] we cannot get a better than $\Omega(n \log n)$ for $\mathscr{C}\mathscr{C}(\texttt{FSortingWT})$ using the comparison tree model. We believe this bound to be weak for this problem. In fact for some graphs we get degenerate bounds. For example if

G has $n-1$ edges and so G is a Hamiltonian path. Then we can determine the unique total order by just making one comparison. Since there are only two acyclic orientations of the edges of this Hamiltonian path. A solitary probe is then used to determine the direction of this ordering.

For the latter case where a total ordering may not be achieved we have a better bound. Note that the lower bound is given by finding a specific comparison graph that serves as an obstruction.

Theorem 2.1. There exist some comparison graph G for which, $\mathscr{C}\mathscr{C}(\texttt{FSorting}_{n,q}(X, G)) = \Omega(n^2 - q)$.

Proof. Partition X into sets L and R of equal size. Let G be any bipartite graph on (L, R) with $\binom{n}{2} - q$ edges. We use the adversarial model to prove the lower bound. Given Alice's choice of comparisons, let (u^*, v^*) be a pair not compared by Alice. Choose an (adversarial) partial ordering of X such that for all $u \in L - \{u^*\}$ and $v \in R - \{v^*\}$, $u \geq v$ and make the ordering of u^*, v^* the opposite of what Alice has computed. Clearly, for Alice to be able to determine the correct partial order of X she has to look at all edges of G. $\square$

Recently authors in [9] gave a lower bound of $\Omega(n \log n + q)$. Hence the current lower bound is a combination of these two: $\Omega(\max(n \log n + q, n^2 - q))$. In both of these results a specific graph is use to give the lower bound.

In general given an arbitrary graph the minimum number of relations an omniscient observer must output in a valid proof is the number of covering relations of the underlying partial order determined by the edges of the graph. If the graph is connected then every element must be in a covering relation with some other element: either the element is a cover of some other element or it is covered by some element. Thus there must be at least $n-1$ covering relations. Unfortunately the number of covering relations could be much higher than $n-1$ for a given graph if there is no total order guarantee. Thus giving the algorithmic results in terms of competitive ratios for `FSorting` will not be very useful. Hence we state the performance of our algorithms directly in terms of comparison complexity. This is consistent with previous studies.

2.2 Background

In this thesis we look at $\mathscr{C}\mathscr{C}(\texttt{FSorting})$. First we concern ourselves with only deterministic results. Randomizations are discussed in Chapter 4. The problem is still open for the most part. It is closely related to the problem of *partial sorting* ($\texttt{PSorting}_n(X, \mathbb{Q})$) under a relation oracle model. In this model we are given a set X of elements and a oracle $\mathbb{Q}$ which is used to determine relations between pairs of elements in X. The goal is to determine all the valid relations. Given a pair $u, v \in X$ the query "Is $u \geq v$?" results in yes, no or incomparable ($u \perp v$). The number of queries made to $\mathbb{Q}$ is defined as the *query complexity.* Queries are equivalent to comparisons, except here we allow incomparable as a valid answer. Since there are $\Omega(2^{n^2/4})$ labeled posets with n elements[10], it immediately follows that query complexity

$$\mathscr{C}\mathscr{C}(\texttt{PSorting}_n(X, \mathbb{Q})) = \Omega(n^2)$$

This has been investigated for width-bounded posets in [11], where the authors show that if the underlying partial order of X has width at most w then we require at least $\Omega((w + \log n)n)$ comparisons. They presented a query optimal algorithm for width-bounded posets whose total complexity is $O(nw^2 \log \frac{n}{w})$. This algorithm can be generalized for any poset with an additional $\log w$ factor multiplied to the comparison complexity. Their results were the first major extension in this line of research after the seminal work by Faigle and Turán[12] which only showed the existence of such an algorithm.

Another related problem is the *partial order production problem*, where given a set T with an unknown total order we are interested in determining the partial order of another set S by comparing pairs in T. The goal is do this with the minimum number of comparisons. The reader is referred to the survey by Cardinal et. al. [13] which discusses some of these and other related problems in detail.

A special case of `FSortingWT` is the *nuts and bolts problem.* This is strictly not a sorting problem rather a matching one.

In this problem one is given two sets of elements, a set of nuts and a set of bolts. Elements in each set have distinct sizes and for each nut it is guaranteed that there exists a unique bolt of same size. Matching is performed by comparing a nut with a bolt. However, pairs of nuts or pairs of bolts cannot be compared. So in this case $G = K(N, B)$ is a complete bipartite graph with edges from the set of nuts N to the set of bolts B. This problem has been solved in the mid 1990s [14, 15]. The existence of an $O(n \log n)$-time deterministic algorithm was proven for it using bipartite expanders [14].

In this work we propose the first non-trivial deterministic algorithm for `FSorting`. The results are expressed in terms of n and q. Expressing the results in terms of the number of missing edges fits naturally with the problem. Note q and w are related, where w is the width of the poset $\mathscr{P}$ found after sorting X. We have,

$$q \geq \# \text{ of incomparable pairs in } \mathfrak{P}_G \geq \binom{w}{2}$$

Hence, $w = O(\sqrt{q})$. Here, $\mathfrak{P}_G$ is the partial order obtained after orienting all edges of G based on the underlying order of X. We cannot directly compare the comparison model used here which has only two outcome with the one also allowing incomparability [11]. However the above relation between w and q gives more context for choosing q as a parameter. Secondly, in the absence of any other structural properties of the input graph G, q gives a good indication of how difficult it is to sort G. For example, when $q = O(\log n)$, it is easy to see that one can sort in $O(n \log n)$ total time. To do this we pick an arbitrary pair of non-adjacent vertices and take out one of them, removing it from the graph. We do the same thing with the remaining graph until the graph remaining is a clique. It is clear that we had to take out at most $O(\log n)$ vertices. Then we sort this graph with $O(n \log n)$ comparisons and merge the vertices we had removed previously by checking all the remaining undirected edges, which is at most $O(n \log n)$. On the other extreme, if $m = \binom{n}{2} - q = O(n)$ then it can be shown that we need to make $\Omega(m)$ comparisons. For example in the case of a *path graph* P_n, having 2^{n-1} acyclic orientations, we need at least $n-1$ comparisons.

In this chapter and the following we discuss our deterministic algorithm. In chapter 4 we introduce a randomized algorithm. The deterministic algorithm is introduce in two stages. First we solve a simpler problem where $q = O(n)$. Then we take the main ideas from this algorithm and generalized for arbitrary q. In this chapter we discuss this special case.

Theorem 2.2. If $q = O(n)$ then there is a deterministic algorithm for `FSorting` which uses $o(n^2)$ comparisons and has a total complexity of $O(n^\omega)$.

2.3 Preliminaries

Before we begin with the description of our algorithm we need to introduce some additional terms. Our input is X and the comparison (cost) graph $G(X, E)$. Elements in $\bar{E}$ are called forbidden pairs. Initially all edges in G are undirected. Let G_i be the graph after i-edges have been oriented and P_i be the associated partial order from the transitive closure of the resulting relations. We denote the degree of a vertex v by $d(v)$ and $n(v) = n - 1 - d(v)$ is the number of vertices that are not adjacent to v. The set of neighbors of a vertex v is denoted by $\mathcal{N}(v)$. We use the notation $E(A, B)$ to denote the set of edges between the sets of vertices $A, B \subset X$. We also define the little-o notation to remove any ambiguity from our exposition.

Definition 2.1. If $f(n) \in o(g(n))$ then $f(n) \in O(g(n))$ but $f(n) \notin \Omega(g(n))$.

Lemma 1. Let $\{f_1(n), f_2(n), ..., f_k(n)\}$ be a finite set of non-negative monotonically increasing functions such that for some $g(n)$:

1. $\forall i \; f_i(n) \in o(g(n))$
2. $\sum_i f_i(n) \leq cg(n)$

If $F(n) = \sum_i f_i^2(n)$ then $F(n) \in o(g^2(n))$.

Proof. First we prove $F(n) = O(g(n))$. Clearly,

$$(\sum_i f_i(n))^2 \leq c^2(g(n))^2$$

$$\sum_i f_i^2(n) + 2\sum_{i>j} f_i(n)f_j(n) \leq c^2 g^2(n)$$

$$F(n) \leq c^2(g(n))^2$$

Now we prove $F(n) \neq \Omega((g(n))^2)$. Assume that $F(n) \in \Omega((g(n))^2)$; then there exists some constant $\hat{c}$ and an integer n_1 such that, $F(n) \geq \hat{c}(g(n))^2$ whenever $n \geq n_1$. Now let $f_i(n) \leq c_i g(n)$ whenever $n \geq n_{c_i}$. Since $f_i(n) \in o(g(n))$, we can pick the c_i's arbitrarily small and independent of each other. Now, for $n \geq \max(n_1, n_2)$ (where $n_2 = \max_i(n_{c_i})$) we have,

$$\sum_i f_i^2(n) \geq \hat{c}(g(n))^2$$

$$\sum_i c_i^2 \geq \hat{c}$$

This contradicts the fact that c_i's can be assigned arbitrary values independent of each other. That is, not all $f_i(n)$ will satisfy the condition $f_i(n) \in o(g(n))$ simultaneously. Hence, $F(n) \neq \Omega((g(n))^2)$.

□

Lemma 2. Let $T(n) = \sum_{i=1}^{k} T(n_i) + f(n)$ where $\sum_i n_i \leq \delta n$ for some $0 < \delta < 1$ and $f(n) \in o(n^2)$. Then $T(n) \in o(n^2)$.

Proof. Let as assume $T(n) = \Omega(n^\alpha)$ for some $\alpha \geq 1$, otherwise we are done. Since $\alpha \geq 1$, $T(n)$ is convex and it follows that $\sum_{i=1}^{k} T(n_i) \leq T(\sum_{i=1}^{k} n_i) = T(\delta n)$. So the recurrence becomes $T(n) \leq T(\delta n) + f(n)$. Using the "Master theorem" we see that, $T(n) = \Theta(f(n)) = o(n^2)$. $\square$

2.4 A Deterministic Algorithm When $q = O(n)$

Here we look at a simpler case. We will use some of the main ideas from this algorithm to extend it to unknown q. This algorithm will have a comparison complexity worse than the general version. In this algorithm we shall do case analysis based on whether a certain quantity is $o(n)$ or not. We acknowledge that this is not an algorithmic test. However, we use it in this algorithm to establish a framework for the second algorithm, which uses a traditional test and does not affect the main claims.

2.4.1 Basic Idea

We know that from Turan's theorem that a graph not having a complete subgraph of size p must have $O(\frac{p-2}{2(p-1)}n^2)$ edges. Thus if $q = O(n)$ we see that G must have a reasonably sized clique of $\Omega(n)$. A clique can be sorted using any standard comparison based sorting algorithm. Since q is small, this clique is also well connected to the rest of G. We can use this fact to partition G in a balanced manner and proceed recursively. The main difficulty arise from the following two facts: 1) even though many elements can be partitioned some elements may not be comparable and must be treated separately. 2) the subgraphs formed after the partition may not have many edges. In the rest of this chapter we show how to overcome these problems by using a composition of two recursive algorithm. However it turns out we can avoid this double recursion by doing more work to partition the graph in the first stage. This is done via the use of multiple cliques and it is the subject of next chapter.

2.4.2 A Restricted Case

Assume $q \leq cn$ for some constant c. If $R = \{v \in X \mid n(v) > c_1\}$ for some constant c_1, then $|R| \leq (2c/c_1)n$. This is immediate from the fact that $\sum_v n(v) \leq 2cn$. We choose $c_1 = 4c$. Let $S = X \setminus R$ and $G[S]$ be the induced subgraph generated by S. We have $|S| \geq n/2$ and $|R| \leq n/2$ if $v \in S$ then $n(v) \leq c_1$.

Lemma 3. There exists a subset $K \subset S$ such that $|K| \geq \frac{n}{2(4c+1)}$ and $G[K]$ is a complete graph.

Proof. We construct K explicitly as follows. Pick any arbitrary vertex u in S. Let $K = \{u\}$. Then we pick successive vertices from S iteratively. Let v be the last vertex to be added to K. Since v has at least $|S| - c_1$ neighbors in S, whenever we pick a neighbor of v from S to add to K we loose at most $c_1 + 1$ vertices (including the vertex we picked). Hence if we pick neighbors of v the size of K is at least $|S|/(c_1 + 1) \geq n/2(4c + 1)$.

□

Clearly the above procedure runs in $O(n^2)$ time and makes no comparisons between elements of X. From here onwards running time is used synonymously with total complexity. Now we are ready to describe our algorithm. The main algorithm is recursive. However the recursion is composed over two separate procedures. We shall break up the algorithm into several steps.

2.4.3 Initial Sorting

Given the input graph G, construct $K \subset X$, where the induced subgraph $G[K]$ is a clique, with $|K| \geq n/2(4c + 1)$ (Lemma 3). Let $L = X \setminus K$. Note that $|L| \leq n - n/2(4c + 1) = (8c + 1/8c + 2)n$. Next we sort K using $O(n \log n)$ comparisons. We can use a standard comparison based sorting algorithm for this purpose, as $G[K]$ is a clique. Now we have two possibilities:

Case 1: If $|L| = o(n)$, then we probe all edges of $G[L]$ and $G[K, L]$. Then we take the transitive closure of the resulting relations for which no extra comparisons are necessary. It can be easily seen that the number of comparisons made is $o(n^2)$. For the sake of contradiction if we assume that it is not so then $|K||L| + |L|^2/2 \geq dn^2$ for some d. This implies $|L| \geq dn$, since $|K| + |L|/2 \leq n$. But then, $|L| = \Omega(n)$, which is not true according to our earlier assumption. So, in this case we would have sorted X by making only $o(n^2)$ probes.

Case 2: Otherwise $|L| \geq \delta n$, for some constant δ. In this case we recursively partition L based on elements from K. We call this the partition step.

2.4.4 Partition Step

We will actually recursively partition both K and L. To keep track of the current partition depth we rename K to K_{00} and L to L_{00}. We pick m_{00} the median of K_{00} (after K_{00} is sorted). Since $K_{00} \subset S$ we have $n(m_{00}) \leq c_1$. So m_{00} will be comparable to all but at most c_1 elements of L_{00}. Let

$$A_{00} = \{v \in L_{00} |\ v \in N(m_{00})\}$$

$$B_{00} = L_{00} \setminus A_{00}$$

Note $|B_{00}| \leq c_1$. Let U_{00} be the subset of A_{00} whose elements are $\geq m_{00}$ and the set R_{00} accounts for the rest of A_{00}. Let K_{10} and K_{11} be the elements of K_{00} that are $<$ and $\geq$ to m_{00} respectively. We recursively partition the sets U_{00} and R_{00} using the medians of K_{10} and K_{11}. The B-sets are kept for later processing. We rename the sets U_{00} and R_{00} to L_{10} and L_{11}. So, the pairs (K_{10}, L_{10}) and (K_{11}, L_{11}) are processed as above generating the sets A_{10}, A_{11}, B_{10} and B_{11}. We continue doing this until the size of the K-set is $\leq c_2$, where c_2 is some constant. At this point we don't know the size of the L-set paired with it. There are two cases we need to consider:

Case 1 Let $|L| = o(n)$. Then we probe all the edges of $G[L]$ and $G[K, L]$ which uses at most $c_2|L| + \binom{|L|}{2}$ number of comparisons.

Case 2 Otherwise $|L| \geq \delta n$, for some constant δ. Hence the graph $G[L]$ can have at most $\leq (c/\delta)|L|$ missing edges. This satisfies our initial premise that the number of missing edges in $G[L]$ is linear in the number of vertices. Hence we can apply our initial strategy recursively. That is we first find a large enough clique (which according to Lemma 3 must exist) and then use it to partition the rest of the set L.

Note that (c/δ) is an absolute constant. If the input graph has at most cn missing edges we apply the procedure recursively to subgraphs whose number of missing edges are at most (c/δ) times the number of vertices in the subgraph at any level of recursion. This (c/δ) factor is not successively multiplied within each level of recursion.

Let us visualize the partitioning using a partial recursion tree T (see Figure 2.1). We shall call T the "partial recursion tree" for reasons that will soon be clear. At the root we have the pair (K_{00}, L_{00}). It has two children node (K_{10}, L_{10}) and (K_{11}, L_{11}) each having two children of their own and so on. Now at each level, the size of the K-set gets halved. So the number of levels in T is at most $O(\log n)$. However, the L-sets need not get divided in equal proportions. So, at the frontier (the deepest level) we will have nodes of the above two types, depending on the size of their corresponding L-sets. Let the collection of these frontier nodes be partitioned in two sets F_1 and F_2 corresponding to case 1 and case 2 respectively.

Computing comparisons in F_1: We can conclude that the total number of probes needed to compute all relations in F_1 is $o(n^2)$. This follows from Lemma 1. Here we can map the size of the L-sets of the nodes in the collection F_1 to functions $f_i(n)$. We know that the total elements in the union of these L-sets is $\leq |L_{00}| \leq \frac{8c+1}{8c+2}n$. The total number of comparisons will be $F(n)$ in worst case. What is the total number of comparisons on the internal nodes of T? We know that in the internal nodes we compare the median of

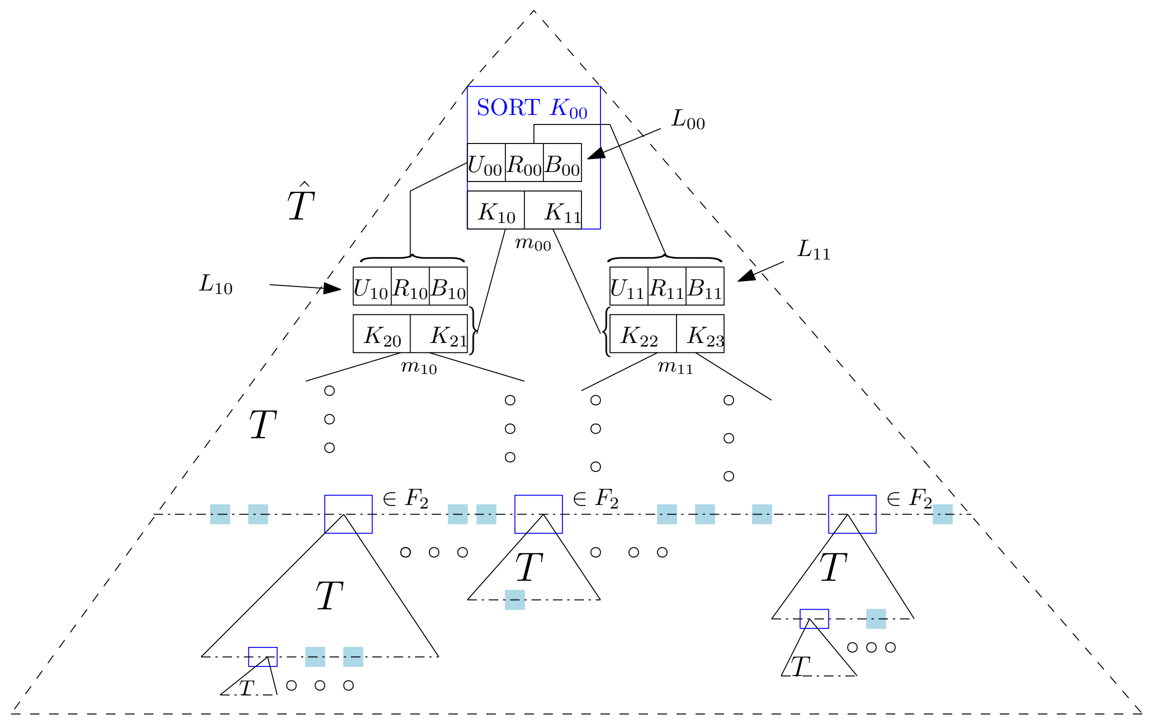


Figure 2.1: Visualizing the steps. At the bottom of T the shaded boxes represents the F_1-nodes and the blue rectangles the F_2-nodes. The outer dashed triangle represents the full tree $\hat{T}$. The tree $\hat{T}$ is created during the partitioning step and in the merge step we start from the deepest leaves of $\hat{T}$ and move upwards.

the K-set with the elements of the A-set, which takes $|A|$ probes. Since the union of these A-sets cannot exceed the total number of vertices in G, at each level of T we do at most $O(n)$ probes, totaling to $O(n \log n)$ probes over all the internal nodes.

Computing comparisons in F_2: Unlike the nodes in F_1, the nodes in F_2 recursively call the "initial strategy" using the input graph $G[L]$. Let the comparison complexity of our initial strategy be $Q(n)$. Then the recursion for Q is as follows:

$$Q(n) = \sum_{i=1}^{|F_2|} Q(n_i) + o(n^2)$$

Here we assume that the nodes in F_2 are indexed according to some arbitrary order. We can solve this recurrence using Lemma 2, giving $Q(n) \in o(n^2)$. This follows from, $\sum_{i=1}^{|F_2|} n_i \leq \frac{8c+1}{8c+2}n$. Note here that $|F_2|$ is bounded by a constant since the size of the L-sets are $\geq \delta n$.

We call $\hat{T}$ the *full* recursion tree. All leaf nodes in $\hat{T}$ are in F_1. It is straightforward to show that $\hat{T}$ has $O(\log^2 n)$ levels. Since any of the leaf nodes of T has $|L| \leq \beta n$ (where $\beta = \frac{8c+1}{8c+2}$), its subtree in $\hat{T}$ for some constant α can have at most

$$\alpha \log \beta n = \alpha \log n - \alpha\beta$$

levels, and any of its leaves having at most $\alpha \log n - 2\alpha\beta$ levels, and so on.

2.4.5 Merge Step

Once we have completed building $\hat{T}$ we proceed with the final stage of our algorithm. Recall that during the forward partition step we had generated many of these B-sets in the internal nodes of $\hat{T}$. Now we start from the leaves of $\hat{T}$ and proceed upwards. Each pair of leaf nodes l and r sharing a common parent p, sends a partial order to it (computed as in case 1). When we merge these two partial orders we know that no extra comparisons are needed since they have already been split by the median of the K-set of p. What remains is to compare all edges between the B-set in p and elements in this newly merged partial order (which constitutes the set of elements $A \cup K$ of the node p) as well as the edges in $G[B]$. Then we pass the resulting partial order to the parent of p, and so on. Since the size of the B-sets are bounded by c_1 (at any level in $\hat{T}$), total number of comparisons we make is then $\leq c_1 \sum_i (|A_i| + |K_i| + c_1)$, where sum is taken over all the nodes in that level. Hence this is bounded by $c_1 n$. So at each level we do at most $O(n)$ comparisons in the backward merging step. Since there are at most $O(\log^2 n)$ levels, it totals to $O(n \log^2 n)$ additional comparisons. Adding this to the comparison cost of partitioning in the forward step does not effect the total comparison complexity, which was $o(n^2)$. The final step is to compute the transitive closure of the resulting set of relations, which can be done without any additional probing. Since computing the transitive closure is equivalent to boolean matrix multiplication[16] therefore the total complexity is $O(n^{\omega})$. Recall $\omega \in [2, 2.38]$ is the exponent in the complexity of matrix multiplication.

Chapter 3: Restricted Sorting Continued

In the last chapter we saw how we can use a clique of G to efficiently partially partition of elements of X. A natural question to ask here is if we can employ multiple cliques at a time. First, we show that if G has enough edges then we can get a large collection of cliques that are relatively big. Second, we will show how we can use them to efficiently partition X.

The sets R and S are defined analogously to last chapter.

$$R = \{v \in V \mid n(v) > c_1 q/n\}$$

for some constant c_1. With $c_1 = 4$, we get $|R| \leq \delta_1 n$ where $\delta_1 \leq 2/c_1 = 1/2$. Hence $|S| \geq (1 - \delta_1)n \geq n/2$. Now we will apply Lemma 3 successively to construct a "large-enough" set $K \subset S$ which we will use to find an approximate median of X. The set K consists of disjoint subsets K_i such that $G[K_i]$ is a clique.

3.1 Constructing the Set K

We construct the first clique $K_1 \subset S$ using the method detailed in Lemma 3. Let us define $S_i = S \setminus \bigcup_{j=1}^{i} K_j$. There are two cases:

Case 1: $q < n$: In this case we have,

$$|K_1| \geq (n/2)/(c_1 q/n + 1) \geq (n/2)/(c_1 + 1) \geq n/10$$

We take the first $n/10$ elements (if K_1 turns out to be bigger than $n/10$) and keep the rest for the second round. Now we construct the second clique K_2 from S_1 which has at least $2n/25$ vertices. We let $K = K_1 \cup K_2$. Hence K has at least $9n/50$ vertices.

Case 2: $q \geq n$: In this case we have

$$|K_1| \geq (n/2)/(c_1 q/n + 1) = (n^2/2)/(4q + n) \geq n^2/10q$$

Again we take $|K_1| = (1/10)n^2/q$ discarding some vertices if necessary. Similarly we construct $K_2 \subset S_1$. It can be shown that $|K_2| \geq (n^2/10q)(1 - n/5q)$ and we keep $(n^2/10q)(1 - n/5q)$ vertices in K_2 and the rest are discarded to be processed in the next round. In general for the clique K_r we have $|K_r| \geq (n^2/10q)(1 - n/5q)^{r-1}$. Now we let $K = \bigcup_{i=1}^{r} K_i$. We will show that $|K| \geq \delta_2 n$ for some constant $\delta_2 > 0$. Let $r = 5q/n + 1$. Then we have

$$|K_r| \geq (n^2/10q)(1 - n/5q)^{r-1} \geq (n^2/10q)(1 - n/5q)^{5q/n} > 3n^2/100q$$

since $q \geq n$. Hence, $|K| = \sum_{i=1}^{r} |K_i| \geq r|K_r| \geq (9/50)n$, giving $\delta_2 = 9/50$. Now for each K_i $(1 \leq i \leq r)$ we keep a subset K_i' of size $|K_r|$ and throw away the rest. Clearly, for each i, the induced sub-graph $G[K_i']$ is also a clique. Let $K' = \bigcup_{i=1}^{r} K_i'$. We also have $|K'| \geq (9/50)n$.

3.1.1 Computing An Approximate Median Of X

We shall compute an approximate median with respect to all the vertices of G (the set X) and not just the set S. This element divides the set X in constant proportions. We do this using the cliques in K'. For each K_i' we find its median using $\Theta(|K_i'|)$ probes since $G[K_i']$ is a complete graph. Let this median be m_i and M be the set of these r medians. Since $m_i \in S$, $n(m_i) \leq 4q/n$. We define the upper set of $m \in M$ with respect to a set $A \subset X$ (m

may not be a member of A) as

$$U(m, A) = \{a \in A \mid a \geq m\}$$

Similarly we define the lower set

$$L(m, A) = \{a \in A \mid a < m\}$$

The sets $U(m, K')$ and $L(m, K')$ need to be determined. However, m may not be a neighbors of all the elements in K'. So we compute the upper and lower set approximately by probing all the edges in $E(\{m\}, K' \setminus \{m\})$. These sets are denoted by $\widetilde{U}(m, K')$ and $\widetilde{L}(m, K')$ respectively. There exists some $m \in M$ which divides K' into sets of roughly equal sizes (their sizes are a constant factor of each other); in fact the median of M is such an element. However the elements in M may not be a clique. Hence we will approximate m using the ranks of the elements in M with respect to the set K' (which is $|\widetilde{L}(m, K')|$). Let m^* be picked using the above procedure. Next we prove that the element m^* is an approximate median of M. It is also an approximate median of K'.

Lemma 4. The element m^* picked as described above is an approximate median of K', for $n^2 \geq 200q$.

Proof. First we show that the median of M is an approximate median of K'. Let us take the elements in M in sorted order $(m_1, ..., m_r)$, so the median of M is $m_{\lfloor r/2 \rfloor}$. Now

$$|L(m_{\lfloor r/2 \rfloor}, K')| \geq \left| \bigcup_{i=1}^{\lfloor r/2 \rfloor} L(m_i, K_i) \right|$$

Since the sets K_i are disjoint and $|L(m_i, K_i')| \geq |K_r|/2$, we have (ignoring the floor)

$$|L(m_{\lfloor r/2 \rfloor}, K')| \geq |K_r| r/4$$

Similarly we can show that $|U(m_{\lfloor r/2 \rfloor}, K')| \geq |K_r|r/4$. Hence $m_{\lfloor r/2 \rfloor}$ is an approximate median of K'. Now we show that

$$| \, |L(m^*, K')| - |L(m_{\lfloor r/2 \rfloor}, K')| \, |< 4q/n$$

Consider the sorted order of elements in M according to $|\widetilde{L}(m^*, K')|$. Since each $m \in M$ has at most $4q/n$ missing neighbors in Y, we have $| \, |\widetilde{L}(m, K')| - |L(m, K')| \, |< 4q/n$. So the rank of an element in the sorted order is at most $4q/n$ less than its actual rank. Thus an element m^* picked as the median of M using its approximate rank $|\widetilde{L}(m, K')|$ cannot be more than $4q/n$ apart from $m_{\lfloor r/2 \rfloor}$ in the sorted order of K'. Hence

$$|L(m^*, K')| \geq |K_r|r/4 - 4q/n \geq 9n/200 - 4q/n \geq n/40 \tag{3.1}$$

whenever $n^2 \geq 200q$. In an identical manner we can show that $|U(m^*, K')| \geq n/40$. Hence, m^* is an approximate median of K'. When $q < n$ we just take m^* as the median with the higher $|\widetilde{L}(\cdot, K')|$ value, which guarantees $|L(m^*, K')| \geq n/40$ whenever $n^2 \geq 800q/13$. So we take $n^2 \geq 200q$ to cover both the cases. □

It immediately follows that m^* is also an approximate median of X with both $|L(m^*, X)|$ and $|U(m^*, X)|$ are lower bounded by $n/40$. Lastly, we note that the above process of computing an approximate median makes $\Theta(q + n)$ comparisons. This follows from the fact that computing the medians requires $\Theta(n)$ comparisons in total and for each of the $\leq 5q/n + 1$ medians we make $O(n)$ probes.

3.1.2 A Divide-and-Conquer Approach

Now that we have computed an approximate median of X we proceed with a recursive approach. Let m^* be the approximate median. As with our algorithm in the previous chapter we partition X into three sets U, L and B. The U and L are the upper and lower sets with respect to m^*. B is the set of vertices that do not fall into either, because they are not neighbors of m^*. Since $m^* \in S$ we have $|B| \leq 4q/n$. We recursively proceed to partially sort the sets U and L with the corresponding graphs $G[U]$ and $G[L]$ and keep B for later processing (as we did in the merging step previously). Like before we can imagine a recursion tree T. Let $\bar{E}_P$ be the set of forbidden edges in $G[P]$, the graph corresponding to some node in the recursion tree T. We take $n_P = |P|$ and $q_P = |\bar{E}_P|$.

Case 1: When $n_P^2 \geq 200q_P$ we recursively sort P. In this case we can guarantee that the approximate median m_P^* of P will satisfy equation (3.1). That is both $|L(m_P^*, P)|$ and $|U(m_P^*, P)|$ is $\geq n_P/40$.

Case 2: Otherwise we compare all edges in $G[P]$. In this case P will become a leaf node in T.

It can be seen that the depth of the recursion tree is bounded by $O(\log n)$. At each internal node P of T we pass sets of constant proportions (where the size of the larger of the two set is upper bounded by $(39/40)n_P$) to its children nodes.

3.1.3 Merge Step

In this step we start with the leaves of T and proceed upwards. A parent node P gets two partial orders from its left and right children respectively. Then it probes all the edges between its B-set and these partial orders to generate a new partial order and pass it on to its own parent. This step works exactly as the "merge step" of the previous algorithm. The only difference is that the B-sets here may not be of constant size but of size $\leq 4q/n$.

3.1.4 Comparison Complexity

We can determine the comparisons complexity by looking at the recursion tree T. First we determine the complexity of the forward partition step. At each internal node of T we compute a set of medians and pick an appropriate element from it. Then we partition the set of elements at the node by comparing all edges between the selected element and rest of the elements in the node. As mentioned before, this only takes $\Theta(q_P + n_P)$ comparisons for some internal node P. We assume that all the leaves of T are at the same depth, otherwise we can insert internal dummy nodes and make it so. Nodes that are in the same level are vertex disjoint, hence the total sum of all the vertices in these nodes are $\leq n$. Similarly, the total number of the forbidden edges is $\leq q$. Hence we perform $O(q+n)$ comparisons at any internal level of T. With $O(\log n)$ internal levels in T the number of comparisons made is $((q+n)\log n)$ during the forward partition step. If P is a leaf node then we compare all edges in $G[P]$. There are at most $\binom{n_P}{2} - q_P$ edges in $G[P]$. Since P is a leaf node, according case 2, $n_P^2 < 200q_P$. Hence we make $\binom{n_P}{2} - q_P = O(q_P)$ comparisons. Aggregating this over all the leaves gives a total of $O(q)$ comparisons. Thus total number of comparisons made during the forward step is $O((q+n)\log n)$.

Now we analyze at the merging step. Merging happens only at the internal nodes. Look at an arbitrary internal level of T. At each node P of this level we compare all the edges in $E(B_P, U_P \cup L_P \cup m_P^*)$ and in $G[B_P]$. We do not have to make any comparisons between U and L as they were already separated by the approximate median m_P^*. Hence the total number of comparisons made in this node is

$$\leq (|U_P| + |L_P| + |B_P| + 1)|B_P| \leq (n_P)(4q_P/n_P) \leq 4q_P$$

Summing over all the nodes at any given level gives us $O(q)$ as the comparison complexity per level. So the total number of comparisons during the merging stage is $O(q\log n)$. Combining the two halves, partition step and merge step, we see that the total number comparisons needed to (partially) sort X is $O((q+n)\log n)$.

Total Complexity

Now we look at the total complexity of the previous procedure. Again the analysis is divided into the partition step and the merge step. During the partition step at each node P we perform $O(n_P^2)$ operations. This includes computing the degrees, finding the cliques, computing the approximate median. So at any level of T, whether it is an internal level or not, we perform $O(n^2)$ operations. Hence, in total $O(n^2 \log n)$ operations are executed in the partition step. However this is a conservative estimate and we can remove the $\log n$ factor as argued below: we can define the recurrence for the forward computation as,

$$T(n) = \begin{cases} T(n/40) + T(39n/40) + O(n^2) & \text{if } n^2 \geq 200q \\ O(q) & \text{otherwise} \end{cases} \tag{3.2}$$

This follows from the previous discussion. If we don't recurse on a node we guarantee that $n_P^2 < 200q_p$ for that node. Hence, we have $T(n) = O(n^2 + q)$ using the Akra-Bazzi method[17]. In the merge step, we only make $O(q_P)$ comparisons at any given node. We compute transitive closures only at the leaves. However for any leaf P we have $n_P^2 < 200q_P$. Hence computing the transitive closure of $G[P]$ takes $O(q_P^{\omega/2})$ time using the boolean matrix multiplication method. Hence, the total complexity of the above procedure is $O(n^2 + q^{\omega/2})$. We summarize the results of this chapter with the following theorem:

Theorem 3.1. Given a set X with n elements and a graph G on X having q missing edges, one can partially sort X with $O((q+n)\log n)$ comparisons and in total $O(n^2 + q^{\omega/2})$ time.

Proof. Follows from the preceding discussions. $\square$

Chapter 4: Restricted Comparison Model Under Randomization

So far we have discussed CBPs only in terms of deterministic algorithms. In this chapter we look at randomized algorithms. Let us start with some definitions. We also introduce randomized algorithms using comparison trees. For some comparison-based problem $\mathtt{P}$ let $\mathfrak{A}_{\mathtt{P}}$ be the set of all comparison trees (as defined in Chapter 1) that solves $\mathtt{P}$. Then a randomized algorithm A for $\mathtt{P}$ is determined by a probability distribution D over the collection $\mathfrak{A}_{\mathtt{P}}$. That is, given some input of $\mathtt{P}$, the algorithm A picks a comparison tree T with probability $\mathrm{Prob}_D[T]$ according to the distribution D and proceeds to solve $\mathtt{P}$ for that input using the selected tree.

Definition 4.1. The randomized comparison complexity $\mathscr{R}(\mathtt{P})$ is defined to be the expected number of comparisons required by a randomized comparison based algorithm (RCBA) which uses the optimal probability distribution over the collection $\mathfrak{A}_{\mathtt{P}}$.

However the above definition does not necessarily give us a procedure for computing randomized comparison complexity. Let $\mathfrak{I}_{\mathtt{P}}$ be the collection of all input instances of $\mathtt{P}$. We can use Yao's minimax principle[18] to determine the randomized comparison complexity:

Theorem 4.1. (Minimax) For some arbitrary probability distribution $D_{\mathfrak{I}}$ over $\mathfrak{I}$, $\mathscr{R}(\mathtt{P})$ is at least the expected number of comparison required by any optimal deterministic algorithm that solves $\mathtt{P}$ when the input is chosen according to $D_{\mathfrak{I}}$.

We can choose the distribution $D_{\mathfrak{I}}$ ourselves, however, the deterministic algorithm whose expected runtime we are planning on computing has access to this distribution in advance. We can also express this principle using the comparison tree paradigm. Let T be a comparison tree that solves $\mathtt{P}$. Let $I \in \mathfrak{I}_{\mathtt{P}}$ chosen according to $D_{\mathfrak{I}}$ and p_I be the path in T

that corresponds to the sequence of comparisons. Here the path p_I is a random variable and its expected value is the number of comparisons needed on average to solve `P` using T. So one way to determine randomized comparison complexity is to choose an optimal comparison tree whose leaves are all at some depth $f(n)$, only dependent on the size of the input n. Then we can choose $D_{\mathfrak{J}}$ as the uniform distribution and get a lower bound of $f(n)$ for the average case complexity for randomized algorithms. However many CBPs do not have optimal comparison trees whose depth only depend on the size of the input. In fact for many CBPs we do not yet know an optimal comparison tree. One classical example is the Minimum Spanning Forest problem. For these cases we cannot use the above method effectively.

Just like we expressed randomized algorithms with comparison trees we can also discuss them using Pointer Machines. The only difference is that we work with a probability distribution over the Pointer Machines solving `P`. Thus most of the concepts we introduce carry earlier over in to the randomized setting.

For randomized algorithms there are two ways to qualify the complexity results. One is using expectations and the other is with high probability. We say that an algorithm makes at most $f(n)$ number of comparisons with high probability if the probability that the number of comparisons made is greater than $f(n)$ for some input is $o(1)$. That is, the probability tends to 0 as n tends to infinity. However this is generally a weaker guarantee than saying the algorithm makes $f(n)$ comparisons in expectation. Using Chernoff bounds we can show that the latter implies the former. However, it is not always possible to express results in terms of expectations.

4.1 Restricted Sorting With Randomization

In the context of randomized algorithms, this problem has been studied in [19, 20]. The authors in [19] proposed a randomized algorithm that sorts G with a probe complexity of $\widetilde{O}(n^{3/2})$ with high probability. However their implementation uses a sub-routine that is a polynomial time uniform sampling algorithm to sample points from a convex polytope[21]. The authors did not discuss the exact bound for the total time complexity in their paper. Briefly, their algorithm works as follows. At each step the algorithm either finds a edge which is a balancing pair or finds a subset of elements that can be sorted quickly. Their algorithm relies on an extension of the balancing pair results on posets, where they show, a pair can be balancing even if their average over the linear extensions is more than 1 but bounded by some constant. They use a randomized sampling scheme to sample from the convex polytope of the currently known partial order to estimate the average rank of elements over the set of all linear extension and use it to determine a balancing pair.

In this section we give the following results. First, a randomized algorithm which sorts X with $O(n^2/\sqrt{n+q}+n\sqrt{q})$ comparisons with high probability. We use a random graph model for this purpose. The main feature of our randomized algorithm is that it does not need the expensive subroutine to sample points from a convex polytopes. This reduces the total complexity for our algorithm. The second result is specific to random graphs. If G is a random graph with edge probability p (that is an edge is present with probability p independent of other edges) we show that one can sort G with high probability using only $\widetilde{O}(\min(n^{3/2}, pn^2))$ comparisons. These are discussed in the following sections.

4.2 A Randomized Algorithm

In this section we look at a more direct way of sorting by making random comparisons. The proposed method is inspired by the literature on two-step oblivious parallel sorting algorithms [22, 23]. In particular Bollobás and Brightwell showed certain sparse graphs can be used to construct efficient sorting networks [24, 25]. Oblivious sorting algorithms are discussed in detail in the second part of this thesis, but, for our discussion here the reader does not need to know any terminology about these algorithms. It was shown that if a graph satisfies certain properties then comparing its edges and taking the transitive closure of the resulting relation set would yield a large number of additional relations. Then we just probe the remaining edges that are not oriented, which is guaranteed (with high probability) to be a "small" set.

The main idea is as follows: Let $\mathscr{H}_n$ be a collection of undirected graphs on n vertices having certain properties. An acyclic orientation of a graph $H(X,E) \in \mathscr{H}_n$ is an ordering of V and the induced orientation of the edges of H based on that ordering. Let σ be a total ordering on X and $\mathfrak{P}(H,\sigma)$ be the partial order generated by this ordering σ on H. It is a partial order since H may not be sortable. Let $t(P)$ be the number of incomparable pairs in $\mathfrak{P}$. We want H to be such that $t(p)$ is small. If that is the case then $\mathfrak{P}$ will have many relations and if H is sparse then we can compare all the edges of H and afterwards we will be left with comparing only a small number of pairs. These are pairs which were not oriented during the first round of comparisons and after the transitive closure computation. A graph H is *useful* for our purpose if every acyclic orientation of H results in many relations. We want to find a collection $\mathscr{H}_n$ such that every graph in it is useful with high probability.

We extend the results in [24,25] to show that a collection of certain conditional random graphs are useful, with high probability. In our case this random graph will be a spanning subgraph of the input graph G. Here we recall an important result from [24] (Theorem 7) which we will use in our proof.

Theorem 4.2 ([24])**.** If G is any graph on n vertices and G satisfies the following property:

Q1 Any two subsets A, B of vertices having size l have at least one edge between them.

Then the number of incomparable pairs in $\mathfrak{P}(G,\sigma)$ is $O(nl \log l)$ for any ordering σ of X.

The input graph G is chosen by our adversary. However, we show that any random spanning subgraph of G with an appropriate edge probability will satisfy Q1 with high probability. Let $H_{n,p}(G)$ be a random spanning subgraph of G, where $H_{n,p}(G)$ has the same vertex set as G and a pair of vertices in $H_{n,p}(G)$ has an edge between them with probability p if they are adjacent in G, otherwise they are also non-adjacent in $H_{n,p}(G)$. All we need to prove is that any random spanning subgraph $H_{n,p}(G)$ given G with n-vertices and edge probability p will satisfy Q1 with high probability. Since G has at most q forbidden edges any two subsets of vertices A, B (not necessarily distinct) of size l must have at least $\binom{l}{2} - q$ edge between them. Let E_{AB} be the event that the pair (A, B) is bad (they have no edges between them), then the probability $S_{n,p}$ that there exists a bad pair is:

$$S_{n,p} := \text{Prob}\left(\bigcup_{i,j} E_{A_iB_j}\right) \leq \sum_{i,j} \text{Prob}(E_{A_iB_j}) \leq \sum_{i,j} (1-p)^{e(A_i,B_j)} \tag{4.1}$$

where the sum is taken over all such $\binom{n}{l}^2$ pairs of subsets, and the number of edges between the two sets A and B in G is $e(A, B) \geq \binom{l}{2} - q$. So we have,

$$\begin{aligned} S_{n,p} &\leq \binom{n}{l}^2 (1-p)^{\binom{l}{2}-q} \leq \binom{n}{l}^2 e^{-p(\binom{l}{2}-q)} \\ &\leq \left(\frac{en}{l}\right)^{2l} e^{-p(\binom{l}{2}-q)} = \exp(2l(\log en/l) - p(\binom{l}{2} - q)) \end{aligned}$$

Since, $e^{-x} \geq 1 - x$. Hence $S_{n,p} \to 0$ as $n \to \infty$ whenever

$$\exp(2l(\log en/l) - p(\binom{l}{2} - q)) = o(1) \tag{4.2}$$

Given $q < \binom{n}{2}$ it is always possible to find appropriate values for p and l as functions of q and n such that $S_{n,p} = o(1)$. For some value for the pair (p, l), we see that in the first round we make $O(pn^2)$ comparisons with high probability and in the second round $O(nl \log l)$ comparisons (for the remaining unoriented edges) again with high probability. So the comparisons complexity is $\widetilde{O}(pn^2 + nl)$. We can choose appropriate values of p, l which satisfy equation 4.2 such that the comparison complexity becomes $\widetilde{O}(n^2/\sqrt{q+n} + n\sqrt{q})$. We summarize this section with the following theorem:

Theorem 4.3. Given a graph G on n vertices and q forbidden edges one can determine the partial order on G with high probability in two steps by probing only $\widetilde{O}(n^2/\sqrt{q+n}+n\sqrt{q})$ edges in total and in $O(n^\omega)$ time.

Proof. Follows from the preceding discussions and the $O(n^\omega)$ total complexity is due to taking the transitive closure. □

4.3 When G Is A Random Graph

The above technique can easily be extended for the case when the input graph G is random. Let $G_{n,p}$ be the input graph having n vertices and an uniform edge probability p. For such a graph we can use equation 4.1 to bound $S_{n,p}$ as follows:

$$S_{n,p} \leq \binom{n}{l}^2 (1-p)^{l^2} \leq \exp(-pl^2 + 2l\log n)$$

We can choose any $l > 2\log n/p$ such that $S_{n,p} \to 0$ as $n \to \infty$. Let $l = 3\log n/p$. Using Theorem 4.3 we have $t(G_{n,p}) = \widetilde{O}(nl) = \widetilde{O}(n/p)$. Since $G_{n,p}$ has $pn^2/2$ edges (with high probability) the critical value of p when $t(G_{n,p}) = pn^2/2$ is $\widetilde{O}(1/\sqrt{n})$. Let this be $\hat{p}$. Hence if $p > \hat{p}$, we can sort by making only $\widetilde{O}(n^{3/2})$ comparisons. Since given $G_{n,p}$ we can construct an induced subgraph $G_{n,\hat{p}}$ and use it as the random graph in our previous construction. Otherwise we just probe all the edges which makes $O(pn^2)$ comparisons. Thus we can sort $G_{n,p}$ with at most $\widetilde{O}(\min(n^{3/2}, pn^2))$ comparisons with high probability. Hence, we get an elementary technique to sort a random graph with at most $\widetilde{O}(n^{3/2})$ comparisons. The algorithm in [19] has a slightly better bound of $\widetilde{O}(n^{7/5})$ comparisons. However, the total runtime of the algorithm in [19] is only polynomially bounded when p is small. In our algorithm we need compute the transitive closure only twice making it run in $O(n^\omega)$ total time.

Chapter 5: Set Maxima

In the last few chapters we looked at the problem of sorting when some comparisons are forbidden. However, we did not have any additional structures along with the input set X that determined the relations we *needed* to compute. The comparison graph G seemed like this additional structure, but its purpose was only to specify what relations we *can* compute. In this chapter we look at a CBP where comparison costs are uniform but an additional structure is given in the form of a hypergraph which determines the type of relations we *have* to compute.

The *Set Maxima* problem was first introduced in [26], in the context of finding lower bounds for shortest path problems. It was shown at the time that the comparison tree complexity of the problem was weak. The general problem still remains unsolved and along with the Minimum Spanning Forest problem, is one of the outstanding problems in computer science.

5.0.1 Background

We begin by formally introducing the problem and some necessary notations. As before, X will be a set of n elements. The set $\mathbb{Q}$ contains all total orderings of X. Here we do not have any restriction on comparing certain pairs of X, so we can always determine the underlying total order if necessary. Let $\mathfrak{S}$ be a collection of m distinct subsets of X. The sets in the collection are labeled from 1 to m and let S_i be the set with labeled i. Let $\sum_{i=1}^{m} |S_i| = p$. This is the input size of our problem. The Set Maxima problem ($\mathtt{SetMaxima}_{\mathtt{n,m}}(\mathtt{X}, \mathfrak{S})$) asks to determine the maximum of each of the sets in the collection. Specifically we are interested in determining the number of comparisons necessary and sufficient to solve the problem. We will use the Pointer Machine model here. So only comparisons between pairs of elements are counted towards the comparison complexity. Like before we distinguish it from other memory-type operations such as set memberships, computing the union / intersections of the sets etc. They are taken into account only when determining the total complexity.

The best known lower bound for the problem under the comparison tree model is no better than the trivial bound of $O(m+n)$. This was shown in [26] using the *s-uniqueness* property. The original proof was given by Michael Fredman. The best upper bound for the problem is a combination of several trivial upper bounds and is summarized as $O(\min(n+m2^m, n\log n))$. The first term is the results of the following procedure: the sets in the collection $\mathfrak{S}$ can be represented by $2^m - 1$ (possibly empty) pairwise disjoint sets $\mathfrak{S}'$. For example if $m = 2$ we can take $\mathfrak{S}'$ as the following three sets $S'_1 = S_1 \setminus S_2$, $S'_2 = S_2 \setminus S_1$ and $S'_3 = S_1 \cap S_2$. Once we have computed the maximum for each set in $\mathfrak{S}'$ it takes at most $2^m - 1$ comparisons per set in $\mathfrak{S}$ to determine its maxima. The $n \log n$ term comes from the following simple observation: If we sort the set X then without any further comparisons we can determine the maximum of each set by simply iterating over the sorted list from largest to smallest and doing membership queries. In the next section we briefly go over some non-trivial results for certain special cases of the set maxima problem, but first we define the $\mathtt{SetMaxima}_{\mathtt{n,m}}(\mathtt{X}, \mathfrak{S})$ formally:

Definition 5.1. Given an set $X = \{x_1, ..., x_n\}$ with an unknown total order and a collection of subsets $\mathfrak{S}$ ($|\mathfrak{S}| = m$) of X the `SetMaxima` problem asks how to determine the maximum element of each of the subsets of $\mathfrak{S}$ in $O(n+m)$ comparisons, if possible.

Note that we do not use the input size p as one of our output parameters, which could possibly be exponential. This follows from the fact that we are only interested in counting comparisons and operations like reading the input, testing set membership could be said to be “free”.

5.1 Previous and Related Work

In the context of lower bounds, Graham and others investigated the comparison complexity of the set maxima problem [26]. They showed that the number of different arrangements (of maximum elements) given a pair $(X, \mathfrak{S})$ is $\binom{m+n-1}{n-1} < 2^{m+n}$. This gives a $\Omega(n+m)$ lower bound. A tighter analysis of the quantity $\log(\binom{m+n-1}{n-1})$ gives $\Omega(n \log(1+m/n))$ as the lower bound. The proof is based on the notion of *s-uniqueness* and *list representatives* which they introduced in the paper.

There have been four major results for the set maxima problem. We start with the randomized algorithm proposed in [27]. The main idea behind their algorithm is to: 1) choose a random sample of elements (say R),2) sort this set, 3) Use this set to compute the reduced sets S_i' such that $S_i' \subset S_i$ (where elements of S_i' are greater than all elements in R that are also in S_i), 4) solve the set maxima problem on the reduced sets. The critical task is to compute the reduced sets efficiently in expectation. They show that the expected number of comparisons in their algorithm is $O(n \log((m+n)/n))$ which is optimal according to the comparison tree complexity. In fact, the randomized algorithm solves a more general problem of computing the t-maxima, that is, for each subset $S_i \in \mathcal{S}$ it determines its largest t elements. As we shall see shortly this randomized algorithm borrows concepts from a deterministic algorithm proposed in [28].

Moving on to deterministic algorithms, we first discuss the generic algorithm proposed in [28] and then show how it was applied to solve the set maxima problem when the elements of X are vectors and elements of $\mathfrak{S}$ are hyperplanes. The pseudo-code of the generic RankSequence algorithm 1 is given above. The rank sequence R is determined based on the properties of the subset system. It is easy to see that once the rank sequence is computed determining the maximum for each sets s_j', which are called the reduced sets, can be done in time linear in the size of the sets. Therefore, the complexity of the generic RankSequence is:

$$T(n,m) = f_r(n,m) + \sum_{j=1}^{m} |s'_j| \tag{5.1}$$

The term $f_r(n,m)$ is the complexity of computing the rank sequence. Note that line 4 does not require any additional comparisons. The second term accounts for the work done on line 5 where we compute the maxima for each set by brute force. It was shown that when the sets are determined by hyperplanes, with $m = n$, then $T(n,m)$ is linear. This was proved using the fact that number of hyperplanes a point can belong to is limited. We can choose a rank sequence based on this fact that can be computed in linear time. They go on to show that when the set memberships are random we can also solve set maxima with linear number of comparisons with high probability. By random set membership we mean: the probability that $x_i \in s_j$ is p for all set-element pairs. It can also viewed as a random hypergraph with m edges.

Algorithm 1: The Generic RankSequence Algorithm [28]

```
Input  : $X, \mathfrak{S}$
Output: $(x_{j_1}, ..., x_{j_m})$, where $x_{j_l} = \max s_j$
1 $R \leftarrow (r_0, r_1, ..., r_{k+1})$ ; // Integers such that $r_0 = n > r_1 > ... > r_{k+1} = 1$
2 Compute $Z = (z_0, z_1, ..., z_k)$ such that $z_i$ is the element in $X$ whose rank is $r_i$
3 Let
    $Z_0 = \{x \in X |\ x \leq z_0\}$
    $Z_i = \{x \in X |\ z_i \leq x \leq z_{i+1}\}$
    $Z_k = \{x \in X |\ x > z_k\}$
4 For all $1 \leq j \leq m$ let $s'_j = s_j \cap Z_{l(S_j)}$ where $l(S_j)$ is the largest index $i$ such that
    $S_j \cap Z_i \neq \emptyset$
5 Compute the maxima for each $s'_j$.
```

However, the rank-sequence algorithm is no better than the trivial algorithm (where we sort all the elements) in the worst case. It was shown in [29] that for some collection of subsets there does not exist any good rank sequence for which both the sequence and the reduced sets can be computed in linear number of comparisons.

Prior to this, Komlos [30] proposed a special algorithm for the set maxima problem when X is a set of edges in a tree and the collection $\mathcal{S}$ consists of subsets of edges that join two non-adjacent vertices in the tree. The original motivation for this special case was to solve the minimum spanning tree (MST) verification problem. (Given an undirected edge-weighted graph G and a candidate tree T, the MST verification problem asks if T is an MST for G. Using the fact that edges in T cannot be the maximum weighted edge on any cycle in G, we immediately see the reduction to the set maxima problem as stated above.) We see that this special case of the set maxima problem sits at the opposite end of the spectrum to the hyperplane maxima problem according to how their subset structures differ. In this problem the size of subsets can vary between 2 to $n-1$ and they have large overlaps (can be $O(n)$) which is in contrast to the hyperplane subset structure, where overlaps are bounded. However, Komlos showed that this special case can be solved in linear time (in fact he showed that only $5n + n\log\frac{|\mathfrak{S}|+n}{n}$ comparisons suffice) on a pointer machine. Since, $|\mathcal{S}| = O(|E|)$ we see that MST verification can be done in $O\left(n\log\frac{|\mathfrak{S}|+n}{n} + m\right)$ comparisons. The main idea behind Komlos' algorithm is to convert T into a rooted directed tree by picking any node and making it the root. Any two non-adjacent node in the tree are joined via two directed half-paths. It was shown how the maxima of these half paths can be computed efficiently and maintained such that the maxima for any path in the tree (determined by two nodes)can be answered quickly.

Liberatore [31] showed that both of the above cases can be generalized using weighted binary matroids. First we have to introduce some basic notions related to matroids. There are several ways to define matroids but the definition with independent sets will suffice.

Definition 5.2 (Matroid)**.** A matroid $\mathfrak{M}$ is a pair $(X, \mathfrak{I})$ of a set X and a collection of subsets $\mathfrak{I}$ of X such that the following holds:

1. $\emptyset \in \mathfrak{I}$
2. If $A \subset B$ and $B \in \mathfrak{I}$ then $A \in \mathfrak{I}$.
3. If $A, B \in \mathfrak{I}$ and $|A| < |B|$ then $\exists e \in B$ such that $A \cup e \in \mathfrak{I}$.

The sets in $\mathfrak{I}$ are called *independent* sets. A set that is not independent is called dependent. An independent set is maximal if adding any more element to it makes it a dependent set and the set of all maximal independent sets is called the set of bases of $\mathfrak{M}$ and is denoted by $\mathfrak{B}_{\mathfrak{M}}$. Similarly we can define minimal dependent sets called circuits or cycles of $\mathfrak{M}$. The rank function $\mathscr{r} : 2^X \to [|X| - 1]$ is defined for any subset A of X as the cardinality of the largest independent set contained in A. Hence $\mathscr{r}(X)$ is simply the size of a basis in $\mathfrak{M}$. Subsets of X that have a rank one less than X are called hyperplanes of $\mathfrak{M}$. A weighted matroid is any matroid with an weight function on its elements. Given a basis $B \in \mathfrak{B}_{\mathfrak{M}}$ and an element $e \in E \setminus B$, there is an unique circuit $C_B(e)$ that is contained in $B \cup e$, called the *fundamental circuit* of e with respect to B.

Let $P_B(e) = C_B(e) \setminus e$ be the *fundamental path* of e with respect to B. Let A be a matrix of size $m \times n$ over some field GF_q[1]. Columns of A can be thought of as vectors in GF_k^m. The collection of independent column vectors forms the independent sets of a matroid $\mathfrak{M}_A$. Such a matroid is said to be *represented* by A. If some matroid has a representation matrix over GF_2 then it is said to be a binary matroid. Moreover if A is a totally unimodular matrix then $\mathfrak{M}_A$ is said to be *regular*. An *unimodular* matrix is an integer matrix whose determinant is either 1 or -1. A matrix is totally unimodular is every non-singular submatrix is also unimodular. The dual of a matroid $\mathfrak{M}$, denoted by $\mathfrak{M}^*$ is the matroid which has the same element set as $\mathfrak{M}$ and its bases are exactly the complements of the bases of $\mathfrak{M}$.

One of the canonical examples of matroids is the *graphic matroid*. Given a graph $G(V, E)$, the *graphic matroid* $\mathfrak{M}_G$ has element set E and any subset of edges in E that induces a tree in G is an independent set of $\mathfrak{M}_G$. Hence the MST verification problem

[1]Where GF_q is the Galois field of order q.

reduces to finding $\max P_T(e)$ for every non adjacent edge of the candidate tree T. Generalized to binary matroids (since graphic matroids are also regular) this has been termed by Liberatore as the *fundamental path maxima* (`FPM`) problem over such matroids. A *cographic matroid* is a dual of a graphic matroid. For a cographic matroid the `FPM` can be solved in $O((m+n)\log^* n)$ [32]. Liberatore generalized these results to a restricted class of matroids that can be constructed via direct-sums, 2-sums and 3-sums [2] and gave a $O(\min((m+n)\log^*(m+n), n\log n))$ deterministic algorithm [31]. However, this kind of decomposition is only possible for regular binary matroids and its not clear how to extend them to a larger class of binary matroids. Note that a regular matroid can be represented over all fields, thus also over GF_2 as well. Hence it is a subset of binary matroids.

5.1.1 Our Result

We see that only for a few special cases of `SetMaxima` does a non-trivial result exist. This motivated us to search for a new non-trivial set system for which the set-maxima is linear time solvable. In this context we look at a geometric setting [3].

In our formulation elements of X are points on a plane and the sets are convex polygons with a bounded number of sides. Additionally, each point has a weight associated with them. We show that the set-maxima problem for this set-system can be solved with $O(n+m)$ number of comparisons.

5.2 A Generic Formulation

We first create a general framework for solving set-maxima based on the structure of the set-intersection lattice. This will later help us when solving for the convex set system. One drawback of this generic formulation is that it does not account for transitivity; thus it is oblivious. So the set of comparisons determined by the intersection lattice is only dependent

[2]Properly defining this class requires a lot of technical terminology which we did not cover here.

[3]The earlier formulation by [28] for the projective geometry case is only based on the t-design structure of the set-system and does not use any geometric arguments.

on the set system and not on the results of prior comparisons. However, for the geometric case we can still derive a non-trivial bound with only the knowledge of the subset structure.

Let the sets in $\mathfrak{S}$ be labeled from 1 to m and S_i be the set labeled i, where each S_i is a subset of X. $\mathfrak{S}$ can be thought of as a hypergraph on vertex set X. We define a modified set intersection lattice $\mathfrak{L}$ for the collection $\mathfrak{S}$. Each node of the lattice is a subset of $[1 \dots m]$ (index sets). For the index set I, let ϕ_I be a node in the lattice. Recall that normally the set-intersection lattice has m layers and the top layer represents the empty set, which is discarded in $\mathfrak{L}$. The k^{th} layer contains sets $I \subset [1 \dots m]$ whose cardinality is k. The bottom layer corresponds to $[1 \dots m]$. We treat the topmost layer in a special manner. Since, the index sets for this layer are just singletons, we directly use the sets in $\mathfrak{S}$ to represent the corresponding nodes. That is we use the set S_i to denote the node $\phi_{\{i\}}$.

Each node ϕ_I stores some subset of X (possibly empty) and they form a disjoint partition of X. An element x which is stored in ϕ_I has the following two properties:
1) $x \in \bigcap_{i \in I} S_i$
2) For all $J \supset I$, $x \notin \bigcap_{i \in J} S_i$.
Since ϕ_I's form a disjoint partition of X, number of non-empty ϕ_I's is bounded by n. Although the size of $\mathfrak{L}$ could possibly be exponential in n, henceforth we will only be using the linear number of nodes that have a non-empty ϕ_I. As mentioned earlier we treat the nodes $\phi_{\{i\}}$'s differently and keep them in $\mathfrak{L}$ even if they are empty. So we can compute and maintain the pruned $\mathfrak{L}'$ in $O(m+n)$ space and $O(p)$ time respectively without making any key comparisons. Further we can compute *parent* relations in the new pruned lattice; if a path from node ϕ_I down to ϕ_J exists in the original lattice (so $I \subset J$) and now only nodes ϕ_I and ϕ_J from that path exist then node ϕ_I is the parent of ϕ_J. $\mathfrak{L}'$ consists of all $\phi_{\{i\}}$'s and any other ϕ_I's which are non-empty. Let $\mathfrak{I}$ be the collection of all index sets of $\mathfrak{L}'$. The following proposition shows that we can further assume that each ϕ_I's has exactly one element.

Proposition 1. If any ϕ_I has more than one element then we can reduce the set maxima problem to an equivalent one using only $O(n)$ number of extra comparisons, such that each ϕ_I has at most 1 element.

Proof. Since ϕ_I's form a disjoint partition of X we can compute the maxima of every ϕ_I, if it contains more than one element, using a total of $O(n)$ number of comparisons. □

The *cover* C_I for each non-empty ϕ_I is defined as the set of all the parents of ϕ_I in the pruned lattice $\mathfrak{L}'$. A *cover-set* $\zeta_I \subseteq C_I$ of ϕ_I contains a collection of parents of ϕ_I such that the following holds:

$$I \subseteq \bigcup_{\phi_J \in \zeta_I} J$$

Let ζ_I^* be a cover-set of ϕ_I of minimum size.

Theorem 5.1. We can solve set maxima with $O(\sum_{I \in \mathfrak{I}} |\zeta_I^*|)$ comparisons.

Proof. We start at the bottom and work our way up the lattice $\mathfrak{L}'$, For each node ϕ_I we compare the element in ϕ_I with the element in ϕ_J for each $\phi_J \in \zeta_I^*$; the larger element is retained in ϕ_J. If ϕ_J is empty then no comparison is performed and we simply "copy" element into ϕ_J. If two or more nodes are covered by one parent node then, in turn, several compare-and-replace operations will be done on the parent node. It is clear by an inductive argument that each node ϕ_I will learn (via information passed up from its children) the true maximum of the intersection it represents; hence each node $\phi_{\{i\}}$ corresponding to a sets of $\mathfrak{S}$ will be solved. Clearly each element is involved in $\leq |\zeta_I^*|$ comparisons, which proves the theorem. □

5.3 Convex Set-System

Elements of X are points on a plane; associated with X is a key value (the "element" in the original formulation). The sets of $\mathfrak{S}$ are now constrained to be convex polygons. The points that are in S_i are specified as all the points inside a given convex polygon P_i. We make the following restrictions: 1) The number of sides of each polygon is bounded by a constant, k,

2) no points lie on an edge or on a vertex of a polygon. (In fact without these restriction on the polygons, it is possible to represent any arbitrary set system in this geometric setting. This follows easily. Take all the points of X to be on a circle. Then any subset of points are the corners of a convex polygon.) Hence, the above restrictions allow the geometry to play a role.

We present our result in terms of the parameter k. Specifically, we show that we can determine all the maximum for every convex polygons using only $O(m+n)$ total key comparisons. Note that we do not assume any bound on the number of points that can be inside a convex polygon. The algorithmic framework is the same as the framework of section 5.2. Again we will use a lattice but with convex polygons associated with each node. The required nodes (in the pruned lattice) will be associated with each P_i and its convex hull. The other nodes correspond to various non-empty intersections of polygons. Let $Q_I = \bigcap_{j \in I} P_j$. All polygons below will be convex. Let ϕ_I be the set of points from X in Q_I not found in a Q_J where $I \subset J$.

As before a node of the pruned lattice is either (1) one that correspond to a P_i, (2) or contains at least one point from X. We define a cover-set for the convex regions analogous to the general case. If $\zeta_I = \{Q_{I_1}, Q_{I_2}, \ldots, Q_{I_q}\}$ is a cover-set for polygon R then: 1) $Q_{I_i} \supset R$ for all i, 2) the region $Q_i \cap Q_j \setminus R$ is empty (has no points) for all i and j, 3) $I \subset \bigcup_i I_i$ and 4) q is minimum among all such sets.

The rest of the algorithm is analogous to the non-geometric case. The number of nodes in the pruned lattice is linear $O(n+m)$. Again we reduce each ϕ_I to at most one point. It is still true the total number of comparisons depends on the sum of the size of the cover-sets, for the same reasons. The only thing that remains is to bound the quantity $\sum_{I \in \mathfrak{I}} |\zeta_I|$.

Assume $R \subset P$. Let $c_{P,R}$ be a polygonal chain of successive edges of R which are not (part of) some edge of P. In Figure 5.1 we see two such polygonal chains. An upper chain $c_{P,R_1} = (e_1, e_2)$ and a lower chain $c_{P,R_2} = (e_4, e_5, e_6, e_7)$. Let $C_{P,R}$ be the set of all such chains formed by the intersection of P and R. Note that $e_8 \subset f_1$ and $e_3 \subset f_3$ are not

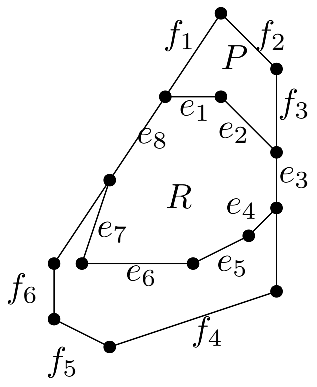


Figure 5.1: Polygonal chains.

part of any chain. If we treat a chain as a set consisting of edges, then we can perform set operations on two of such chains.

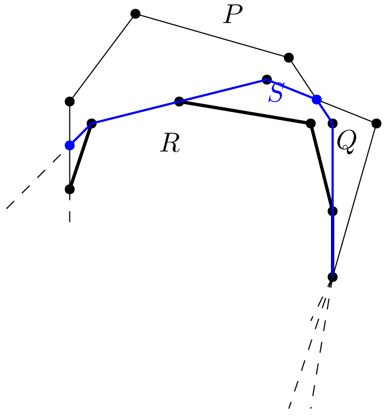


Figure 5.2: A case where P and Q are not covers of R.

Observation 5.2. If P, Q are in the cover-set of R then for any two chains $c \in C_{P,R}$ and $c' \in C_{Q,R}$, $c \cap c' = \emptyset$. If P is in the cover-set of R then $|C_{P,R}| > 0$ since $R \subset P$.

Proof. Let there be some $c \in C_{P,R}$ and $c' \in C_{Q,R}$ such that $c \cap c' \neq \emptyset$. Then from Figure 5.2 we see that there is a convex polygonal region S such that $R \subset S \subseteq P \cap Q$. This contradicts our earlier observation that a pair of sets in a cover-set cannot intersect beyond the region that they are covers of. $\square$

For the next lemma we need an additional combinatorial fact.

Observation 5.3. If T is a set of l elements and $\mathcal{S}$ is a collection of subsets from T, each of which has size $\geq l-k+1$, then every set in $\mathcal{S}$ contains some element from any subset of size $\geq k$ of T.

Proof. We can prove this by contradiction. Suppose there is a subset R of T with $|R| \geq k$ that violates the above claim. Let $S \in \mathcal{S}$ such that $R \cap S = \emptyset$. But S has at least $l-k+1$ and R has at least k elements then $R \cup S \supsetneq T$. But by definition $R \cup S \subseteq T$, which leads to a contradiction. $\square$

Lemma 5. If the set system $\mathfrak{S}$ consists of polygons P_i of at most k sides then every polygon in the lattice has a cover-set of size at most k.

Proof. Let Q_I be any (l-gonal) region formed by the intersection of r polygons, $|I| = r$. Let these polygons be $\{P_1, \ldots, P_r\}$. For each polygon P_i, at least $l-k+1$ of the sides of Q_I will be part of some polygonal chains (i.e., in C_{P_i,Q_I}) of P_i. This can be seen since P_i has at most k sides and is convex, only $k-1$ edges of Q can also be part of the edges of P_i. Let C_i be the collection of all edges of Q_I that are in some chain of P_i. We can use Observation 5.3 to claim that there is a set T of at most k edges of Q_I such that every polygon P_i has at least one edge in one of its chain from the set R. For each edge e of Q which is in T let I_e be the index set corresponding to the collection of polygons which contains the edge in some chain. Let $Q_{I_e} = \bigcap_{i \in I_e} P_i$. Clearly $Q_I \subset Q_{I_e}$. If, for each pair of edges e, f of T, $Q_{I_e} \cap Q_{I_f} = Q_I$ then we are done. Otherwise we have for some pair $Q_{I_e} \cap Q_{I_f} = Q_{I_e \cup I_f} \supset Q_I$. Then we simply replace the two regions Q_{I_e} and Q_{I_f} with $Q_{I_e \cup I_f}$, which can only reduce the size of our cover-set. Since $I = \bigcup_{e \in R} I_e$ we see that the collection $\{Q_{I_e}\}$ or a smaller collection formed through intersections of sets from $\{Q_{I_e}\}$forms a cover-set of Q_I and has size at most k.

$\square$

Theorem 5.4. On a set-system realized by convex polygons of bounded number of sides (as above) with the elements representing points in the plane, we can solve the set maxima with $O(n+m)$ comparisons.

Proof. Proof immediately follows from Theorem 5.1 and Lemma 5. $\square$

Chapter 6: Concluding Remarks

In this part we looked at two comparisons-based problems; restricted sorting and set-maxima. We gave parameterized algorithms for restricted sorting and solved a special case for the set-maxima problem. However, both of these problems have so far turned out to be hard to solve in full generality. In this chapter we discuss some of the difficulties surrounding them and why any progress on the general front could be significant to our understanding of comparisons-based problems.

6.1 Restricted Sorting

Recall that the deterministic algorithm presented in Chapter 3 had the complexity of $O((q+n)\log n)$, where q was the number of forbidden pairs. The algorithm presented in Chapter 3 works with or without a total order guarantee. However, for the former case, we do not have a matching lower bound. The trivial bound is $\Omega(n \log n)$ which follows from case when there are no forbidden pairs. Now, if we make $q = O(n)$ we see that it matches this lower bound. On the other hand if $n - q = O(n \log n)$ then we can simply check every edge, making $O(n \log n)$ comparisons. We see that if the graph is either very dense or reasonably sparse we get a tight result. The subtle point here is that these two cases are easy for completely different reasons. In the case when the graph is dense, it is easy because we can infer relations between many pairs using transitivity. This can be observed form the fact that our original algorithm finds large cliques which can be sorted optimally and uses them to partition other elements. In the case when the graph is sparse in the above sense, we do not know a way to use transitivity and only saved by the fact that checking all the $O(n \log n)$ edges is not very costly. We believe, in order to solve the general case we need to understand the case when the graph is sparse.

The situation is complicated by the fact that with total order guarantee the input graph G determines a set of "forbidden orders" which reduces the set of output orderings and makes the comparison tree bound smaller. Let us discuss this with an example. Consider the input set X having three elements $\{u, v, w\}$ and G be the path uvw. Clearly, a forbidden pair (here (u, w)) cannot be consecutive elements in any of the ordering of X. Otherwise, we cannot sort all the elements. In the above case there are only two permissible linear extensions: (u, v, w) and (w, v, u). Thus G forbids a set of total orders from $\mathfrak{S}$. However, a collection of total orders does not necessarily defines a poset whose linear extensions are exactly those from the set. This follows from the fact that there are many more collections of total orders $(O(2^{n!}))$ than there are labeled posets $(O(2^{\frac{n^2}{4}}))$[10]. For example, in the above the two permissible orders (u, v, w) and (w, v, u) produce an anti-chain. However, the set of linear extensions of an anti-chain consists of all possible orderings of u, v, w. To determine the number of permissible orders we have to determine the number of Hamiltonian paths in G, which is a hard problem. In this context we can generalize the balancing pairs conjecture as follows.

Conjecture 1. [(ϵ, δ)-balancing pairs] For some $0 < \delta < 1$ we can always find a non-empty set of unoriented edges in G of size at most ϵn such that orienting them reduces the number of possible directed Hamiltonian paths in G by at least δ.

The classical balancing pairs conjecture [1] is the special case when G is the complete graph and $\epsilon = 1/n$.

Another important problem is to determine the complexity of finding the median, under the total order guarantee. Recall that we used an approximate median to partition the sets in our deterministic algorithm. Although the median element may not be comparable to all elements, a faster way to find it may still yield a faster sorting algorithm. However we believe that under the restricted comparison model finding the median may be as hard as sorting.

[1]Original conjecture which supports the existence of a $\delta = 1/3$, still remain unsolved. However as stated previously the current best known value of δ is $3/11$.

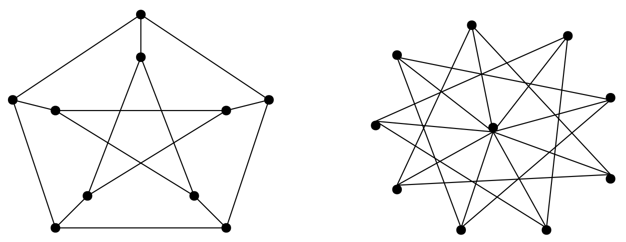


Figure 6.1: For the Peterson's graph (left) there are 10 different Hamiltonian paths (20 total orders). However the starfish graph has none, as it is not Hamiltonian.

Another area to look at is to instead of using the number of forbidden edges, we work with some other parameters of the graph G. For example we can study graphs which satisfy certain local and global connectivity constraints and develop specialized results for such graphs. Figure 6.1 shows an example that contradicts the intuition that having more edges necessarily yields more acyclic orientations with directed Hamiltonian paths.

6.2 Set-Maxima

In this section we discuss the apparently hard input instances for the set-maxima problem. Recall the two instances for which we have an optimal deterministic algorithms. In the first instance the set-system is determined by paths from a vertex weighted tree. In this case we have many sets that overlaps with each other in the form of subpaths of larger paths. We see from Komlos' algorithm [30] that we can use the overlapping nature of this set-system in our favor by using the monotonicity property of the maximum of each of the subsets. That is, for two subset S_1, S_2 with $S_1 \subset S_2$ we have $\max S_1 \leq \max S_2$. Another interesting observation we can make about the Komlos' algorithm is that it processes order information in a local manner as we descend downwards in the tree. In contrast the set system for the projective geometry problem has a bounded overlap. Specifically for $PG(d, q)$ we know that the d hyperplanes intersect at a single point. However as we saw earlier we can also use the bounded overlap property to work for us by noting that such a set-system will have an expanding property. Informally, each set is spread almost uniformly if we look at the bipartite representation of the set-system, where left vertices are elements and right vertices are sets. The edges are determined by set-inclusion. This helps us devise a global strategy by focusing on determining elements of lower ranks (higher value). Similar to the restricted sorting problem we do not know how to solve set-maxima optimally for the intermediate case. That is, the subsets may have moderate overlap and may even have expansion properties but not enough to make it a highly expanding expander. For example the technique used in [28] fails to work when the expansion is not highly expanding.

6.2.1 Local Sorting

An interesting special case of set maxima is the local sorting problem. In this problem each set contains exactly two elements, hence the set system can be represented as a graph. The goal is to determine the orientation of every edge. This looks very similar to the restricted sorting problem. However there is a crucial difference. In this problem we are allowed to compare any pair, regardless of whether they form an edge. However, we only are required to output the relations between adjacent vertices. Naturally this problem has a local flavor and hence the name. This problem also remains open when the graph G is arbitrary. It has the same comparison tree lower bound as the set maxima problem, which is determined by the number of acyclic orientation of G. Although we were not able to come up with an algorithm for local sorting we found a graph which we believe serves as a difficult instance for this problem.

Observe that for any d-regular graph the logarithm of the number of acyclic orientations is $O(n \log d)$, from the formula in Chapter 5. Now consider an easy case, where G is the union of n/d cliques with d vertices in each (assuming n is divisible by d). In this case we can simple sort every clique with $O(d \log d)$ comparisons for a total of $(n \log d)$ comparisons. Thus it seems we should consider graphs that do note have cliques or other dense subgraphs. One good example of such a graph is the hypercube. A hypercube graph with $n = 2^d$ vertices is d-regular. However, it is also bipartite and has expanding properties. Thus every induced subgraph is sparse (has an edge density of d) and determining many relations using transitivity seems difficult in this case. Hence we believe that solving the local sorting for the hypercube graph will give us better insight for the the general case.

Part II

Part II: Problems On Graph Reconfigurations

Chapter 7: Reconfiguration Problems

In the most general sense combinatorial reconfiguration problems concerns how we can transform one combinatorial arrangement to another. Combinatorial arrangements could include, for example, geometric arrangements. Such as arrangements of points in a space, triangulation of convex polytopes, simplicial complexes. Finding the shortest flip sequence between two triangulation of a convex polygon is one such example[33]. One of the major area of applied research concerns how to transform molecular or biological structures.

In this part we to look at reconfiguration problems on graphs. Specifically, we will focus on two types of results: one is *computational* and the other will be *structural.* For the result we look at the computational complexity of some special configuration problems. On the other hand structural results gives bounds on "reconfigurability" of graphs in terms of its properties.

We first start with our model of computation, which is different from the first part. Unless otherwise stated throughout this Part we shall use the RAM model for presenting our results. Problems in this half are generally thought to be NP-Hard or worse. The pointer machine model would be too restrictive for our purpose. It will make some memory-lookup type operations too cost-prohibitive, since pointer representation may result in exponential blowup in complexity.

7.1 General Framework

Before introducing graph reconfiguration let us first discuss a relatively general formulation on groups. We start with some definitions. Let $\mathfrak{S}_n$ be the Symmetric Group: set of all permutations over $[1, n]$ and group operations, the compositions of permutations. A *Permutation Group* is any subgroup of $\mathfrak{S}_n$. Let $\mathfrak{H}$ be a permutation group. A subset S of elements is called a generating set of $\mathfrak{H}$ if every element of $\mathfrak{H}$ can be generated by sequence of compositions of these elements. There may be many such generating sets for a group.

Suppose S is the set of all permutations that are *involutions*. A permutation π is an involution if every cycle is either of size 1 or 2. If π is an involution then it is its own inverse, that is for any permutation σ, $\pi(\pi\sigma) = \sigma$. Further it can be shown that we can generate any permutation of $\mathfrak{S}_n$ by composing at most two involutions from S. Thus S is a generating set for $\mathfrak{S}_n$. Next we introduce a classical problem on generators:

Definition 7.1. (Minimum-Generating-Sequence Problem) Let $\mathfrak{H}$ be a permutation group and S a generating set. Given a permutation π we want to determine a sequence of generators from S of minimum length that generates π starting from the identity permutation. We denote the length as $\texttt{MGS}(\mathfrak{H}, S, \pi)$.

If $\pi \notin \mathfrak{H}$ then we take $\texttt{MGS}(\mathfrak{H}, S, \pi) = \infty$.

When the group we are working with is clear then we simply use $\texttt{MGS}(S, \pi)$. $\texttt{MGS}$ was first shown to be NP-hard by Evan and Goldreich [34]. For arbitrary permutation groups Jerrum [35] showed this problem to be PSPACE-Complete. This remains true even when each generator is of order 2. A generator $g \in S$ is of order k if k is the minimum number such that $g^k = e$, where g^k denotes composition of g with itself k times and e is the identity permutation. In the next section we introduce a special case for the symmetric group where the generating set S is determined by matchings in a specified graph.

7.2 Examples of Reconfiguration Problems

Let $G(V, E)$ be a simple connected labeled graph with n vertices. Vertices are labeled from 1 to n. Each vertex contains a pebble initially and no pebbles can share a vertex. Without loss of generality we assume p_i is the pebble on vertex labeled i. A *arrangement* A is a collection of permutations. Starting from the input configuration we move pebbles in such way that we reach a configuration that is in A. That is, the final placement of pebbles specify a permutation that belongs in the arrangement. However, an arrangement can be specified implicitly, and the set of permutations that satisfy the arrangement may not be easily computable.

In each step we can move the pebbles in the following way to get from the starting configuration to a final configuration that satisfy the arrangement We pick a matching $M \subset E$ (matching is a set of mutually vertex disjoint edges) and we swap the pebbles across matched edges. Given a reconfiguration problem (G, A) the minimum length sequence of matchings that can reach a satisfying arrangement is the *reconfiguration time* of G with respect to A. The reconfiguration time is a general term, we use different names based on the nature of A. Next we look at some interesting problems that arise from different types of arrangements A.

7.2.1 Permutation Routing

This is one of the more natural model of routing on a communication network. For this problem we specify our arrangement A with a single permutation π, that determines the destination vertex for each pebble. This problem was originally introduced by Alon and others [36] and has a rich literature. Some authors have termed it as the Parallel Token Swapping problem. This problem is also a special case of the MGS problem. Given a graph G and a permutation π the minimum number of steps necessary to route all the pebbles are known as the *routing time*, $\mathcal{rt}(G, \pi)$. We will discuss permutation routing in more detail in the next chapter, so we skip it for now. (Figure 7.2 gives an example of routing a permutation on a cube.)

7.2.2 Acquaintance Time

This new property of graph was introduced recently [37]. Given a graph G with each vertex containing a pebble, a pebble must be acquainted with every other pebble. We say two pebbles are acquainted if they shared an edge at some point. The goal is acquaint all pebbles with each other using the minimum number of steps. This is called the acquaintance time of a graph $(\mathcal{ac}(G))$. In [37] authors showed that it is NP-Hardto compute the acquaintance time and for every graph the acquaintance time is bounded by $O(n^{3/2})$. In this problem no final arrangement is specified directly, but implied in the following sense. We can think of the constraint that every pebble must be acquainted in the following way. Let Π_{ij} be the set of all permutation where pebble p_i is adjacent to pebble p_j. Let $\mathfrak{M} = (M_1, M_2, \ldots, M_t)$ be a sequence of matchings that acquaints every pebble at least once. Let $\pi_k = M_k(M_{k-1}(\ldots(e)\ldots))$ be the permutation achieved after $k \leq t$ steps. Then, for every i, j there is a $t(i,j) \leq t$ such that $\pi_{t(i,j)} \in \Pi_{ij}$. (Example of graph with highest acquaintance time is given in Figure 7.1.)

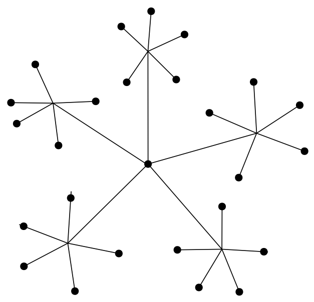


Figure 7.1: The stars of stars has the highest value of acquaintance time $O(n^{3/2})$. The outer and inner stars all have degree $\theta(\sqrt{n})$. We can acquaint a pebble with others by placing it on the center of various stars.

7.2.3 Sorting Permutations

Sorting on graphs using pebbles is different from the sorting algorithms we discussed previously. In our previous framework we used a global memory to store the elements (here these are the pebbles). Further, we choose a series of comparisons that were dependent on outcomes of previous comparisons. When sorting in graph we will do away both of the above features.

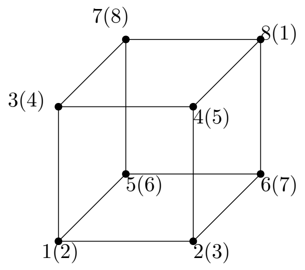
In this problem the initial configuration is unknown. That is we do not know the label of the pebble which is on the vertex labeled i. Additionally, we are given a permutation which determines the final destination of pebbles, as in the case of permutation routing. We still use matchings to move pebbles. However, we need to compare pebbles along matched edges to help guide the pebbles to their destination. One obvious question is then why do not we sort the pebbles first and then use their rank to determine the final permutation? There are mainly two reasons why we don't this. Firstly, reconfiguration model that we discuss here can be realized as a parallel or distributed computing model. Parallelism is exploited via matchings, where we can move many pebbles simultaneously. Reconfiguration problems is essentially the study of how much parallelism one can achieve on a family of graphs for some arrangement constraints. If we had sorted the pebbles initially (say using a global memory) then this local flavor is lost. Secondly, these matchings which we use to compare pebbles and swap based on their labels is oblivious. That is the next matching is not dependent on the results of the comparisons arising out of the previous ones. This adds robustness in the sense that the same sequence of matching can be used to sort every possible input configuration of pebbles. Thus they can be realized using specialized hardwares, which are known as sorting networks. Additionally, as we shall see in later chapters, for many graphs we can efficiently sort pebbles using this parallel model.

7.2.4 Visiting Time

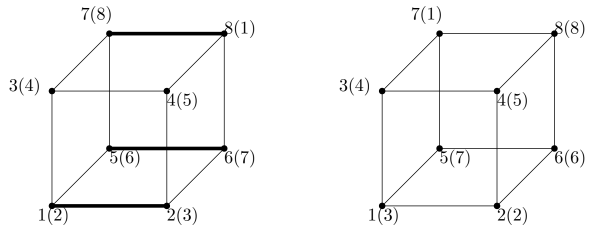
This problem is a relaxation of the permutation routing problem. It also has a flavor of the acquaintance problem, in that the arrangement is implicitly specified. Given a permutation π we want every pebble p_i to visit the node labeled $\pi(i)$ at some point during the routing. However, we relax the condition that they have to be at this node when the routing finishes. The *visiting time* $vt(G, \pi)$ for graph G and a permutation π is defined similarly. It is obvious that for every π, $vt(G, \pi) \leq rt(G, \pi)$. As an example, the visiting time for the permutation given in Figure 7.2 is 3, which is one less than the routing time. This can be done be choosing the following sequence of matching: first match all edges $\{(1, 2), (3, 4), (5, 6), (7, 8)\}$, then $\{(1, 3), (2, 4), (5, 7), (6, 8)\}$ and lastly $\{(1, 5), (2, 6), (3, 7), (4, 8)\}$. Thus the visiting time problem is useful to determining lower bound for routing time.

7.3 Sequential Model

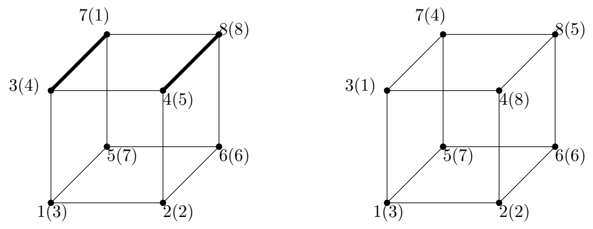
As the name suggests in this model each step consists of choosing a single edge and swapping the pebbles on its incident vertices. In the context of permutation routing and sorting this model has been studied extensively. This model is relevant to our matching model as any upper bound on the number steps for reconfiguration time is also going to be an upper bound on the parallel model. Also it is generally easier to prove result using the sequential model than the parallel model. For example there is 4-approximation algorithm for the routing time in the sequential model[38]. However, we do not have such results in the parallel model. We are going to discuss some literature on routing time for graphs under sequential model in the next chapter.



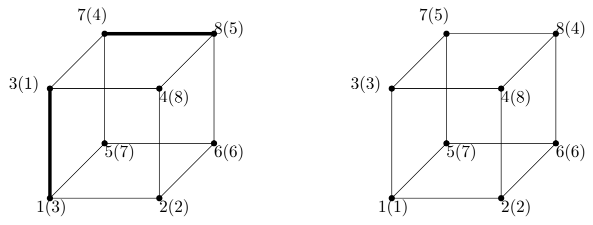
(a) Intial Configuration.



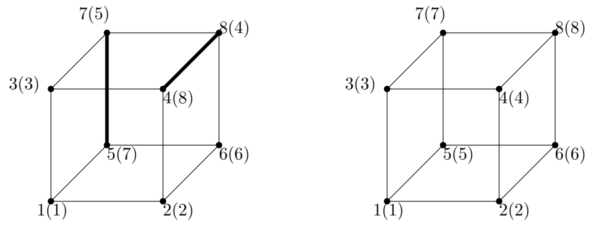
(b) After step 1



(c) After step 2



(d) After step 3



(e) After step 4

Figure 7.2: Routing the permutation $\pi = (12345678)$ on a Cube.

Chapter 8: Hardness Of Permutation Routing

This chapter is about permutation routing on graphs. In particular we explore some computational aspects of the model. As mentioned previously, this model was originally introduced by Alon and others [36] in the early 1990's. We repeat some of the definitions that are specific to this chapter.

Consider a undirected labeled graph $G(V, E)$. Each vertex of V contains a pebble with the same label as the vertex. Pebbles move along edges using a sequence of swaps. A set of swaps (necessarily disjoint) that occurs concurrently is called a step. Such a set is determined by a matching. A permutation π gives the destination of each pebble; that is, the pebble p_i on vertex i is destined for the vertex $\pi(i)$. The task is to route each pebble to their destination via a sequence of matchings. The *routing time* $rt(G, \pi)$ is defined as the minimum number of steps necessary to route all the pebbles for a given permutation. The *routing number* of G, denoted by $rt(G)$, is defined as the maximum routing time over all permutations. Let $m = |E|$ and $|G| = n = |V|$.

Determining the routing time is a special case of the `MGS` problem for groups, as follows. Consider a spanning tree T of G. We can use the edges of T to route pebbles in G. For simplicity, assume we move one pebble at a time to its destination. We can pick a pebble whose destination is a leaf in T; there will always be such a pebble. We move this pebble to its destination using a sequence of swaps (matchings with singleton edges). Once the pebble is at its destination, we recursively solve the routing problem for the tree T' which we get from T after removing that leaf. This entire routing scheme takes $O(n^2)$ steps. But more importantly it shows that as long as G is connected we can route every permutation in G in a finite number of steps. That is, the set of matchings in G, denote by $\mathfrak{M}_G$, forms a generating set for the symmetric group. Hence, the permutation routing problem is a special case of the MGS problem.

The serial version, where swaps takes place one at a time, is also of interest. This has recently garnered interest after its introduction by Yamanaka and others [39]. They have termed it the *token swapping problem.* This problem is also NPC as shown by Miltzow and others [38] in a recent paper. The authors also prove the token swapping problem is hard to approximate within a $(1 + \delta)$ factor, for any $\delta > 0$. They also provide a simple 4-approximation scheme for the problem. A generalization of the token swapping problem (and also the permutation routing problem) is the colored token swapping problem [39, 40]. In this problem the vertices and the tokens are partitioned into equivalence classes (using colors) and the goal is to route all pebbles in such a way that each pebble ends up in some vertex with the same class as the pebble. If each pebble (and vertex) belong to a unique class then this problem reduces to the original token swapping problem. This problem is also proven to be NP-complete by Yamanaka and others [39] when the number of colors is at least 3. The problem is polynomial time solvable for the two-color case.

8.1 Prior Results

Almost all previous literature on this problem focused on determining the routing number for typical graphs. In the introductory paper, Alon and others [36] show that for any connected graph G, $rt(G) \leq 3n$. This was shown by considering a spanning tree of G and using only the edges of the tree to route permutations. Note that this is an order of magnitude better than the trivial algorithm we discussed above. Later Zhang and others [41] improve this upper bound to $3n/2 + O(\log n)$. This was done using a new decomposition called the caterpillar decomposition. This bound is essentially tight as it takes $\lfloor 3(n-1)/2 \rfloor$ steps to route some permutations on the star $K_{1,n-1}$.

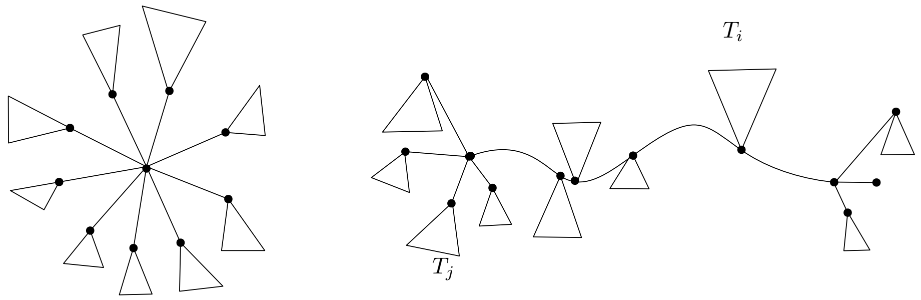


Figure 8.1: A star on the left and a Caterpillar to the right. A tree has γ-type star decomposition if every subtree has $\leq \gamma n$ vertices. In a (α, β)-caterpillar decomposition, every T_j connected to one of the end vertices has size at most βn and sum total of all vertices in the trees connected to the central spine (like T_i) is $\leq \alpha n$. Every tree has a $\frac{1}{3}$-star or a $(\frac{1}{3}, \frac{1}{3})$-caterpillar decomposition.

There are also some known results for routing numbers of graphs besides trees. It is known that for the complete graph and the complete bipartite graph the routing number is 2 and 4 respectively [36]. The result for complete graph was known much earlier as a fact about permutation groups. A possible two step routing scheme given by two involutions is as follows: without loss of generality we can assume the permutation to be routed is just one cycle. Otherwise, we can route each cycle independently on the disjoint complete subgraphs. The two involutions for the cycle $(1234\ldots n)$ are

$$\pi_1 = (1, n)(2, n-1)\ldots(i, m-i+1)\ldots$$

$$\pi_2 = (1, n-1)(2, n-2)\ldots(i, n-i)\ldots$$

The result for the complete bipartite graph is attributed to W. Goddard. Let A and B ($|A| = |B| = m$) be the left and right sets respectively. Then, without loss of generality we can assume the permutation we want to route moves every pebble from A to B and vice versa. Note that if this was not the case then it will take us one extra step. So we have to come up with a 3-step scheme for this special case. A possible 3-step matching strategy via permutations (π_1, π_2, π_3):

$$\pi_1 = (1,2)(3,4)\ldots(2\lfloor\frac{m}{2}\rfloor - 1, 2\lfloor\frac{m}{2}\rfloor)(2\lfloor\frac{m}{2}\rfloor + 2, 2\lfloor\frac{m}{2}\rfloor + 3)\ldots(2m-2, 2m-1),$$

$$\pi_2 = (1,2m)(3,2m-2)\ldots(2\lceil\frac{m}{2}\rceil - 1, 2\lfloor\frac{m}{2}\rfloor + 2),$$

$$\pi_3 = (3,2m)\ldots(2\lfloor\frac{m}{2}\rfloor + 1, 2\lceil\frac{m}{2}\rceil + 1).$$

Li and others [42] extend these results to show $rt(K_{s,t}) = \lfloor 3s/2t \rfloor + O(1)$ $(s \geq t)$. For the n-cube Q_n we know that $n + 1 \leq rt(Q_n) \leq 2n - 2$. The lower bound is quite straightforward. The upper bound was discovered by determining the routing number of the Cartesian product of two graphs [36].

Cartesian product of two graphs are defined as follows. For two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ their Cartesian product is the graph $G_1 \Box G_2$ whose vertex set is $V = V_1 \times V_2$ and there is an edge between $(u_1, v_1), (u_2, v_2) \in V$ iff either $u_1 = u_2$ and $(v_1, v_2) \in E_2$ or $v_1 = v_2$ and $(u_1, u_2) \in E_1$. If $G = G_1 \Box G_2$, then:

$$rt(G) \leq 2\min(rt(G_1), rt(G_2)) + \max(rt(G_1), rt(G_2)). \tag{8.1}$$

Since $Q_n = K_2 \Box Q_{n-1}$ the result follow. Where K_2 is the complete graph with two vertices. The base case, $rt(Q_3)$, was determined to be 4 via a computer search[42].

8.2 Computational Results

Some of the computational results we present in this chapter are summarized below.

1. If G is bi-connected then deciding whether $rt(G, \pi) = k$ for any π and $k > 2$ is NP-complete.

2. For any graph, deciding if $rt(G, \pi) \leq 2$ can be done in polynomial time, for which we give a $O(n^{2.5})$ time algorithm.

3. As a consequence of our NP-completeness proof of the routing time we show that the problem of determining a minimum sized partitioning scheme of a colored graph such that each partition induces a connected subgraph is NP-complete.

4. We introduce a notion of approximate routing called *maximum routability* and give an approximation algorithm for it.

8.3 A $O(n^{2.5})$ Time Algorithm for Deciding when $rt(G, \pi) \leq 2$

In this section we present a polynomial time deterministic algorithm to compute a two step routing scheme if one exists. It is trivial to determine whether $rt(G, \pi) = 1$. Hence, we only consider the case if $rt(G, \pi) > 1$. The basic idea centers around whether we can route the individual cycles of the permutation within 2 steps. Let $\pi = \pi_1 \pi_2 \ldots \pi_k$ consists of k cycles and $\pi_i = (\pi_{i,1} \ldots \pi_{i,a_i})$, where a_i is the number of elements in π_i. A cycle π_i is identified with the vertex set $V_i \subset V$ whose pebbles need to be routed around that cycle. We say a cycle π_i is *self-routable* if it can be routed on the induced subgraph $G[V_i]$ in 2 steps.

If all cycles were self-routable we would be done, so suppose that there is a cycle π_i that needs to match across an edge between it and another cycle π_j. Let $G[V_i, V_j]$ be the induced bipartite subgraph corresponding to the two sets V_i and V_j.

Lemma 6. If π_i is not self-routable and it is routed with an edge from V_i to V_j then π_i and π_j are both routable in 2 steps when all of the edges used are from $G[V_i, V_j]$ and when $|V_i| = |V_j|$.

Proof. We prove this assuming G is a complete graph. Since for any other case the induced subgraph $G[V_i \cup V_j]$ would have fewer edges, hence this is a stronger claim. Let the cycle $\pi_i = (\pi_{i,1}, \ldots, \pi_{i,s}, \ldots, \pi_{i,|V_i|})$. If there is an edge used between the cycles then there must be such an edge in the first step, since pebbles need to cross from one cycle to another and

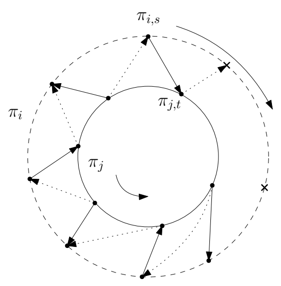


Figure 8.2: The two cycles are shown as concentric circles. The direction of rotation for the outer circle is clockwise and the inner circle is counter-clockwise. Once, we choose $(\pi_{i,s}, \pi_{j,t})$ as the first matched pair, the rest of the matching is forced. Solid arrows indicate matched vertices during the first round. Note that if the cycles are unequal then the crossed vertices in the figure will not be routed.

back. Assume $\pi_{i,s}$ is matched with $\pi_{j,t}$ in the first step. From Figure 8.2 we see that the crossing pattern is forced and unless $|V_i| = |V_j|$, the pattern will fail. □

A pair of cycles π_i, π_j is *mutually-routable* in the case described by Lemma 6. Naively verifying whether a cycle π_i is self-routable, or a pair (π_i, π_j) is mutually-routable takes $O(|V_i|^2)$ and $O((|V_i| + |V_j|)^2)$ time respectively. However, with additional bookkeeping we can compute this in linear time on the size of the induced graphs. This can be done by considering the fact that no edge can belong to more than one routing scheme on $G[V_i]$ or on $G[V_i, V_j]$. Hence the set of edges are partitioned by the collection of 2 step routing schemes. Self-routable schemes, if they exist, are forced by the choice of any edge to be in the first step; no edge is forced by more than four initial choices, leading to a test that runs in time proportional in $|V_i|$. Mutually-routable schemes, if they exist, are one of $|V_i|$ $(= |V_j|)$ possible schemes; each edge votes for a scheme and a scheme is routable if it gets enough votes, leading to a test that runs in time proportional in $|G[V_i, V_j]|$. All the tests can be done in $O(m)$ time.

We define a graph $G_{cycle} = (V_{cycle}, E_{cycle})$ whose vertices are the cycles $(V_{cycle} = \{\pi_i\})$ and two cycle are adjacent iff they are mutually-routable in 2 steps. Additionally, G_{cycle} has self-loops corresponding to vertices which are self-routable cycles. We can modify existing maximum matching algorithm to check whether G_{cycle} has a perfect matching (assuming self-loops) with only a linear overhead. This can be done by creating two copies of the graph and adding an edge between two copies of a vertex with a self loop. Then remove the self loops. The next lemma follows immediately:

Lemma 7. There is a perfect matching in G_{cycle} iff $rt(G,\pi) = 2$.

The graph G_{cycle} can be constructed in $O(m)$ time by determining self and mutual routability of cycles and pair of cycles respectively. Since we have at most k cycles, G_{cycle} has $\leq 2k$ vertices and thus $O(k^2)$ edges. Hence we can determine a maximum matching in G_{cycle} in $O(k^{2.5})$ time [43]. This gives a total runtime of $O(n+m+k^{2.5})$ for our algorithm to find a 2-step routing scheme of a connected graph if one exists.

Corollary 1. $rt(G) = 2$ iff G is a clique.

Proof. ($\Rightarrow$) A two step routing scheme for K_n was given in [36].
($\Leftarrow$) If G is not a clique then there is at least a pair of non-adjacent vertices. Let (i,j) be a non-edge. By Lemma 6 the permutation $(ij)(1)(2)\ldots(n)$ cannot be routed in two steps. $\square$

8.4 Determining $rt(G,\pi) \leq k$ is Hard for Any $k \geq 3$

Theorem 8.1. For $k \geq 3$ computing $rt(G,\pi)$ is NP-Complete.

Proof. Proving the problem is in NP is trivial, we can use a set of matchings as a witness. We give a reduction from 3-SAT. We first define three *atomic* gadgets (see Figure 8.3) which will be use to construct the variable and clause gadgets. Vertices whose pebbles are fixed (1-cycles) are represented as red circles. Otherwise they are represented as black dots. In the first three sub-figures ((a)-(c)) the input permutation is (a,b)[1]. In all our constructions we shall use permutations consisting of only 1 or 2 cycles. Each cycle labeled i will be

[1]We do not write the 1 cycles explicitly as is common.

represented as the pair (a_i, b_i). If the correspondence between a pair is clear from the figure then we shall omit the subscript. It is an easy observation that $rt(P_3, ((a, b))) = rt(P_4, ((a, b))) = rt(H, ((a, b))) = 3$. In the case of the hexagon H we see that in order to route the pebbles within 3 steps we have to use the left or the right path, but we cannot use both paths simultaneously (i.e., a goes along the left path but b goes along the right or vice-versa). Figure 8.3(e) shows a chain of diamonds connecting u to v. Where each diamond has a 2-cycle, top and bottom. If vertex u is used to route any pebble other than 2-cycles to its right then the chain construction forces v to be used in routing the 2-cycles to its left or vice-versa. This chain is called a *diamond-chain* and it is used to propagate a choice (of left or right). In our construction we only use chains of constant length to simplify the presentation of our construction.

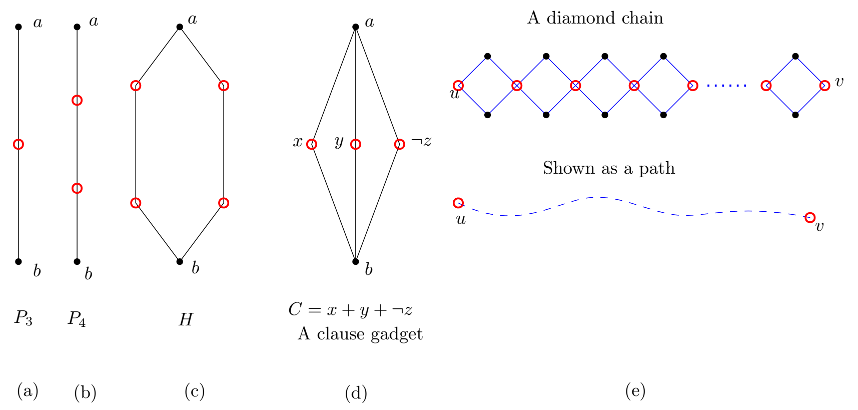


Figure 8.3: Atomic Gadgets, pairs (a, b) need to swap their pebbles. The unmarked red circles have pebbles that are fixed.

Clause Gadget:

Say we have clause $C = x \vee y \vee \neg z$. In Figure 8.3(d) we show how to create a clause gadget. This is referred to as the *clause graph* G_C for the clause C. The graph in Figure 8.3(d) can route $\pi_C = (a_C, b_C)$ in three steps by using one of the three paths between a_C and b_C. Say, a_C is routed to b_C via x. Then it must be the case that vertex x is not used to route

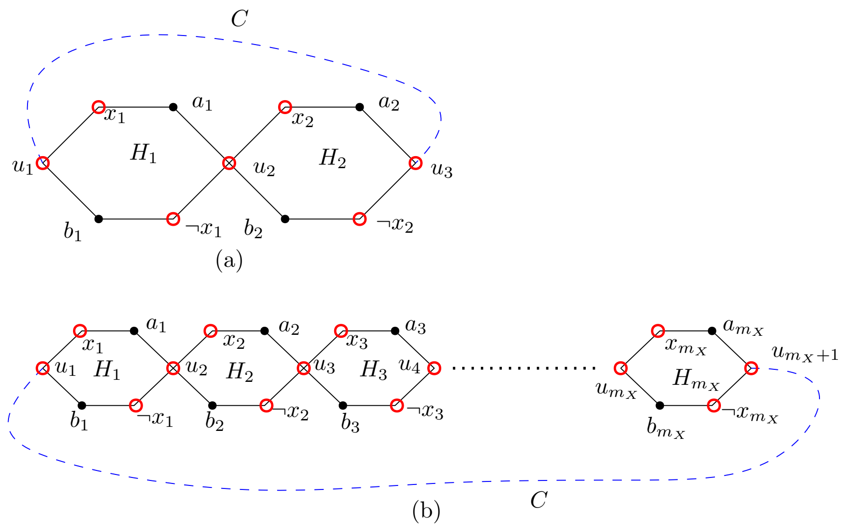


Figure 8.4: Variable graph of X. (a) is a special case for $m_X = 2$, (b) is the general case.

any other pebbles. We say the vertex x is *owned* by the clause. Otherwise, it would be not possible to route a_C to b_C in three steps via x. We can interpret this as follows. A clause has a satisfying assignment iff each clause graph owns a vertex.

Variable Gadget:

Construction of the variable gadgets is done in a similar manner. The variable gadget G_X corresponding to the variable x is shown in Figure 8.4(b). Figure 8.4(a) is essentially a smaller version of 4(b) and is easier to understand. If we choose to route a_1 and b_1 via the top-left path passing through x_1 and u_1 then (a_2, b_2) must be routed via x_2 and u_2. This follows from the fact that since u_1 is occupied the pebbles in the diamond chain C (the dashed line connecting u_1 with u_3) must use u_3 to route the right-most pair. By symmetry, if we choose to route (a_1, b_1) using the bottom-right path (via $\neg x_1$, u_2) then we also have to choose the bottom right path for (a_2, b_2). These two (and only two) possible (optimal) routing schemes can be interpreted as variable assignments. Let G_X be the graph corresponding to the variable X (Figure 3(b)). The top-left routing scheme leaves the vertices $\neg x_1, \neg x_2, \ldots$ free to be used for other purposes since they will not be able take part in routing pebbles in G_X. Thus this can be interpreted as setting the variable X to false. This "free" vertex can be used by a clause (if the clause has that literal) to route its own pebble pair. That is they can become owned vertices of some clause. Similarly, the bottom-right routing scheme can be interpreted as setting X to true. For each variable we shall have a separate graph and a corresponding permutation on its vertices. The permutation we will route on G_X is $\pi_X = (a_1b_1)(a_2b_2)\dots(a_{m_X}, b_{m_X})\pi_{f_X}$. The permutation π_{f_X} corresponds to the diamond chain connecting u_1 with u_{m_X+1}. The size of the graph G_X is determined by m_X, the number of clauses the variable X appears in.

Reduction:

For each clause C, if the literal $x \in C$ then we connect $x_i \in G_X$ (for some i) to the vertex labeled $x \in G_C$ via a diamond chain. If $\neg x \in C$ then we connect it with $\neg x_i$ via a diamond chain. This is our final graph G_ϕ corresponding to an instance ϕ of a 3-`SAT` formula. The input permutation is $\pi = \pi_X \dots \pi_C \dots \pi_f \dots$, which is the concatenation of all the individual permutations on the variable graphs, clause graphs and the diamond chains. This completes our construction. Figure 8.5 gives sketch of the final graph G_ϕ. We need to show, $rt(G_\phi, \pi) = 3$ iff ϕ is satisfiable. Suppose ϕ is satisfiable. Then for each variable X, if the literal x is true then we use bottom-right routing on G_X, otherwise we use top-left routing. This ensures in each clause graph there will be at least one owned vertex. Now suppose $rt(G_\phi, \pi) = 3$. Then each clause graph has at least one owned vertex. If x is a free vertex in some clause graph then $\neg x$ is not a free vertex in any of the other clause graphs, otherwise variable graph G_X will not be able route its own permutation in 3 steps. Hence the set of free vertices will be a satisfying assignment for ϕ. It is an easy observation that the number of vertices in G_ϕ is polynomially bounded in n, m; the number of variables and clauses in ϕ respectively and that G_ϕ can be explicitly constructed in polynomial time. $\square$

Corollary 2. Computing $rt(G, \pi)$ remains hard even when G is restricted to being 2-connected.

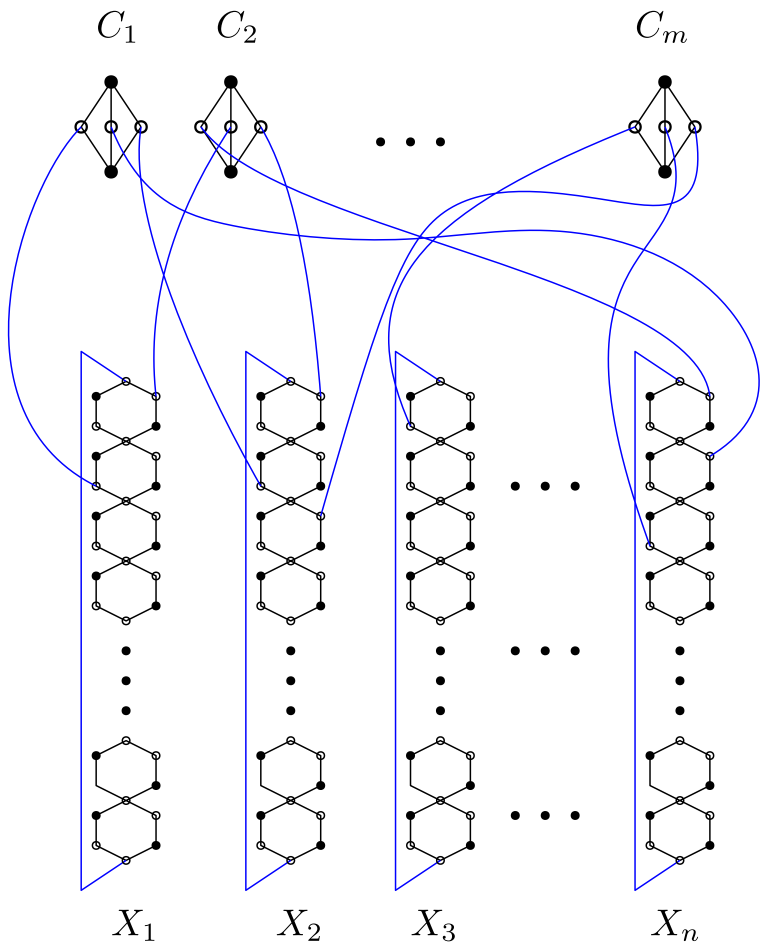


Figure 8.5: The graph G_ϕ.

Proof. This easily follows from the fact that the graph G_ϕ has minimum degree of 2 and hence it is 2-connected. □

8.5 Connected Colored Partition Problem (CCPP)

In this section we go slightly off track to show that the gadgets used to prove permutation routing is hard can be used to prove a hardness result of a partitioning problem on graphs; this demonstrates robustness of our construction Let G be a graph whose vertices are colored with k colors. We say a partition $\mathfrak{S} = \{S_1, \ldots, S_r\}$ of the the vertex set V respects the coloring C (where $C : V \rightarrow \{1, \ldots, k\}$) if each set of vertices of a particular color is contained within a single block of the partition (necessarily $r \leq k$). Note that a block may contain many colors. Further, we require the induced subgraph $G[S_i]$ be connected, for every i. Figure 8.6 gives an example of a CCPP instance.

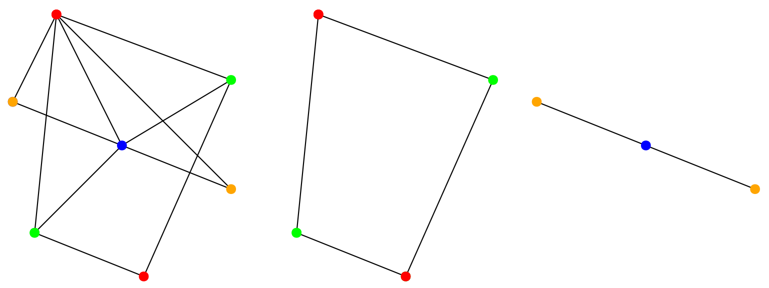


Figure 8.6: An example of a CCPP instance.

Given a graph G, a coloring C (with k colors) and a integer $t \leq n$ the decision version of the problem asks, whether there exists a valid partitioning whose largest block has size at most t. We denote this problem by CCPP(G, k, t). If we replace the requirement of connectedness of the induced subgraphs with other efficiently verifiable properties then it is a strict generalization of the better known monochromatic partitioning problems on colored graphs (see for example [44]). Note that the connectivity requirement on the induced subgraphs is what makes this problem graphical. In fact without it the problem becomes trivial; as one can simply partition the vertices into monochromatic sets, which is the best possible outcome. CCPP(G, k, t) is in P if t is constant, since one can simply enumerate all partitions and there are $O(k^t)$ of them.

Theorem 8.2. CCPP(G, k, t) is NP-complete for arbitrary k and t.

Proof. The proof essentially uses a similar set of gadgets as used in the proof of Theorem 8.1. The idea is to interpret a route as a connected partition. We show that even when each

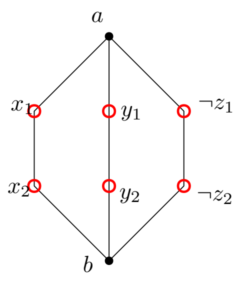


Figure 8.7: A modified clause gadget (from the proof of Theorem 7.1) for the clause $C = x \vee y \vee \neg z$.

color class is restricted to at most two vertices the problem remain NPC . This is done via a reduction from the 3-SAT problem. We reuse some of the gadgets from the proof of Theorem 8.1, but we interpret them differently. Lets discuss the clause gadgets first. In Figure 8.7 we see the graph corresponding to the clause $C = x \vee y \vee \neg z$. The vertices a and b have the same color, lets say c so that they identify with the clause C. All the other vertices (shown as red circles) have unique colors which differ from c. Clearly any valid partition of C must include both pair of vertices along some line. Hence in an optimal partition of the clause graph the largest subgraph is of size 4.

The variable gadgets are same as before (see Figure 8.4) except we do away with the diamond chain and fuse the two end vertices together. So for example in Figure 8.4(a), $u_1 = u_3$. We also make the gadgets twice as long, so instead of having m_X hexagons we now have $2m_X$ of them. The vertices (a_i, b_i) have the same color, which is distinct from every other color. As with the clause graphs the red vertices in the variable gadgets all have unique colors.

To construct the graph G_ϕ corresponding to a boolean formula ϕ we do the following. Since we are not using the diamond chains anymore we directly fuse vertices from clause graphs and variable graphs. So for example if the clause C has variable x as a true literal we fuse two vertices x_i and x_j (for some $i, j \leq 2m_X$) to the two vertices x_1 and x_2 in the clause graph. That is $x_1, x_2 \in \{x_i, x_j\}$. In every partition the pair a_i, b_i must be included since they have the same color. Since each partition must be connected this can happen if either we take the segment (a_i, x_i, u_i, b_i) or the segment $(a_i, u_{i+1 \bmod 2m_X}, \neg x_{i+1 \bmod 2m_x}, b_i)$. Call them top-left and bottom-right segments. Clearly, if we take the top-left segment as part of some partition for any pair (a_i, b_i) we have to use the corresponding top-left segments for all other (a, b) pairs in the same variable graph. Otherwise, the segments will not be connected. The same is true with the bottom-right segments. This forces variable assignments. If we choose the top-left segments then the bottom-right vertices ($\neg x_i$'s) which corresponds to the negated literals will be free and a pair of them can be use to partition a clause graph which contains those literal vertices. If G_ϕ has a partitioning scheme such that every partition is of size 4 then ϕ is satisfiable. We can look at the partition to determine which segments were chosen from the variable graph which determines the variable assignment. Since every clause graph has been partitioned into components of size at most 4, we conclude that every clause is satisfied. The other direction can be proven in a similar manner. $\square$

8.6 Routing As Best You Can

It is often desirable to determine how many packets we can send to their destination within a certain number of steps. Consider the problem of propagating information on social media. In the context of permutation routing this leads to a notion of *maximum routability.* Given two permutation π and σ let $|\pi - \sigma|$ denote the number of fixed points in τ, such that $\tau\pi = \sigma$. We define *maximum routability* $\mathcal{mr}(G, \pi, k)$ as follows:

$$\mathcal{mr}(G, \pi, k) = \max_{\sigma \in S_n,\ \mathcal{rt}(G,\sigma) \leq k} |\pi - \sigma|$$

We denote by `MaxRoute` the problem of computing the maximum routability. Essentially, σ is one permutation, out of all permutations, that can be routed in $\leq k$ steps and that has the maximum number of elements in their correct position as given by π. The permutation σ may not be unique. It can be easily shown (as a corollary to Theorem 8.1) that the decision version of this problem is NP-Hard, since we can determine $\mathcal{rt}(G, \pi)$ by asking whether $\mathcal{mr}(G, \pi, k) = n$. (Of course $\mathcal{rt}(G, \pi) = O(n)$ for any graph, hence with $O(\log n)$ number of different choices of k we can to compute $\mathcal{rt}(G, \pi)$ exactly.)

In this section we give an approximation algorithm for computing the maximum routability when the input graph G satisfies the following restriction. If the maximum degree of G is Δ such that $(\Delta + 1)^k = O(\log^2 n)$ then $\mathcal{mr}(G, \pi, k)$ can be approximated within a factor of $O(n \log\log n/ \log n)$ from the optimal. Unfortunately a good approximation for $\mathcal{rt}(G, \pi)$ does not lead to a good approximation ratio when computing $\mathcal{mr}(G, \pi, k)$ for any $k > 2$. The reason being that with an optimal algorithm for routing π on G it is conceivable that all pebbles are displaced at the penultimate stage and the last matching fixes all the displaced pebbles.

Our approximation algorithm is based on a reduction to the `MaxClique` problem. The `MaxClique` problem has been extensively studied. In fact it is one of the defining problems for PCP-type systems of probabilistic verifiers [45]. It has been shown that `MaxClique` can not be approximated within a $n^{1-o(n)}$ factor of the optimal [46]. The best known upper bound for the approximation ratio is by Feige [47] of $O(n(\log\log n/ \log^3 n))$ which improves upon Boppana and Halldorsson's [48] result of $O(n/ \log^2 n)$. Note that if there is a $f(n)$-approximation for `MaxClique` then whenever the clique number of the graph is $\omega(f(n))$, the approximation algorithm returns a non-trivial clique (not a singleton vertex).

Theorem 8.3. Given a graph G whose maximum degree is Δ, in polynomial time we can construct another graph G_{clique}, with $|G_{clique}| = O(n(\Delta + 1)^k)$, such that if the clique number of G_{clique} is κ then $\mathcal{mr}(G, \pi, k) = \kappa$.

In the above theorem the graph G_{clique} will be an n-partite graph. Hence $\kappa \leq n = O(|G_{clique}|/(\Delta+1)^k)$. As long as we have $(\Delta+1)^k = O(\log^2 n)$ we can use the approximation algorithm for `MaxClique` to get a non-trivial approximation ratio of $O(n \log\log n/ \log n)$.

Proof. Here we give the reduction from MaxRoute to MaxClique. First we augment G by adding self-loops. Let this new graph be G'. Hence we can make every matching in G'

perfect by assuming each unmatched vertex is matched to itself. Observe that any routing scheme on G' induces a collection of walks for each pebble. This collection of walks are constrained as follows. Let W_i and W_j corresponds to walks of pebbles starting at vertices i and j respectively. Let $W_i[t]$ be the position of the pebble at time step t. They must satisfy the following two conditions: 1) $W_i[t] \neq W_j[t]$ for all $t \geq 0$. 2) $W_i[t+1] = W_j[t]$ iff $W_i[t] = W_j[t+1]$. Now consider two arbitrary walks in G'. We call them compatible iff they satisfy the above two conditions. We can check if two walks are compatible in linear time.

Let $\mathscr{W}_i$ be the collection of all possible length k walks starting from i and ending at $\pi(i)$. Note that $|\mathscr{W}_i| = O((\Delta+1)^k)$. For each $w \in \mathscr{W}_i$ we create a vertex in G_{clique}. Two vertices u, v in G_{clique} are adjacent if they do not come from the same collection ($u \in \mathscr{W}_i$ then $v \notin \mathscr{W}_i$) and u and v are compatible walks in G'. Clearly, G_{clique} is n-partite, where each collection of vertices from $\mathscr{W}_i$ forming a block. Furthermore, if G_{clique} has a clique of size κ then it must be the case that there are κ mutually compatible walks in G'. These walks determines a routing scheme (since they are compatible) that routes κ pebbles to their destination. Now if G_{clique} has a clique number $< \kappa$ then the largest collection of mutually compatible length k walks must be $< \kappa$. Hence number of pebbles that can be routed to their destination in at most k steps will be $< \kappa$.

In order to get a non-trivial approximation ratio we require that $(\Delta+1)^k = O(\log^2 n)$ which implies that the above reduction is polynomial in n. This completes the proof.

□

Chapter 9: Structural Results On Permutation Routing

As we have seen previously, a major focus of research on permutation routing has been to determine the bounds for the routing numbers of well-studied classes graphs. However, there are, to our knowledge, very few results based on specific graph properties, such as connectivity, chromatic number, regularity, forbidden minors etc. One important result among them is as follows.

A graph $G(V, E)$ is a (n, d, α)-expander if for every $X \subset V$ with $|X| \leq n/2$ we have $(|\mathcal{N}(X)| - |X|)/|X| \geq \alpha$. This condition of the neighborhood size tells us that starting from any vertex it is possible to reach another vertex quickly via a random walk. Another important aspect of the expansion property is that it is local. It was shown in [36] that if a graph is an (n, d, α)-expander then its routing number is bounded by $O(\frac{d^2}{\alpha^4}\log^2 n)$. In the best case scenario, when $\alpha = O(\sqrt{d})$, the routing number $O(\log^2 n)$. This also tells us that a graph need not have high degree in order to have a small routing number. In fact it is easy to construct examples where a graph with lot of edges has a large routing number. Consider two cliques joined be a single edge and the permutation that moves every pebble from one clique to the other and vice versa.

9.1 Graph Connectivity

We believe that connectivity properties of a graph play an important role in determine its routing number. Next discuss some basic notions regarding graph connectivity. We assume $G(V, E)$ to be a simple undirected graph, $|V| = n$. Let $A, B \subset V$ and $E(A, B) \subset E$ be the set of edges of G whose endpoints lie inside both the sets. If A, B partitions V then we say $E(A, B)$ is a *cut* of G. If G is not connected then we define $\mathscr{C}(G)$ as the set of connected components of G. A *cut-set* of edges (vertices) is defined as a subset of edges (vertices) which disconnects G. We call a pair of edges, *vertex disjoint* if they do not share any vertices. Similarly, we define vertex disjoint paths. A pair of paths are *edge disjoint* if they do not share any edges.

Definition 9.1. A graph G is h-connected if G is not a clique and the minimum number of vertices whose removal disconnects G is h.

If G is a clique then it is $(n-1)$-connected. Similarly,

Definition 9.2. A graph G is h-edge-connected if we need to remove at least h edges from G to make it disconnected.

The edge connectivity of a clique is also $n-1$.

Like expansion, connectivity is a local property in the following sense. If G is h-connected then every induced subgraph G' will have at least h vertex-disjoint edges in the cut to $G \setminus G'$. Intuitively if a graph has a small routing number then it seems it should have good connectivity and vice-versa. In this chapter we give a result that moves us closer to proving this fact. On the other hand, it is easy to see that if G does not have good connectivity then it cannot have a small routing number.

This was proved in [36]. Let C be a cutset of vertices and A, B two components of G separated by C as seen in Figure 9.1.

Theorem 9.1. [36] If G has partition A, B, C of vertices as shown then

$$rt(G) \geq \frac{2}{|C|} \min(|A|, |B|)$$

.

This follows from the fact there are at most $|C|$ vertex disjoint paths from A to B via C. It takes 2 steps to move from A to B via C. If a permutation π requires every pebble of A to move to B (assuming $|A| \leq |B|$) then we need at least $\frac{2|A|}{|C|}$ steps. Hence, if G is h-connected, in the worst case $|C| = h$ and $\min(|A|, |B|) = n/2 - h$, then $rt(G) = \Omega(n/h)$. A similar result was shown when C is a cutset of edges.

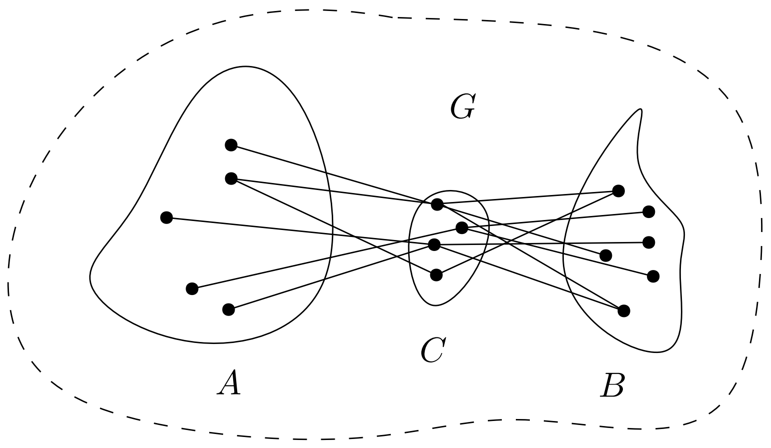


Figure 9.1: A situation where C is a cutset of vertices and sets A, B partitions $V \setminus C$.

9.2 Structural Results

First we give a conditional upper bound for the routing number based on vertex connectivity. We do not believe edge connectivity to be of significant importance with respect to routing number. This is because a large value of edge connectivity does not necessarily imply many vertex disjoint paths (where we can routes pebbles in parallel). In fact a large set of edge-disjoint paths may share many vertices, creating bottlenecks. See for example Figure 9.2 below.

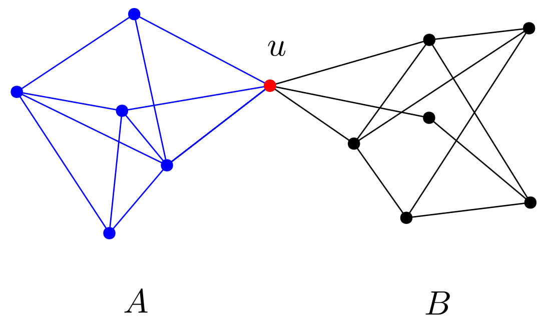


Figure 9.2: Although there are 3 edge disjoint paths between A and B all this paths share the vertex u hence can only be activated (matched) one at a time.

Our second result concerns the relation between *clique number* (size of the largest clique) and routing number. It is known that having a large clique does not necessarily guarantee a small routing time for all permutations. On the other hand we know that the complete bipartite graph, whose clique number is 2, has a constant routing number of 4. Even with these reservations a determining a relation between clique and routing number is important, even if to show that a global property like clique number is inadequate in determining the routing number. We summarize our results of this chapter below:

1. If G is h-connected then G has a routing number of $O(nr_G)$.

   Here $r_G = \min \mathit{rt}(G_h)/|G_h|$, over all induced connected subgraphs $|G_h| \leq h$.

2. A connected graph with a clique number of κ has a routing number of $O(n - \kappa)$.

9.2.1 An Upper Bound For h-connected Graphs

Let G_h be a induced connected subgraph of G having h vertices, we will show $\mathit{rt}(G) = O(n\ \mathit{rt}(G_h)/h)$. Hence if G has a h-clique then $\mathit{rt}(G) = O(n/h)$. In fact the result is more general. If G_h is an induced subgraph with $\leq h$ vertices such that $r = \mathit{rt}(G_h)/|G_h|$ is minimized then $\mathit{rt}(G) = O(nr)$.

We use the classical Lovasz-Gyori partition theorem for h-connected graphs for this purpose:

Theorem 9.2 (Lovasz-Gyori[49])**.** *If G is a h-connected graph then for any choice of positive numbers $n_1, \ldots, n_h$ with $n_1 + \ldots + n_h = n$ and any set of vertices $v_1, \ldots, v_h$ there is a partition of the vertices $V_1, \ldots, V_h$ with $v_i \in V_i$ and $|V_i| = n_i$ such that the induced subgraph $G[V_i]$ is connected for all $1 \leq i \leq h$.*

We prove a combinatorial result. We have a lists L_i, $1 \leq i \leq a$, each of length b. Each element of a list is a number c, $1 \leq c \leq a$. Further, across all lists, each number c occurs exactly b times.

Lemma 8. Given lists as described, there exists an $a \times b$ array A such that the ith row is a permutation of L_i and each column is a permutation of $\{1, 2, 3, \ldots, a\}$.

Proof. By Hall's Theorem for systems of distinct representatives [50], we know that we can choose a representative from each L_i to form the first column of A. The criterion of Hall's Theorem is that, for any k, any set of k lists have at least k distinct numbers; but there are only b of each number so $k-1$ numbers can not fill up k lists. Now remove the representative from each list, and iterate on the collection of lists of length $b-1$. $\square$

To prove our upper bound we need an additional lemma.

Lemma 9. Given a set S of k pebbles and tree T with k pebbles on its k vertices. Suppose we are allowed an operation that replaces the pebble at the root of T by a pebble from S. We can replace all the pebbles in T with the pebbles from S in $\Theta(k)$ steps, each a replace or a matching step.

Proof. Briefly, as each pebble comes from S it is assigned a destination vertex in T, in reverse level order (the root is at level 0). After a replace-root operation, there are two matching steps; these three will repeat. The first matching step uses disjoint edges to move elements of S down to an odd level and the second matching step moves elements of S down to an even level. Each matching moves every pebble from S, that has not reached its destination, towards its destination. The new pebbles move without delay down their paths in this pipelined scheme. (The invariant is that each pebble from S is either at its destination, or at an even level before the next replace-root operation.) $\square$

Theorem 9.3. If G is h-connected and G_h is an induced connected subgraph of order h then $\mathcal{rt}(G) = O(n\ \mathcal{rt}(G_h)/h)$.

Proof. Let $V_h = \{u_1, \dots, u_h\}$ be the vertices in G_h. We take these vertices as the set of k vertices in Theorem 9.2. We call them ports as they will be used to route pebbles between different components. Without loss of generality we can assume $p = n/h$ is an integer. Let $n_1 = n_2 = \dots = n_h = p$ and V_i be the block of the partition such that $u_i \in V_i$. Let $H_i = G[V_i]$. Then for any permutation π on G:

1. Route the pebbles in H_i according to some permutation π_i. Since H_i has n/h vertices and is connected it takes $O(n/h)$ matchings. Since we can use a spanning tree T_i of H_i to accomplish this task.
2. Next use G_h, n/h times, to route pebbles between different partitions. We show that this can be done in $O(n\,rt(G_h)/h)$ matchings. (The "replace-root" step of Lemma 9, is actually the root replacements done by routing on G_h.)
3. Finally, route the pebbles in each H_i in parallel. Like step 1, this also can be accomplished in $O(n/h)$ matchings.

Clearly the two most important thing to attend to in the above procedure are the permutations in step 1 and the routing scheme of step 2. We can assume that each H_i is a tree rooted at u_i (since each H_i has a spanning tree). Thus the decomposition looks like the one shown in Figure 9.3.

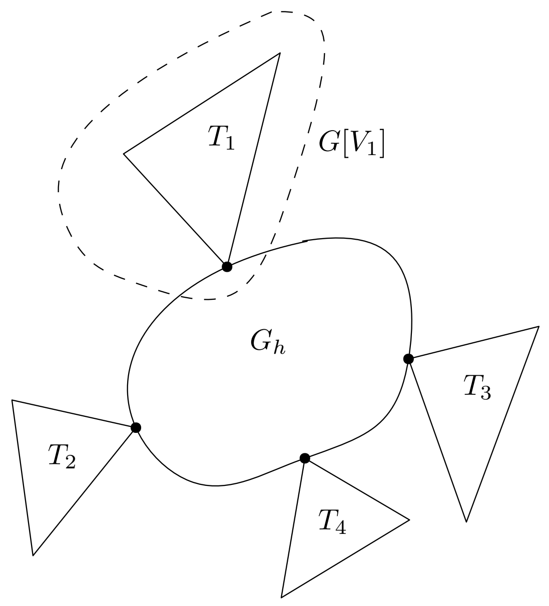


Figure 9.3: G is decomposed into 4 connected blocks, which are connected to each other via G_h.

The permutation π on G indicates for each element of H_i, which H_j it wants to be routed to, where j could be i. So each H_i can build a list L_i of indices of the ports of G_h that it wants to route its elements to (again, possibly to its own port). The lists satisfy the conditions of Lemma 8, with $a = h$ and $b = n/h$, We will use the columns of the array A to specify the permutations routed using G_h in step 2. Note that step 1 will need to preprocess each H_i so that the algorithm of Lemma 9 will automatically deliver the elements of H_i up to u_i in the order specified by the ith row of A.

Once the pebbles are rearranged in step 1, we use the graph G_h to route them to their destination components. Each such routing takes $rt(G_h)$ steps. Between these routings on G_h the incoming pebble at any of the port vertices is replaced by the next pebble to be ported; this requires 2 matching steps as seen in Lemma 9. Hence, after $rt(G_h) + 2$ steps a set of h pebbles are routed to their destination components. This immediately gives the bound of the theorem. $\square$

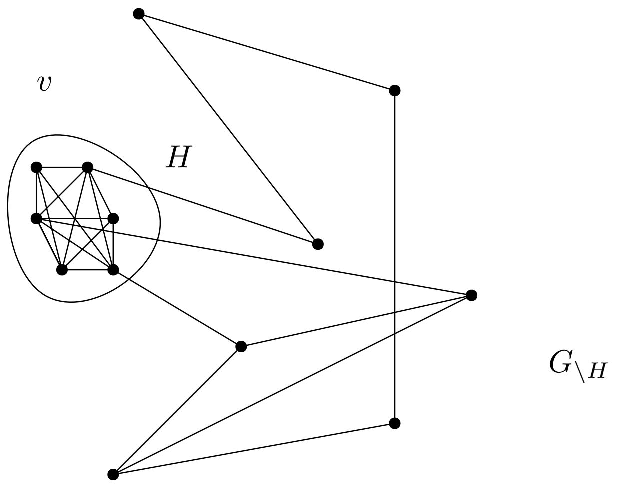


Figure 9.4: The clique H has been contracted into a super-vertex v.

9.2.2 Relation Between Clique Number and Routing Number

Theorem 9.4. For a connected graph G with clique number κ its routing number is bounded by $O(n-\kappa)$.

Proof. Let H be a clique in G of size κ. Let $G_{\setminus H}$ be the minor of G after the contraction of the subgraph H. Let the vertex that H has been contracted to be v. Further, let T be a spanning tree of $G_{\setminus H}$. When routing on $G_{\setminus H}$ we can treat v as any other vertex of $G_{\setminus H}$. The situation is shown in Figure 8.4. Taking into account the fact that v can store more than one pebble internally. When v participates in a matching with some other vertex u in $G_{\setminus H}$ we assume that exchanging pebbles takes 3 steps. This accounts for the fact that the pebble thats need to be swapped with the pebbles at u was not on a vertex adjacent to u in the un-contracted graph G. The basic idea is to break the routing into two steps. In the first step we simply move all pebbles in v whose final detination is not in v (i.e. not in un-contracted H) out. For a tree, it is known that [37] we can route a subset of p pebbles where each pebble needs to be moved at most l distance in $\leq p+2l$ steps. Since T has a diameter at most $n-\kappa$ and at most $\min(\kappa, n-\kappa)$ pebbles need to be moved out of v the first step can be accomplished $\leq 3(n-\kappa)+O(1)$ steps. At this point we can employ any tree routing algorithm on T where we charge 3 time units whenever v is part of the matching to route all the pebbles in $G_{\setminus H}$. If we use the algorithm presented in [41], which requires $3n/2+o(n)$ steps, then we see that the routing takes at most $\frac{15}{2}(n-\kappa)+o(n)$ steps for any permutation.

□

Chapter 10: Sorting Permutations and Sorting Number

In this chapter we introduce the well known concept of sorting networks from a new perspective: as a reconfiguration problem on graphs. Let us first begin with an overview of sorting networks.

An oblivious sorting algorithm is one that has decided what key comparisons to make before seeing the input. A sorting network, which typically arises in discussions of parallelism, is an oblivious algorithm in which the comparisons are grouped into consecutive *stages*, where the comparisons in each stage are disjoint. In the context of permutation sorting, these stages are matchings where we compare the matched pairs and if necessary swap them. Which vertex to send the lower rank pebble and which to send the higher rank one induces a direction on edges. Hence we augment our matching model to include a possible direction for the matched edge. Note that the underlying graph G remain undirected.

Most prior work has regarded all possible pairs as candidates for "comparison-swap operations," (i.e., comparators) where we would say the given graph is a complete network. There is also much work on specific graphs, such as hypercubes, shuffle-exchange graphs, meshes and linear arrays. On the other hand when general graphs are considered in the past typically the discussion has assumed sorting algorithms to be distributed and asynchronous. We strike a new middle ground in allowing the graphs to be arbitrary but synchronous, allowing a schedule of parallel swaps. The terminology of sorting networks is natural in this setting. We focus here on general trees since every connected graph has a spanning tree it is another fundamental case.

We regard a sorting network as a sequence of stages and each stage is specified by a matching of some pairs of vertices; a comparator is assigned to each matched pair. There are fixed locations each containing a key (label or rank of a pebble) and comparators look at the keys at the two locations and swap them if they are not in the order desired by the underlying oblivious algorithm. (A location is a memory location, but often, in diagrams, it is identified with a set of memory locations connected by wires.) The goal is move the pebbles to the vertex with the matching label. The *depth* of a sorting network is the number of stages.

We study the following restricted variant of sorting networks. We begin by taking a graph $G = (V, E)$, where the vertices correspond to the locations of an oblivious sorting algorithm, $V = \{1, 2, \ldots, n\}$. Let a *sorted order* of G be given by a permutation π that assigns the rank $\pi(i)$ to the vertex $i \in V$. As mentioned before, the keys will be modeled by labeled pebbles, one per vertex; the label on the vertex indicates the destination vertex, so if it is labeled k it will be sent to the vertex i with rank k, $k = \pi(i)$. This is slightly different from how used labels for permutation roting. However assigning a rank to a vertex (in addition to its label) allows more constrained subproblems. In simple situations π is the identity permutation, so we just send the pebble labeled i to vertex labeled i. The edges of G represent pairs of vertices where the pebbles can be compared and/or swapped. Given a graph G the goal is to design a sorting network that uses only the edges of G. We formally define such a sorting network.

We use matchings of the graph to represent the swaps that are to be done in parallel. Of course matched edges are vertex disjoint. Usually the matched edges are directed, indicating which vertex is to receive the minimum of the two pebbles, with the maximum going the other way. Sometime a matched edge will be undirected indicating the two pebbles are swapped regardless. The reason for this is that when the graph is sparser some of the sorting effort is routing pebbles to a point where they are adjacent and can be compared.

Definition 10.1 (Sorting Network on a Graph)**.** A sorting network is a triple $\mathcal{S}(G, \mathscr{M}, \pi)$ such that:

1. $G = (V, E)$ is a connected graph with a bijection $\pi : V \to \{1, \ldots, n\}$ specifying the sorted order on the vertices. Initially, each vertex of G contains a pebble having some value.

2. $\mathscr{M} = (M_1, \ldots, M_t)$ is a sequence of matchings in G, for which some edges in the matching are assigned a direction. Sorting occurs in stages. At stage i we use the matching M_i to exchange the pebbles between matched vertices according to their orientation. For an edge $\overrightarrow{uv}$, when swapped the smaller of the two pebbles goes to u. If an edge is undirected then the pebbles swap regardless of their order.

3. After t stages the vertex labeled i contains the pebble whose rank is $\pi(i)$ in the sorted order. We stress that this must hold for all $(n!)$ initial arrangement of the pebbles. $|\mathscr{M}|$ is called the *depth* of the network.

Definition 10.2 (Sorting Number)**.** Sorting number $\mathfrak{st}(G)$ of a graph G is defined to be minimum depth of any sorting network $\mathcal{S}(G, \mathscr{M}, \pi)$, where $\mathscr{M}$ and π are free variables. Additionally, $\mathfrak{st}(G, \pi)$ is the sorting number of G over all possible sorting networks $\mathcal{S}(G, \mathscr{M}, \pi)$, where $\mathscr{M}$ is a free variables but π is a fixed sorted order.

Note that if G is connected then there exists a spanning tree T of G and $\mathfrak{st}(G) \leq \mathfrak{st}(T)$. We start by briefly restating some results from the oblivious sorting literature on specific graphs. This will provide a better context for the proposed framework, which unifies these previous results as special case of graph reconfiguration. Given below in Figure 10.1 is an illustration of a sorting network for a given tree.

The path graph P_n is the simplest tree case. We know that $\mathfrak{st}(P_n) = 2n$. This follows from the fact that the classical odd-even transposition sort (OETS) [51] takes $2n$ matching steps and that it is known to be optimal. Some known bounds for the sorting numbers of common graphs are summarized in Table 10.1. The Ajtai-Komlos-Szemerdi (AKS) sorting network [52] directly gives an upper bound of $O(\log n)$ for the sorting number of the complete graph K_n. In this case there is a matching lower bound. For the n-cube Q_n (with 2^n vertices) we can use Batcher's bitonic sorting network, which has a depth of $O((\log n)^2)$ [53]. This was later improved to $2^{O(\sqrt{\log \log n})} \log n$ by Plaxton and Suel[54]. We also have a lower bound of $\Omega(\frac{\log n \log \log n}{\log \log \log n})$ due to Leighton and Plaxton [55]. For the square mesh $P_n \times P_n$

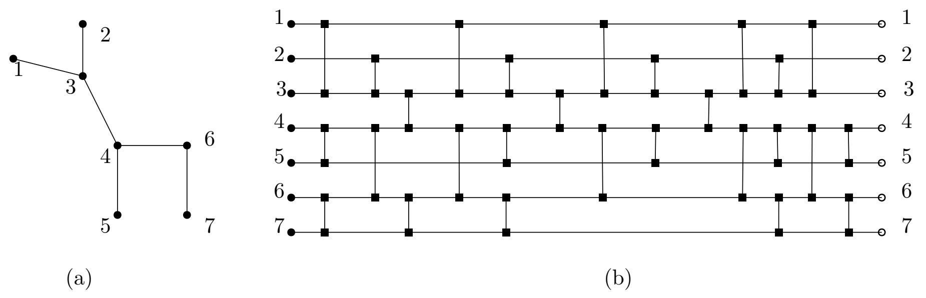


Figure 10.1: Figure (b) corresponds to a sorting network (not necessarily of optimal depth) for the tree given in (a). Here π is the identity permutation. Every comparator, given by vertical segments (all directed upwards) joining two wires, always conform to the edges of the tree.

it is known that $\mathfrak{st}(P_n \times P_n) = 3n + o(n)$, which is tight with respect to the constant factor of the largest term. This follows from results of Schnorr and Shamir [56], where they introduced the $3n$-sorter for the square mesh. We also have a tight result for the general d-dimensional mesh of $\Theta(dn)$ due to Kunde [57]. These results are, in fact, more general, as they apply to meshes with non-uniform aspect ratios.

Table 10.1: Known bounds on the sorting numbers of various graphs

Graph	Lower Bound	Upper Bound	Remark
Complete Graph (K_n)	$\log n$	$O(\log n)$	AKS Network [52]
Hypercube (Q_n)	$\Omega(\frac{\log n \log\log n}{\log\log\log n})$	$2^{O(\sqrt{\log\log n})}\log n$	Plaxton et. al[54, 55]
Path (P_n)	$n-1$	n	OETS. [51]
Mesh ($P_n \times P_n$)	$3n - 2\sqrt{n} - 3$	$3n + O(n^{3/4})$	Schnorr & Shamir [56]
d-dimensional Mesh	$\Omega(dn)$	$O(dn)$	Kunde [57]

Our sorting network, when it uses undirected edges, uses the permutation routing model to move pebbles around in the graph when we know where we want them to go (i.e. without making comparisons). Since comparisons can only be done at adjacent vertices, we use routing to move pebbles towards other pebbles. In particular, routing will always be done for some permutation τ, which does not depends on the label value of the pebbles, but the rank of the vertices. Hence, these routing steps preserve the oblivious nature of our algorithm.

10.1 Some Additional Results On Routing

In this section we present some additional properties of permutation routing, which we will use later in our discussion on sorting. We start this section with the following simple lemma.

Lemma 10. For any graph G and any order π of the vertices of G it holds that

$$rt(G) \leq st(G, \pi) \leq st(G) + rt(G).$$

Proof. We show first that $rt(G, \sigma) \leq st(G, \pi)$ for any two permutations π, σ of the vertices. Indeed, suppose that the keys of the pebbles are $\{1, \dots, n\}$. For all $i \in V$ place the pebble ranked i in the vertex $\sigma^{-1}(\pi^{-1}(i))$. Then there exists a sorting network of depth $st(G, \pi)$ that sends the pebble ranked i to the vertex $\pi^{-1}(i)$ for all $i \in V$. That is, the pebble from the vertex $j = \sigma^{-1}(\pi^{-1}(i))$ is sent to the vertex $\pi^{-1}(i) = \sigma(j)$. Therefore, $rt(G, \sigma) \leq st(G, \pi)$ for all permutations σ, and thus $rt(G) \leq st(G, \pi)$.

For the upper bound let $\mathcal{S}(G, \mathcal{M}, \tau)$ be a sorting network on G of depth $st(G) = st(G, \tau)$. We use τ to create another sorting network $\mathcal{S}(G, \mathcal{M}', \pi)$ of depth at most $st(G) + rt(G)$. This is done in two stages. First we apply the sorting network $\mathcal{S}(G, \mathcal{M}, \tau)$. After this stage we know that the pebble at vertex i has a rank $\tau(i)$. Next, we apply a routing strategy with at most $rt(G)$ steps that routes to the permutation $\pi^{-1} \circ \tau$, i.e., sending a pebble in the vertex i to $\pi^{-1}(\tau(i))$ for all $i \in V$. After this step the vertex i contains the pebble of rank $\pi(i)$. This proves that $st(G, \pi) \leq st(G) + rt(G)$. □

The above lemma implies that if we construct a sorting network for an arbitrary sorted order on the vertices then we suffer a penalty of $rt(G)$ on the depth of our network as compared to the optimal one.

10.1.1 Routing on subgraphs of G

Below, we study the notion of routing a subset of the pebbles to a specific subgraph. We start with the following lemma, which is from [58].

Lemma 11. Let T be a tree with diameter d, and let P be a path of length d in T. We can route any set of d pebbles to P in $3d - 2$ steps.

Proof sketch. The proof is done via induction on the number of pebbles to move. The basic idea is that after some initial delay we can pipeline the movement of pebbles so that at each time step sufficiently many pebble can reach P. $\square$

Next we discuss the question of partial routing, where only a small number of pebbles are required to reach their destination. These results we found to be of independent interest.

Definition 10.3. Given a graph $G = (V, E)$ let $A, B \subset V$ be two subset of vertices with $|A| = |B|$, not necessarily distinct. Let π_{AB} be a bijection between A and B. Routing of the pebbles from A to their respective destinations on B given by π_{AB} is a partial routing in G, where each pebble in $a \in A$ is required to reach $\pi_{AB}(a) \in B$ using the edges of G (and there are no requirements on the pebbles outside A). Further,

1. Let $rt(G, A, B, \pi_{AB})$ be the minimum number of matchings needed to route every pebble $a \in A$ to $\pi_{AB}(a) \in B$ using the edges of G.
2. Let $rt(G, A, B) = \max_{\pi_{AB}} rt(G, A, B, \pi_{AB})$.
3. For $U \subseteq V$ let $rt_U(G) = \max_{A \subseteq V} rt(G, A, U)$.
4. For $p \in [1 \dots |V|]$ let, $rt_p(G) = \max_{A,B \subset V, |A|=|B| \leq p} rt(G, A, B)$

Clearly, for any connected n-vertex graph G we have $rt(G) = rt_n(G)$. Some of the bounds for $rt(G)$ also holds for $rt_p(G)$. For example, $rt_p(G) \geq d$, where d is the diameter of G. Furthermore, $rt_p(G) = \Theta(rt(G))$ for any p if and only if $rt(G) = \Theta(d)$. We illustrate $rt_p(G)$ by computing it explicitly for some typical graphs. Recall from [36] that

$rt(K_n) = 2$. It is easy to see that $rt_p(K_n) = 2$ for all $p \geq 3$, and $rt_2(K_n) = 1$. For the complete bipartite graph we have $rt_{n/2}(K_{n/2,n/2}) = 2$ and is $rt_p(K_{n/2,n/2}) = 4$ for $p > n/2$.

Theorem 10.1. For any tree T with diameter d, $rt_p(G) = O((d+p)\min(d, \log \frac{n}{d}))$.

Proof. The proof is similar to the proof used in [36] for determining the routing number of trees. Let r be the centroid whose removal disconnects the tree into a forest of trees each of which is of size at most $n/2$. Let $(T_1, \ldots, T_r)$ be the set of trees in the forest, with $r \in T_1$. For a tree T_i let S_i be the set of "improper" pebbles that need to be moved out of T_i. All other pebbles in T_i are "proper". In the first round we move all the pebbles in S_i as close to the root of T_i as possible, for all i. Using the argument used in [36] it can be shown that for a tree with diameter d this first phase can be accomplished in $c_1 d$ steps for some constant c_1. First we label each node in T_i as odd or even based on their distance from r_i, the root of T_i. In each odd round we match nodes in odd layers with proper pebbles to one of its children containing an improper pebble if one exists. Similarly, in even rounds we match nodes in even layers with proper pebbles to one of its children containing an improper pebble if one exists. Since T has diameter d any path from r_i to some leaf must be of length at most $d-1$. Now consider an improper pebble u initially at distance k from the root. During a pair of odd-even matchings either the pebble moves one step closer to the root or one of the following must be true: (1) another pebble from one of its sibling node jumps in front of it or (2) there is some improper pebble already in front of it. It can then be argued (we omit the details here) that after $c_1 d$ matchings for some constant c_1 if u ends up in position j from r_i then all pebbles between u and r_i must be improper. Next we exchange a pair of pebbles between subtrees using the root vertex r, since at most $p/2$ pairs needs to be exchanged, the arguments used in [36] can be modified to show that this phase also takes $c_2 p$ steps for some constant c_2. After each pebble is moved to their corresponding destination subtrees we can route them in parallel. Noting that each tree T_i has diameter at most $d-1$. Hence we have the following recurrence:

$$T(n, d, p) \leq T(n/2, d-1, p) + c_1 d + c_2 p \tag{10.1}$$

where $T(n,d,p)$ is the time it takes to route p pebbles in a tree of diameter d with n vertices. Taking $T(\cdot,d,p)=O(d)$, and solving equation (9.1) gives the stated bound of the lemma. $\square$

10.2 General Upper Bounds on $\mathcal{st}(G)$

The AKS sorting network can be trivially converted into a network of depth $O(n\log(n))$ by making a single comparison in each round. However, it is not clear a priori whether for *any* graph there is a sorting network of depth $O(n\log(n))$. We show later that this bound indeed holds for all graphs.

We relate the sorting number of a graph to its routing number and the size of its maximum matching.

Theorem 10.2. Let G be an n-vertex graph with routing number $\mathcal{rt}(G)$ and a matching of size $\nu(G)$. Then $\mathcal{st}(G)=O\left(n\log(n)\cdot\frac{\mathcal{rt}(G)}{\nu(G)}\right)$.

Proof. We prove the theorem by using G to simulate the AKS sorting network on the complete graph K_n of depth $O(\log(n))$. Specifically, we show that each stage (a matching) of the sorting network on K_n can be simulated by at most $O(\frac{n}{\nu(G)}\mathcal{rt}(G))$ stages (matchings) in G. Let M be a matching at some stage of the AKS sorting network on the complete graph. We simulate the compare-exchanges and swaps in M by a sequence of matchings in G as follows. First we partition the edges in M into $t=\lceil n/\nu(G)\rceil$ disjoint subsets $M=M_1\cup\cdots\cup M_t$, where $|M_i|=\nu(G)$ for all except maybe the last set M_t, which can be smaller. Let M_G be a maximum matching in G. Corresponding to each pair $(u,v)\in M_i$ we pick a distinct pair $(u',v')\in M_G$, this can always be done since the sets M_i and M_G are of the same size. Note that the pair (u,v) may not be adjacent in G, and so, we route each pair $(u,v)\in M_i$ to its destination in $(u',v')\in M_G$. This can be done in $\mathcal{rt}(G)$ steps, where each step consists of only undirected matchings. Once the pairs have been placed into their corresponding positions we relabel the vertices such that the pair labeled (u',v')

is now (u, v). Unmatched vertices keep their label. Since the pairs in M_i are now adjacent in G we can perform the compare-exchange or swap operation according to M_i. Therefore, the total number of matchings to execute the i^{th} set of compare-exchanges and swaps in M_i is $\mathit{rt}(G) + 1$ in G. We remark that the routing maintains the oblivious nature of the network, and the swaps made while routing, are data independent. We can then reverse the oblivious routing that set up the exchanges for M_i. The set up for M_{i+1} invokes routing a different permutation. Note that the two phases of oblivious routing can be combined by using the composition of the two permutation. This implies that we can simulate M using at most $(\mathit{rt}(G) + 1) \cdot t = O(\frac{n}{\nu(G)} \cdot \mathit{rt}(G))$ matchings in G. Therefore, since the depth of the AKS sorting network on the complete graph K_n is $O(\log(n))$, we conclude that $\mathit{st}(G) = O(n \log(n) \cdot \frac{\mathit{rt}(G)}{\nu(G)})$, as required. □

In the following theorem we upper bound $\mathit{st}(G)$ for graphs G that contain a large subgraph H whose $\mathit{st}(H)$ is small. This result will later be used to bound sorting number of a tree with given diameter.

Theorem 10.3. Let G be an n-vertex graph, and let H be a vertex-induced subgraph of G on p vertices. Then $\mathit{st}(G) = O\left(\frac{n}{p} \log(\frac{n}{p}) \cdot (\mathit{rt}(G) + \mathit{st}(H))\right)$.

Specifically, we will prove that if H be a vertex-induced subgraph of G on p vertices then $\mathit{st}(G) = O\left(\frac{n}{p} \log(\frac{n}{p}) \cdot (\mathit{rt}_H(G) + \mathit{st}(H))\right)$, where $\mathit{rt}_H(G)$ bounds the number of matchings required to route any set of p vertices to H. (Here we slightly abuse the notation from Definition 10.3, by identifying the subscript in $\mathit{rt}_H(G)$ with the vertex-set of H.) Later we will use this result to prove Theorem 10.4.

Proof. Let us partition the vertex set V of G into $q = \lceil n/\lfloor p/2 \rfloor \rceil$ parts $V = A_1 \cup \cdots \cup A_q$ in a balanced manner (i.e., the size of each A_i is either $\lfloor n/q \rfloor$ or $\lfloor n/q \rfloor + 1$). Note that $|A_i| + |A_j|$ could be less than p. However it is always possible to modify an existing sorting network to sort a subset of elements in the same number of stages. Let K_q be a complete graph whose vertices are identified with $\{A_1, \ldots, A_q\}$, and let S be an oblivious sorting algorithm with $O(q \log q)$ comparisons on the complete graph K_q. (Here the sequence of comparisons is performed sequentially, not in parallel.) In an ordinary sorting network in each step we perform a compare-exchange or a swap between two matched vertices (i, j) so that if $i < j$, then the pebble in the vertex i will be smaller than the pebble in j We will simulate S on G using a sorting network on H by sorting in each stage the elements in $A_i \cup A_j$. That is, for $i < j$ we are going to sort the elements in $A_i \cup A_j$ so that all the elements of A_i are smaller than every element of A_j, and the elements within each subset are internally sorted. This is done using an optimal sorting network in H, which we will denote by $\mathcal{S}_H$.

We can simulate any such compare-exchange in G between pairs of sets in A in $O(\mathfrak{rt}(G) + \mathfrak{st}(H))$ steps. Indeed, suppose the k^{th} round in S compares the vertices $i < j$. In order to simulate this comparison we first route all the pebbles in $A_i \cup A_j$ to the subgraph H and relabel the vertices. This relabeling is done so that we can keep track of the vertices when sorting H. Then we use $\mathcal{S}_H$ to sort $A_i \cup A_j$ which takes $\mathfrak{st}(H)$ steps. Once the sorting is done we split up the sets again and appropriately relabel the vertices so that the first $|A_i|$ vertices in the sorted order on H will now belong to A_i and the next $|A_j|$ vertices will belong to A_j. If instead the k^{th} comparison is actually a swap then we simply swap the labels of the multisets (A_i is labeled A_j and vice versa). Hence performing the above simulation takes $O(\mathfrak{rt}_H(G) + \mathfrak{st}(H))$ steps per compare exchange or swap operation, which gives the result of the theorem. $\square$

In the proof of Theorem 10.3 above we only used an oblivious sorting algorithm with $O(q \log q)$ comparisons on the complete graph K_q, and did not use the fact that the comparisons can be done in parallel, e.g., using the AKS sorting network. This is because Theorem 10.3 only assumes that there is one subgraph H with small $\mathfrak{st}(H)$. If instead we assumed that there are many such subgraphs, then we could sort the A_i's in different subgraphs in parallel. This is described in the corollary below.

Corollary 3. (Due to Shinkar[59]) Let $G = (V, E)$ be an n-vertex graph. Let $V = V_1 \cup$

$\cdots \cup V_q$ be a partition of the vertices, with $|V_i| = n/q$ for all $i \in \{1,\dots,q\}$, such that H_i, the subgraph induced by V_i, is connected for each $i \in \{1,\dots,q\}$. Then

$$\mathfrak{s}t(G) = O\left(\log(q) \cdot (\mathfrak{r}t(G) + \max_{k\in\{1,\dots,q\}}\{\mathfrak{s}t(H_k)\})\right).$$

Proof sketch. The proof uses the same idea that Theorem 10.3. We start by partitioning the vertex set V of G into $2q$ parts $V = A_1 \cup \cdots \cup A_{2q}$ of equal sizes. Then, we simulate oblivious sorting algorithm on K_{2q} with the sets A_i. The only difference is that instead of an oblivious sorting algorithm with $O(q\log(q))$ comparisons on the complete graph K_{2q} we use the AKS sorting network on $2q$ vertices of depth $O(\log(q))$. In each round of the sorting network there are at most q comparisons, and the corresponding sorting of $A_i \cup A_j$ can be performed in parallel, one in each H_k in time $\mathfrak{s}t(H_k)$. □

Theorem 10.4. Let G be an n-vertex graph, and suppose that G contains a simple path of length d. Then $\mathfrak{s}t(G) = O(n\log(n/d))$. In particular, for every n-vertex graph G it holds that $\mathfrak{s}t(G) = O(n\log(n))$.

Proof. It is easy to see that if G contains a simple path of length d, then G has a spanning tree T with diameter at least d. The proof follows easily from Theorem 10.3 and Lemma 11. Indeed, in the setting of Theorem 10.3, let H be a path of length d in T. Then $\mathfrak{s}t(H) = d$. By Theorem 10.3 if any set of d vertices can be routed to H in r steps, then $\mathfrak{s}t(T) \le O(\frac{n}{d}\log(\frac{n}{d}) \cdot (r + \mathfrak{s}t(H)))$. By Lemma 11 we have $r = O(d)$, and thus $\mathfrak{s}t(G) \le \mathfrak{s}t(T) = O(n\log(n/d))$. □

Theorem 10.5. Let G be an n-vertex graph with maximal degree Δ. Then $st(G) = O(\Delta n)$.

This result was proven in [58], who proved that the acquaintance time of a G, is upper bounded by $20\Delta n$. Let T be a spanning tree of G with maximum degree Δ. The basic idea is to use an n round sorting network for P_n (such as OETS [51]), and simulate this network in T with an overhead that depends only on Δ. This is done via walking along a *contour*

P_{2n-3} of T with n vertices (Figure 10.2). A contour of a tree is a cycle that visits every edge exactly once and every vertex as often as its degree. The δ fact comes from the fact that two odd or two even edges in the path which shares the same copy of a vertex of T cannot be matched in the same routing step.

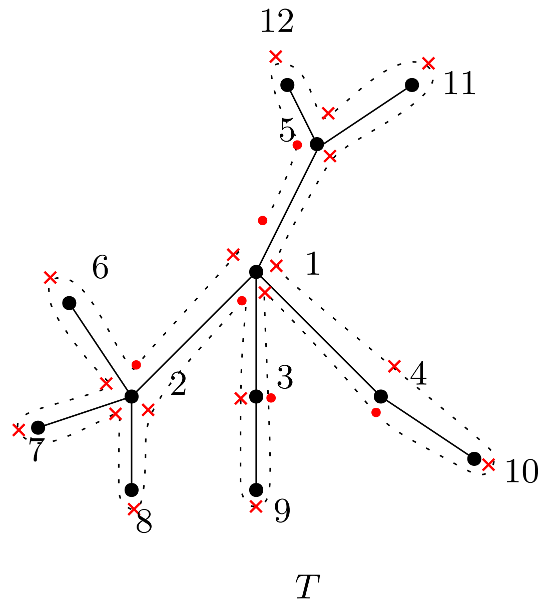


Figure 10.2: In the tree above, rooted at vertex labeled 1, the dotted line traces a contour for the tree. The contour represented by a path consists of the following sequence of vertices from T and their duplicates (indicated by crosses): $(1_1, a_1, 6_1, 2_1, 7_1, 2_2, 8_1, 2_3, a_2, 3_1, 9_1, a_3, a_4, 1_2, 10_1, 4_1, 1_3, 5_1, 11_1, 5_2, 12_1, a_5, a_6)$, Where i_j is the j^{th} copy of vertex labeled i and a_k's are additional vertices indicated by small red dots.

10.3 Bounds on Concrete Graph Families

Below we state several results concerning the sorting time of some concrete families of graphs.

Proposition 2 (Tree)**.** For a tree with diameter d and maximum degree of Δ we have, $st(T) = O(\min(\log \frac{n}{d}), \Delta n)$.

Proof. This immediately follows by combining the results from Theorem 10.4 and Theorem 10.5. $\square$

This result is optimal when T is a star, as it requires at $\Omega(n \log n)$ stages to sort in a star. It is also optimal for a path, trivially. We believe that the optimality holds asymptotically for arbitrary trees. However, this result is non-constructive, as it does not give an actual sorting network for trees. In the next chapter we will construct an actual sorting network but with an worse bound on the depth.

Proposition 3 (Complete p-partite graph)**.** Let G be the complete p-partite graph $K_{n/p,\ldots,n/p}$ on n vertices. Then $\mathscr{st}(G) = \Theta(\log n)$.

Proof. The lower bound is trivial. For the upper bound note that $K_{n/p,\ldots,n/p}$ contains the bipartite graph $K_{\lfloor \frac{p}{2} \rfloor \frac{n}{p}, \lceil \frac{p}{2} \rceil \frac{n}{p}}$. In particular, it contains a matching of size $\nu(G) = \lfloor \frac{p}{2} \rfloor \cdot n/p$. Therefore, by Theorem 3 in [36] and the remark after the proof, we have $\mathscr{rt}(G) \leq \mathscr{rt}(K_{\lfloor \frac{p}{2} \rfloor \frac{n}{p}, \lceil \frac{p}{2} \rceil \frac{n}{p}}) \leq 2 \left\lceil \frac{\lceil \frac{p}{2} \rceil}{\lfloor \frac{p}{2} \rfloor} \right\rceil + 2 \leq 6$, and hence by Theorem 10.2 it follows that $\mathscr{st}(G) \leq O(\log(n))$. $\square$

A graph G is said to be a (n, d, λ)-expander if it is a d-regular graph on n vertices and the absolute value of every eigenvalue of its adjacency matrix other than the trivial one is at most λ. In Chapter 6 we defined expanders using the expansion coefficients instead, but both of these definitions are equivalent.

Proposition 4 (Expander graphs)**.** Let G be an (n, d, λ)-expander. Then,

$$\mathscr{st}(G) \leq O(\frac{d^3}{(d-\lambda)^2} \log^3(n)).$$

In particular, if $\lambda < (1 - \frac{1}{\log^c(n)})d$, then $\mathscr{st}(G) \leq O(d \cdot \log^{2c+3}(n))$.

Proof. Recall from [36] that if G is an (n, d, λ)-expander, then $\mathscr{rt}(G) = O\left(\frac{d^2}{(d-\lambda)^2} \log^2(n)\right)$.

Therefore, since any d-regular graph contains a matching of size $n/2d$ it follows from Theorem 10.2 that $\mathcal{st}(G) \leq O(\frac{d^3}{(d-\lambda)^2}\log^3(n))$. $\square$

A graph $G = (V, E)$ is said to be *vertex transitive* if for any two vertices $u, v \in V$, there is some automorphism[1] $f \colon V \to V$ of the graph such that $f(u) = v$.

Proposition 5 (Vertex transitive graphs [59])**.** Let G be a vertex transitive graph with n vertices of degree polylog(n). Then the diameter of G, given by $d_G = O((\log n)^{c_1})$ if and only if $\mathcal{st}(G) = O((\log n)^{c_2})$, where c, c_2 are constants

Proof. It is trivial that $d_G \leq \mathcal{st}(G)$, where d_G is the diameter of G. For the other direcion, Babai and Szegedy [60] showed that for vertex-transitive graphs if the diameter of G is $O((\log n)^c)$, for some constant c, then its vertex expansion rate is $\Omega(1/(\log n)^c)$. Therefore, $\lambda \leq d(1 - 1/(\log n)^c)$, where $d = O((\log n)^c)$ is the degree of the graph. Therefore, by Proposition 4 we have $\mathcal{st}(G) = O((\log n)^c)$. $\square$

Since all Cayley graphs are also vertex transitive, the above bound is applicable to them as well.

Next we bound the sorting number of Cartesian product of two graphs in terms of its components.

Proposition 6. Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ be two graphs and $G = G_1 \square G_2$. Then

$$\mathcal{st}(G) \leq O(\min(\log|V_1|(\mathcal{rt}(G) + \mathcal{st}(G_2)), \log|V_2|(\mathcal{rt}(G) + \mathcal{st}(G_1))).$$

Proof. We will prove this in terms of $\mathcal{rt}(G)$, and then use Theorem 4 in [36] saying that

$$\mathcal{rt}(G) \leq \min\{\mathcal{rt}(G_1), \mathcal{rt}(G_2)\} + \mathcal{rt}(G_1) + \mathcal{rt}(G_2)$$

Since G has $|V_1|$ vertex disjoint subgraphs that are copies of G_2 we can apply Corollary 3

[1]A mapping $f \colon V \to V$ is an automorphism of $G = (V, E)$ if for all $v_1, v_2 \in V$ it holds that $(v_1, v_2) \in E \Leftrightarrow (f(v_1), f(v_2)) \in E$

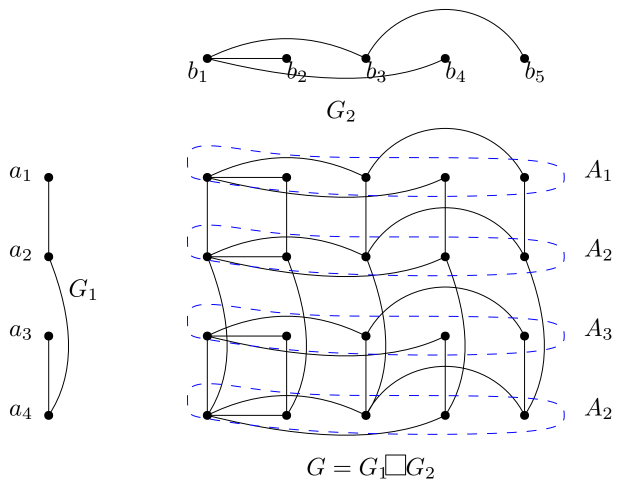


Figure 10.3: The Cartesian product graph $G = G_1 \square G_2$. The rows highlighted by blue regions represents copies of G_2.

with these $q = |V_1|$ subgraphs and all H_i being isomorphic to G_2. Therefore, we get $\mathcal{s}t(G) \leq O(\log(|V_1|) \cdot (\mathcal{r}t(G) + \mathcal{s}t(G_2))$. The bound $\mathcal{s}t(G) \leq O(\log(|V_2|) \cdot (\mathcal{r}t(G) + \mathcal{s}t(G_1))$ follows using the same argument by changing the roles of G_1 and G_2. $\square$

As an example of an application of the above corollary consider the d-dimensional mesh $M_{n,d}$ with n^d vertices. We know that $\mathcal{r}t(M_{n,d}) \leq 2dn$ since $M_{n,d} = M_{n,d-1} \times P_n$. Therefore, $\mathcal{s}t(M_{n,d}) \leq O(\log(n^{d-1}) \cdot (\mathcal{r}t(M_{n,d}) + \mathcal{s}t(P_n))) = O(dn \log(n))$. Although this bound is not optimal (it is known [57] that $\mathcal{s}t(M_{n,d}) = O(dn)$), we still find this example interesting.

Chapter 11: Sorting Network On Trees

In this chapter we present an oblivious sorting algorithm for trees. Our OddEvenTreeSort is a natural generalization of the classical OETS algorithm, but it will be explained without reference to OETS. First recall the following fact about trees: for any tree T with n vertices there exists a vertex r whose removal produces connected components of size $\leq n/2$ (i.e., r is a centroid). We root T at r, which we assume has d children, see Figure 11.1. Let the subtree T_i, rooted at a_i, have $n_i \leq n/2$ nodes. Further, every node in T_i has at most β_i children. Further, assume the subtrees are arranged in descending order according to their size from left to right ($n_1 \geq n_2 \geq \ldots \geq n_d$). Define T_i' to be $T_1 \cup \{r\}$, rooted r. The sorted order $\pi(T)$ is defined recursively as follows.

1. *T_1' will have the $n_1 + 1$ smallest pebbles.*

2. *The tree T_i has pebbles whose ranks (labels) are between $\sum_{j=1}^{i-1} n_j + 2$ to $\sum_{j=1}^{i} n_j + 1$.*

3. *Labeling of each subtree T_i (or T_1') is defined recursively based on an appropriately chosen root r_i which partitions T_i in a balanced manner.*

We call this the *multi-root pre-order* (MP) sorted order. While a_i is the root of the subtree T_i of T, from the viewpoint of r each r_i is in the interior of T_i. However the "root" r_i of T_i is determined for each i after detaching T_i and rerooting it; so it is usually not a_i. Given a tree this ordering can be easily precomputed and it is fixed afterwards. Furthermore once we have our sorting network, using Lemma 10 we can easily create another sorting network with a more natural ordering [1] using an additional $O(n)$ steps.

[1]For example we can use the pre-order ranks of the vertices in T as our sorted order.

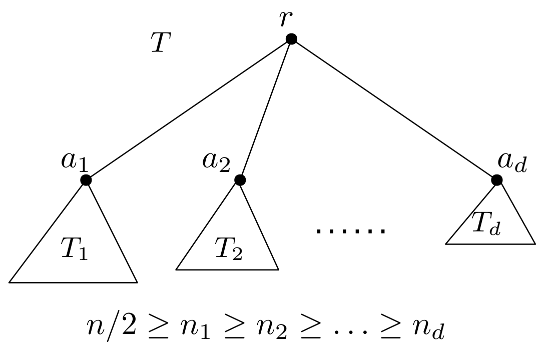


Figure 11.1: A balanced decomposition of a tree.

Algorithm 2: OddEvenTreeSort(T, r)

```
Input  : $T$ with root $r$
Output: Pebbles are sorted according to an MP sorted order
begin
    if $|T| == 1$ then
        return
    for $i$ from $1$ to $d - 1$ do
        for $j$ from 1 to $d - i$ do
            $k \leftarrow n_{i+j}$; Heap$^{\uparrow}(T'_j)$; Heap$^{\downarrow}(T_{j+1})$; Swap$(T'_j, T_{j+1}, k)$;
    pardo $i$ from 1 to $d$ do
        if $i$ == 1 then
            OddEvenTreeSort$(T'_1, r_1)$
        else
            OddEvenTreeSort$(T_i, r_i)$
```

The OddEvenTreeSort(T, r) has two main *phases*. The first phase (lines 4-6) uses the subtrees as buckets to partition the pebbles such that T'_1 gets the first $n_1 + 1$ smallest pebbles, T_2 the next n_2 smallest pebbles and so on. The second phase (lines 7-11) calls

$\mathsf{OddEvenTreeSort}(T_i, r_i)$ recursively for all the subtrees $T_1', T_2, \ldots, T_d$. Sorting on these subtrees happens in parallel. Let the number of matchings needed to partition the pebbles during the first phase be $S(T, r)$. Then the total number of stages in $\mathsf{OddEvenTreeSort}$ is given by the following recurrence:

$$C(T) = S(T, r) + \max(C(T_1'), \max_i C(T_i)) \tag{11.1}$$

For the first phase, there are $d-1$ *passes*. In each pass, since any movement between the subtrees must use the root r, we route the pebbles between each pair of consecutive subtrees, one after the other. We take a pair of consecutive subtrees and move the pebbles so that each pebble in $T_j' = T_j \cup \{r\}$ is not larger than any pebble in T_{j+1}. We do this in three steps. First we make T_j' into a max-heap with $\mathsf{Heap}^{\uparrow}(T_j')$. [2] Second we make T_{j+1} into a min-heap with $\mathsf{Heap}^{\downarrow}(T_{j+1})$, which is just a minor variation of the first procedure. Third we use the procedure $\mathsf{Swap}(T_j', T_{j+1}, k)$ which moves, through r, the smallest pebbles of T_{j+1} into T_j', while moving the largest pebbles of T_j' into T_{j+1}. The parameter k is used to bound the number of pebbles that move between trees in Swap.

We describe these three steps in line 6 in the context illustrated in Figure 11.2, where the two subtrees T_i and T_j are connected via r, for arbitrary i and j.

Lemma 12. The $\mathsf{OddEvenTreeSort}(T, r)$ procedure is correct.

Proof. We begin by assuming $k = n$ and then show the smaller k large enough. We also assume the correctness of Heap and Swap, which are shown below. Therefore the (sequence in step 6 in Algorithm 2) $\mathsf{Heap}^{\uparrow}(T_j')$; $\mathsf{Heap}^{\downarrow}(T_{j+1})$; $\mathsf{Swap}(T_j', T_{j+1}, k)$ will have the effect of moving the largest k pebbles into T_{j+1} from amongst the pebbles originally in T_j, T_{j+1} and r. Now consider when $i = 1$ and recall the sizes are nonincreasing. As j increases it is

[2] A textbook (max) heap, maintains the heap-ordering property of having no child larger than its parent..

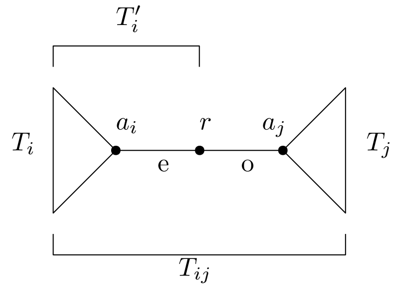


Figure 11.2: Pair of subtrees joined at the root r. The e/o labels indicate parity.

inductively true that T_{j+1} will contain the largest n_{j+1} pebbles from amongst the pebbles originally in $T_1, T_2, \ldots T_{j+1}$ and r. So after $d-1$ iterations, T_d already contains the correct set of pebbles, in some order. Similarly when $i = 2$ after $d-2$ iterations T_{d-1} will contain the correct set of pebbles. And so on.

But when k is limited Swap does less work. We need to show that for any i and j that only the n_{i+j} smallest pebbles from T_{j+1} might need to move into T_j'; the remainder need not even be considered. We prove this by induction on i and on j. When $i = 1$ clearly $n_{i+j} = n_{j+1}$ is enough for all j, since that is the size of T_{j+1}. Consider the pass with $i = l+1$ and assume that during pass l only n_{l+j} needed to pass from T_{j+1} to T_j'. On the previous pass, when $j = 1$, T_2 gave up to T_1' every pebble it needed to, T_3 then dumped at most n_{l+2} new pebbles into T_2. So at most $n_{(l+1)+1}$ will need to be given up from T_2 to T_1' during pass $l+1$. As j increases, holding $i = l+1$, the inductive proof continues to hold. □

To explain $\mathsf{Heap}^{\uparrow}(T_j')$ we first explain a purely sequential algorithm for max-heap construction. This is not the normal textbook approach since there are faster sequential algorithms. We label the nodes in T_j' according to a breadth-first (BF) order, with the root labeled 0. The heap building proceeds in a sequential manner visiting the nodes in BF order, starting at node 1. When we visit node i we let it bubble up the path to the root, continuing to swap with its parent if it is larger, or until it reaches the root. It is invariant that the subtree containing the nodes 0 up to i will form a max-heap. To parallelize this we will pipeline the bubble-up processes.

The $\mathsf{Heap}^{\uparrow}(T'_j)$ procedure works as follows; the min-heap version is symmetric. When we visit node i we initiate a sequence of matched edges, going up the path to the root. If node $i+1$ is on the same level as node i then node $i+1$ does not initiate its traversal of the path to the root until it has idled for one stage. If node $i+1$ is on the next level then node $i+1$ does not idle before initiating its path. This insures that when node $i+1$ uses an edge on, say, level l then node i will already be using an edge on level $l-2$; similarly progress up one path will be pipelined with progress up another path. It follows that it is invariant that:

(a) at each level of the tree, during any stage, there is at most one edge being matched, and

(b) there are never two adjacent levels containing matched edges.

Therefore matched edges in all these paths will always be disjoint.

Lemma 13. The $\mathsf{Heap}^{\uparrow}(T_j)$ procedures is correct and requires at most $3n_j$ stages. Similarly for $\mathsf{Heap}^{\downarrow}(T_j)$.

Proof. The correctness would follow from the correctness of the sequential algorithm except for work up one path may interfere with work up another path. Consider the pebbles starting up from nodes i and j, $i < j$. Due to the pipelining, the pebble starting at i will stay at a level above the pebble from j as long as it is moving up its path P_i. If it stops moving it is because it is (by induction) less than all the pebbles above it on P_i. Now suppose the pebble from j moves into the path P_i (at node k, the least common ancestor of i and j). If it moves into P_i it will only be because it is replacing a pebble on P_i by a larger pebble. If this new pebble happens to be larger than its parent in P_i, the pebble will immediately repair that by continuing up the remainder of P_i since P_i and P_j coincide between node k and the root.

Note that it follows inductively that the node i initiates its pebble's upward movement after at most $2i$ stages, and that it moves up without any further delays until it stops. The run-time follows since the final initiation of a path is from the node $n_j - 1$ and the final path is finished before n_j additional stages have passed, because every path length is bounded by n_j. □

The $\mathsf{Swap}(T_i, T_j k)$ procedure works as follows. Note that in Figure 11.2 the edge (a_1, r) is marked as even, while the edge (r, a_2) is marked as odd. From these we can designated every edge in T_{ij}, in alternating layers,as odd or even. Assume that a_2 contains the smallest pebble in T_j and r contains the largest in T'_i. The two roots of T'_i and T_j, r and a_j are connected by an odd edge. We call (r, a_2) the *crossing edge* and it is oriented so that r receives the minimum. A single compare-swap on the crossing edge is understood as doing:

(a) an Extractmax from T'_i followed by an insert-at-root for T_j, and

(b) an Extractmin from T_j followed by an insert-at-root for T'_i.

These are the standard textbook heap operations. Another textbook operation, HEAPIFY, restores the heap-ordering property to any max-heap after the pebble at its root, say t, is changed (by a swap simulating an insert-at-root operation). The normal sequential algorithm works by finding the root t' of its child subtree with largest pebble. If the pebble at t' exceeds the pebble at t then t and t' are swapped and recursively we HEAPIFY the subtree rooted at t' (since it has a new pebble at its root). (Min-heaps are, again, symmetric.)

We will implement HEAPIFY on T_i in a parallel fashion by fanning-out through the tree. Since our algorithm must be oblivious we will HEAPIFY all the subtrees of r in parallel, after sequentially giving every child of r a chance to swap with r, if it is larger. Unlike the sequential algorithm the root r can be replaced (increased) several times while polling its children. As a result several subtrees can have a new (smaller) pebble at the root. But we will recursively HEAPIFY all of r's subtrees so it does not matter which subtrees have new roots. For the second level all of r's children will, in parallel, sequentially match with their children at the next level. Recall β_i is the largest number of children out-degree (out-degree or arity) of any node in T_i. If we force the matching of parents to children across one level to take β_i stages then every parent will have enough time to talk to every one of its children;

if a parent has $\alpha < \beta_i$ children it will idle for $\beta_i - \alpha$ stages. (We can easily reduce the idling but it will have no effect on the worst-case analysis.) We continue descending, doing one level at a time. Because of the fanning-out the new pebble at the root could take any path towards a leaf, in the same way that the sequential algorithm can. The correctness of this algorithm is clear from its recursive description. However the external observer watching it execute, sees a wave of fan-outs descending one level at a time, spending β_i time at each level.

Our $\mathsf{Swap}(T_i', T_j k)$ method (see Algorithm 3) connects the ideas above. During the first stage it will use the crossing edge and repeat that every 2β time steps thereafter; where $\beta = max(\beta_i, \beta_j)$. This swap will initiate both a max-HEAPIFY in T_i' and a min-HEAPIFY in T_j. As we have seen above both operations will be implemented as a wave of fan-outs on subsequent levels, going away from r and a_2. In particular it will use β stages on the first even level, then β stages on the next odd level, then that many on the next even level, and so on down to the leaves.

However note that the crossing edge will swap again during stage $1 + 2\beta$ which will initiate two new HEAPIFYs. The reason that this can be done before the previous HEAPIFYs are finished is that the new HEAPIFYs will initiate new waves that will follow the previous waves in a pipelined way. Recall that all layers will be synchronized since each will take exactly β stages.

Algorithm 3: $\mathsf{Swap}(T_i', T_j, k)$

```
Input  : max-heap $T_i'$ and min-heap $T_j$ and parameter $k$
Output: $T_j$ contains the largest pebbles and heap-ordering is restored
1 begin
2     $\beta = max(\beta_i, \beta_j)$
3     repeat $2k$ times do
4         in $\beta$ stages compare-swap every odd edge once
5         in $\beta$ stages compare-swap every even edge once
```

While the algorithm appears to be related to the odd-even transposition sort, it is actually understood to be a sequence of k pairs of HEAPIFYs executing in a pipelined fashion. The last pair of HEAPIFYs are initiated from r and a_2 after $2k\beta + 1$ stages and they will be finished after less than $2k\beta$ more stages. Recall that we have already proven that when the algorithm is invoked that k will be larger than the number of pebbles that need to leave T_j and enter T_i', and vice versa.

Lemma 14. The procedure $\mathsf{Swap}(T_i, T_j; k)$ is correct, and uses $4k\beta$ time steps.

Proof. Note the we have argued that the HEAPIFY procedures are correct. Further it is clear that subsequent HEAPIFYs in the same tree can be pipelined since they are all using even levels simultaneously followed by all using odd levels. Also, doing compare-swaps at some level not needed by HEAPIFY is of no concern since all such comparisons will never result in a swap. (The algorithm could do fewer operations but, again, it has no effect on the worst-case run-time.)

By the definition and correctness of the HEAPIFY operations, it follows that the k smallest pebbles can be transferred over to T_j'. Of course, if fewer than k pebbles need to move the subsequent compares will not result in swaps. Similarly for the k largest pebbles in T_i'. Also note that pipelining will never be needed at a level deeper than level k in either tree, since no new pebbles will ever intrude that deep. (As written the algorithm might do useless operations at deeper levels.) So the last HEAPIFYs initiated by the crossing edge need only propagate for k levels. □

Lemma 15. The bound on the total number of stages for the first phase is

$$S(T, r) = O(\min(\Delta^2 n, n^2))$$

Proof. Recall the first phase has $d - 1$ passes. Let $\alpha_j = \max(\beta_j, \beta_{j+1})$. Recall the total number of stages in the i^{th} pass is

$$c_i \leq \sum_{j=1}^{d-i} (3n_j + 3n_{j+1} + 4\alpha_j n_{i+j}). \tag{11.2}$$

So the total number of stages during the first phase is

$$\begin{aligned}
\sum_{i=1}^{d-1} c_i &\leq \sum_{i=1}^{d-1}\sum_{j=1}^{d-i} (3n_j + 3n_{j+1} + 4\alpha_j n_{i+j}) && (11.3)\\
&\leq 6\Delta n + 4\sum_{j=1}^{d-1}\sum_{i=1}^{d-j} \alpha_j n_{i+j} && (11.4)\\
&\leq 6\Delta n + 4\sum_{j=1}^{d-1}\alpha_j \sum_{i=1}^{d-j} n_{i+j} && (11.5)\\
&\leq 6\Delta n + 4n\sum_{j=1}^{d-1}\alpha_j && (11.6)\\
&\leq 6\Delta n + 4n\min(\Delta^2, 2n) && (11.7)
\end{aligned}$$

The last inequality follows from the fact that $\sum_{j=1}^{d-1}\alpha_j \leq 2\sum_{j=1}^{d-1}\beta_j < 2n$ and $\sum_{j=1}^{d-1}\alpha_j \leq \Delta^2$, since $d \leq \Delta$ and $\alpha_j \leq \Delta$ for all j. □

Putting this upper bound of $S(T, r)$ in Equation 11.1 we get a simplified recurrence:

$$C(n) \leq C(n/2) + O(\min(\Delta^2 n, n^2)) \tag{11.8}$$

This shows that OddEvenTreeSort requires $O(\min(\Delta^2 n, n^2))$ stages to correctly sort any input with respect to the MP ordering. This is a good bound on the sorting number for trees, when Δ is small.

Chapter 12: Sorting Network On A Pyramid

In this chapter we propose a sorting network for the pyramid graph. A 1-dimensional pyramid with m-levels is defined as the complete binary tree of $2^m - 1$ nodes, where the nodes in each level are connected by a path (an one-dimensional mesh). We assume the apex (root) to be at level 0, and subsequent levels are numbered in ascending order. A 2-dimensional pyramid is shown in Figure 12.1. In this case each level l is a $2^l \times 2^l$ square mesh. Similarly a d-dimensional pyramid having m levels is denoted by $\triangle_{m,d}$, where the level l is a d-dimensional regular mesh of length 2^l in each dimension. Clearly, the size of layer l is $|M_l| = n_l = 2^{ld}$ and the number of vertices in the graph is $N = |\triangle_{m,d}| = \sum_{l=0}^{m-1} 2^{ld} = \frac{2^{md}-1}{2^d-1}$. We treat a vertex $x \in M_l$ as a vector in $[1, 2^l]^d$ which denotes its position on the mesh.

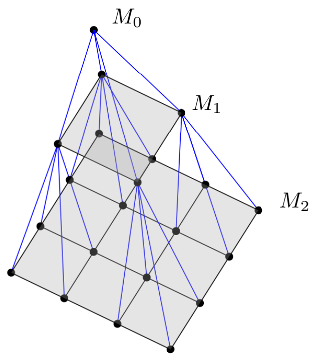


Figure 12.1: A pyramid $\triangle_{3,2}$ in 3-dimension

Next we prove an upper bound on $st(\triangle_{m,d})$. In order to derive this bound we first need to prove the following bound on the routing number of pyramid.

Lemma 16. The routing number for a pyramid $rt(\triangle_{m,d}) = O(dN^{1/d})$.

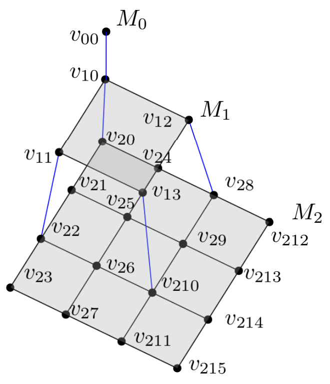


Figure 12.2: The graph $\triangle'_{3,2}$ after stripping way edges from $\triangle_{3,2}$

Proof. Given the pyramid $\triangle_{3,2}$ consider a subgraph $\triangle'_{3,2}$ as shown in Figure 12.2. In the literature this subgraph is sometimes referred to as a "multi-grid", see for example [61]. We get this subgraph as we move down from the apex we remove all but the "first" edge from the set of edges that connects a vertex to its neighbors in the level below. The surviving edges that connects two adjacent layers will be referred to as vertical edges. These edges can be grouped into disjoint vertical paths as shown by the blue lines in Figure 12.2. The above construction naturally generalizes to higher dimensions. Clearly $\mathscr{rt}(\triangle_{m,d}) \leq \mathscr{rt}(\triangle'_{m,d})$ where $\triangle'_{m,d}$ is the multi-grid obtained from $\triangle_{m,d}$. We only need to show $\mathscr{rt}(\triangle'_{m,d}) = O(dN^{1/d})$.

Let π be some input permutation. Without loss of generality we assume that π consists only of 2-cycles or 1-cycles. From [36] we know that any arbitrary permutation can be written as a composition of at most two such permutations. This follows from the fact that we can route any complete graph in two steps (see Chapter 8). In order to route π we first route the pebbles into their appropriate levels and then route within these levels. Routing consists of five rounds where in the odd numbered rounds we route within the levels and in the even numbered rounds we use the vertical paths to route between the levels. The first four rounds will move the pebbles to their appropriate destination level.

Let v_{ij} be the j^{th} node at level i, where $j \in [0, n_i - 1]$. Let ϕ_k be the number of maximal vertical paths of length k. For example, in Figure 12.2 we have $\phi_2 = 1$ and $\phi_1 = 3$. In general in a $\triangle'_{m,d}$, $\phi_k = n_{m-k-1} - n_{m-k-2}$ for $k \in [1, m-2]$ and $\phi_{m-1} = 1$. We group the cycles in π based on their source and destination level (in case of 1-cycles the source and destination levels are the same). Let P_{ij} $(i < j)$ be the set of pebble pairs that need to be moved from level i down to level j and vice-versa and P_{ii} be the set of pebbles that stay in level i. Let $\mu_{ij} = |P_{ij}|$. Let $P_i = \bigcup_{i<j} P_{ij}$ be the set of pebble pairs that move a pebble up to level i. We shall only use disjoint vertical paths of length $m-i-1$ to route the pebbles in P_i. During a round of inter-level routing only paths of length $m-i-1$ will be used, for some j, to swap two pebbles between levels i and j; all other pebbles on that path will not move. As an example consider the case in Figure 12.2. Suppose $\pi(v_{00}) = v_{21}$. Then during the intra-level routing on the first round we will move the pebble at v_{21} to v_{20}. All intermediate nodes on this path, which in this case is just v_{10} will be ignored (i.e., a pebble on these nodes will return to their original position at the end of the round). The four pebbles $\{v_{10}, v_{11}, v_{12}, v_{13}\}$ will only use the three paths of length 1 to move to the bottom level (if necessary). In general $|P_i| = \sum_{j>i} \mu_{ij} \leq n_i \leq 2(n_i - n_{i-1}) = 2\phi_{m-i-1}$. Hence we need at most two rounds of inter-level routing along these vertical paths to move all pebbles in P_i.

Routing within the levels (which happens in parallel) is dominated by the routing number of the last level which is known to be $O(dn_m^{1/d}) = O(dN^{1/d})$ (for example, we can use Corollary 2 of Theorem 4 in [36]). Hence the three odd rounds take $O(dN^{1/d})$ in total. In the even rounds routing happens in parallel along the disjoint vertical paths. The routing time in this case is $O(m)$. Since, $N^{1/d} = \Omega(2^m)$, the even rounds do not contribute to the overall routing time, which remains $O(dN^{1/d})$, as claimed. □

Using the above theorem we give an upper bound on the sorting number of the pyramid.

Theorem 12.1 (Pyramid)**.** The sorting number of a pyramid $\triangle_{m,d}$, $\mathfrak{st}(\triangle_{m,d}) = O(d\, N^{1/d})$.

Proof. Let $\triangle_{i,d}$ denote the sub-pyramid from level 0 to level i and let M_i be the d-dimensional mesh at level i. Let π_i be some ordering of the mesh M_i. Note that $\pi_i : [1, 2^i]^d \to [n_i]$ is a bijection and π_0 is the identity permutation. Next we define a sorted order π for the pyramid $\triangle_{m,d}$ based on the π_i's; in the permutation π we assume the layers are ordered among themselves in ascending order starting from the apex. So the vertex labeled (with respect to M_i) i on layer j has a global rank $\pi(i) = \pi_j(i) + |\triangle_{j-1,d}|$. Recall that $\mathcal{st}(M_i) = O(dn_i^{1/d})$ which is due to Kunde[57] where he used a π_i that was a general snake-like ordering. From Lemma 10 we see that this bound still holds if we replace the snake-like ordering with some arbitrary permutation. In this case $\mathcal{rt}(M_i) = \Theta(\mathcal{st}(M_i))$. Next we describe the sorting network $\mathcal{S}(\triangle_{m,d}, \mathcal{M}, \pi)$ by specifying the matchings as follows. $\mathcal{S}(\triangle_{m,d}, \mathcal{M}, \pi)$

1. Route all pebbles of $\triangle_{m-1,d}$ to M_{m-1} and sort them using the mesh M_{m-1}.

2. Route these pebbles back to $\triangle_{m-1,d}$ such that they are in sorted order (according to π).

3. Sort the mesh M_{m-1} according to π_{m-1}.

4. Route a pebble of rank $i \le n_{m-2}$ at position $x_i \in M_{m-1}$ to $y_i \in M_{m-1}$ where

$$y_i[j] = 2\pi_{m-2}^{-1}(n_{m-2} + 1 - i)[j] - 1$$

   Let $Y = (y_1, \ldots, y_{n_{m-2}})$.

5. Merge Y with M_{m-2} using pairwise compare-exchanges, where y_i is compared with $z \in M_{m-2}$ such that $\pi_{m-2}(z) = i$.

6. Repeat 1-5 once.

7. Repeat 1-3 once.

Depth

Note that the number of times we route on $\triangle_{m,d}$ is 6. Also sorting on the mesh M_{m-1} occurs 6 times. We know that both routing and sorting on a mesh takes $O(dN^{1/d})$ steps. From Lemma 16 we see that routing on $\triangle_{m,d}$ also takes $O(dN^{1/d})$ steps. So the total contribution of all the steps except 4 and 5 is $O(dN^{1/d})$. It is easy to see that step 4 also takes $O(dN^{1/d})$ as it just routes a permutation on the mesh M_{m-1}. And step 5 can be accomplished in constant time. Putting it all together we see that $\mathfrak{st}(\triangle_{m,d}) = O(dN^{1/d})$ as claimed.

Correctness

Here we show $\mathcal{S}(\triangle_{m,d}, \mathscr{M}, \pi)$ is a sorting network. Clearly the algorithm is oblivious. Therefore, by invoking the 0-1 principle [51] we may assume that our pebbles are all 1's and 0's. Before the execution of step 7 if every pebble in $\triangle_{m-1,d}$ is smaller than every pebble in M_{m-1} then after step 7 we shall have our desired sorted order. Otherwise there must be some pebbles $x \in M_{m-1}$ that is supposed to be in $\triangle_{m-1,d}$. If that is the case then x must be a 0, otherwise x is greater than or equal to every pebble in $\triangle_{m-1,d}$ and we are done. Now look at step 4 and 5. In step 4 we route the set of n_{m-2} smallest pebbles in M_{m-1} such that the i^{th} smallest pebble is at some vertex of M_{m-1} which is directly connected to the vertex in M_{m-2} that has the i^{th} largest pebble of M_{m-2}. Since x was not exchanged during both the iteration of step 4 and 5 then x must be larger than at least $2n_{m-2}$ elements in $\triangle_{m,d}$, but then x should not belong to $\triangle_{m-1,d}$ (since $|\triangle_{m-1,d}| \leq 2n_{m-2} - 1$ for any d) contradicting our assumption. Hence, after step 6 we see that all pebbles of $\triangle_{m-1,d}$ must be smaller than every pebble of M_{m-1} hence sorting these pebbles independently in the final step gives the desired sorted order. $\square$

Bibliography

[1] D. Knuth, "The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching," *MA: Addison-Wesley*, vol. 30, 1968.

[2] J. Kahn and M. Saks, "Balancing poset extensions," *Order*, vol. 1, no. 2, pp. 113–126, 1984.

[3] D. E. Knuth, "The art of computer programming. vol. 1: Fundamental algorithms. second printing," 1969.

[4] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 110–127, 1979.

[5] S. Kannan and S. Khanna, "Selection with monotone comparison costs," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2003, pp. 10–17.

[6] M. Charikar, R. Fagin, V. Guruswami, J. M. Kleinberg, P. Raghavan, and A. Sahai, "Query strategies for priced information," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 785–819, 2002. [Online]. Available: http://dx.doi.org/10.1006/jcss.2002.1828

[7] A. Gupta and A. Kumar, "Sorting and selection with structured costs," in *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, 2001, pp. 416–425. [Online]. Available: http://dx.doi.org/10.1109/SFCS.2001.959916

[8] W. Goddard, C. Kenyon, V. King, and L. J. Schulman, "Optimal randomized algorithms for local sorting and set-maxima," *SIAM J. Comput.*, vol. 22, no. 2, pp. 272–283, 1993. [Online]. Available: http://dx.doi.org/10.1137/0222020

[9] A. Biswas, V. Jayapaul, and V. Raman, "Improved bounds for poset sorting in the forbidden-comparison regime," in *Conference on Algorithms and Discrete Applied Mathematics.* Springer, 2017, pp. 50–59.

[10] S. Chatterji, "The number of topologies on n points, kent state university," *NASA Technical Report*, 1966.

[11] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin, "Sorting and selection in posets," *SIAM J. Comput.*, vol. 40, no. 3, pp. 597–622, 2011. [Online]. Available: http://dx.doi.org/10.1137/070697720

[12] U. Faigle and G. Turán, "Sorting and recognition problems for ordered sets," *SIAM J. Comput.*, vol. 17, no. 1, pp. 100–113, 1988. [Online]. Available: http://dx.doi.org/10.1137/0217007

[13] J. Cardinal and S. Fiorini, "On generalized comparison-based sorting problems," in *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, 2013, pp. 164–175. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40273-9_12

[14] N. Alon, M. Blum, A. Fiat, S. Kannan, M. Naor, and R. Ostrovsky, "Matching nuts and bolts," in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia.*, 1994, pp. 690–696. [Online]. Available: http://dl.acm.org/citation.cfm?id=314464.314673

[15] J. Komlós, Y. Ma, and E. Szemerédi, "Matching nuts and bolts in o(n log n) time," *SIAM J. Discrete Math.*, vol. 11, no. 3, pp. 347–372, 1998. [Online]. Available: http://dx.doi.org/10.1137/S0895480196304982

[16] J. E. Savage, *Models of computation - exploring the power of computing.* Addison-Wesley, 1998.

[17] M. Akra and L. Bazzi, "On the solution of linear recurrence equations," *Comp. Opt. and Appl.*, vol. 10, no. 2, pp. 195–210, 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1018373005182

[18] A. C.-C. Yao, "Probabilistic computations: Toward a unified measure of complexity," in *Foundations of Computer Science, 1977., 18th Annual Symposium on.* IEEE, 1977, pp. 222–227.

[19] Z. Huang, S. Kannan, and S. Khanna, "Algorithms for the generalized sorting problem," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, 2011, pp. 738–747. [Online]. Available: http://dx.doi.org/10.1109/FOCS.2011.54

[20] S. Angelov, K. Kunal, and A. McGregor, "Sorting and selection with random costs," in *LATIN 2008: Theoretical Informatics, 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008, Proceedings*, 2008, pp. 48–59. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78773-0_5

[21] M. E. Dyer, A. M. Frieze, and R. Kannan, "A random polynomial time algorithm for approximating the volume of convex bodies," *J. ACM*, vol. 38, no. 1, pp. 1–17, 1991. [Online]. Available: http://doi.acm.org/10.1145/102782.102783

[22] M. Ajtai, J. Komlós, W. Steiger, and E. Szemerédi, "Almost sorting in one round," *Randomness and Computation*, vol. 5, pp. 117–125, 1989.

[23] B. Bollobás and M. Rosenfeld, "Sorting in one round," *Israel Journal of Mathematics*, vol. 38, no. 1-2, pp. 154–160, 1981.

[24] B. Bollobás and G. R. Brightwell, "Transitive orientations of graphs," *SIAM J. Comput.*, vol. 17, no. 6, pp. 1119–1133, 1988. [Online]. Available: http://dx.doi.org/10.1137/0217072

[25] B. Bollobás and G. Brightwell, "Graphs whose every transitive orientation contains almost every relation," *Israel Journal of Mathematics*, vol. 59, no. 1, pp. 112–128, 1987.

[26] R. L. Graham, A. C. Yao, and F. F. Yao, "Information bounds are weak in the shortest distance problem," *Journal of the ACM (JACM)*, vol. 27, no. 3, pp. 428–444, 1980.

[27] W. Goddard, C. Kenyon, V. King, and L. J. Schulman, "Optimal randomized algorithms for local sorting and set-maxima," *SIAM Journal on Computing*, vol. 22, no. 2, pp. 272–283, 1993.

[28] A. Bar-Noy, R. Motwan, and J. Naor, "A linear time approach to the set maxima problem," *SIAM Journal on Discrete Mathematics*, vol. 5, no. 1, pp. 1–9, 1992.

[29] R. Desper, "The set-maxima problem: an overview," *Master's thesis, Rutgers University*, 1994.

[30] J. N. Komlós, "Linear verification for spanning trees," in *Foundations of Computer Science, 1984. 25th Annual Symposium on.* IEEE, 1984, pp. 201–206.

[31] V. Liberatore, "Matroid decomposition methods for the set maxima problem," in *SODA*, 1998, pp. 400–409.

[32] R. E. Tarjan, "Sensitivity analysis of minimum spanning trees and shortest path trees," *Information Processing Letters*, vol. 14, no. 1, pp. 30–33, 1982.

[33] O. Aichholzer, W. Mulzer, and A. Pilz, "Flip distance between triangulations of a simple polygon is np-complete," *Discrete & Computational Geometry*, vol. 54, no. 2, pp. 368–389, 2015.

[34] S. Even and O. Goldreich, "The minimum-length generator sequence problem is np-hard," *Journal of Algorithms*, vol. 2, no. 3, pp. 311–313, 1981.

[35] M. R. Jerrum, "The complexity of finding minimum-length generator sequences," *Theoretical Computer Science*, vol. 36, pp. 265–289, 1985.

[36] N. Alon, F. R. Chung, and R. L. Graham, "Routing permutations on graphs via matchings," *SIAM journal on discrete mathematics*, vol. 7, no. 3, pp. 513–530, 1994.

[37] I. Benjamini, I. Shinkar, and G. Tsur, "Acquaintance time of a graph," *SIAM Journal on Discrete Mathematics*, vol. 28, no. 2, pp. 767–785, 2014.

[38] T. Miltzow, L. Narins, Y. Okamoto, G. Rote, A. Thomas, and T. Uno, "Approximation and hardness for token swapping," *arXiv preprint arXiv:1602.05150*, 2016.

[39] K. Yamanaka, E. D. Demaine, T. Ito, J. Kawahara, M. Kiyomi, Y. Okamoto, T. Saitoh, A. Suzuki, K. Uchizawa, and T. Uno, "Swapping labeled tokens on graphs," *Theoretical Computer Science*, vol. 586, pp. 81–94, 2015.

[40] J. Kawahara, T. Saitoh, and R. Yoshinaka, "The time complexity of the token swapping problem and its parallel variants," *arXiv preprint arXiv:1612.02948*, 2016.

[41] L. Zhang, "Optimal bounds for matching routing on trees," *SIAM Journal on Discrete Mathematics*, vol. 12, no. 1, pp. 64–77, 1999.

[42] W. T. Li, L. Lu, and Y. Yang, "Routing numbers of cycles, complete bipartite graphs, and hypercubes," *SIAM Journal on Discrete Mathematics*, vol. 24, no. 4, pp. 1482–1494, 2010.

[43] S. Micali and V. V. Vazirani, "An O($\sqrt{|V|}$|E|) algorithm for finding maximum matching in general graphs," in *Foundations of Computer Science, 1980., 21st Annual Symposium on.* IEEE, 1980, pp. 17–27.

[44] A. Gyárfás, M. Ruszinkó, G. N. Sárközy, and E. Szemerédi, "Partitioning 3-colored complete graphs into three monochromatic cycles," *Electronic J. of Combinatorics*, vol. 18, no. 1, 2011.

[45] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy, "Interactive proofs and the hardness of approximating cliques," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 268–292, 1996.

[46] L. Engebretsen and J. Holmerin, "Clique is hard to approximate within n 1-o (1)," in *International Colloquium on Automata, Languages, and Programming.* Springer, 2000, pp. 2–12.

[47] U. Feige, "Approximating maximum clique by removing subgraphs," *SIAM Journal on Discrete Mathematics*, vol. 18, no. 2, pp. 219–225, 2004.

[48] R. Boppana and M. M. Halldórsson, "Approximating maximum independent sets by excluding subgraphs," *BIT Numerical Mathematics*, vol. 32, no. 2, pp. 180–196, 1992.

[49] R. Diestel, "Graph theory. pdf," 2005.

[50] M. Hall, *Combinatorial theory.* John Wiley & Sons, 1998, vol. 71.

[51] D. Knuth, *The art of computer programming: Sorting and searching*, ser. The Art of Computer Programming. Addison-Wesley, 1998. [Online]. Available: https://books.google.com/books?id=wRgZAQAAIAAJ

[52] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(nlogn)$ sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing.* ACM, 1983, pp. 1–9.

[53] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* ACM, 1968, pp. 307–314.

[54] T. Leighton and C. G. Plaxton, "Hypercubic sorting networks," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 1–47, 1998.

[55] C. G. Plaxton and T. Suel, "A super-logarithmic lower bound for hypercubic sorting networks," in *International Colloquium on Automata, Languages, and Programming.* Springer, 1994, pp. 618–629.

[56] C. P. Schnorr and A. Shamir, "An optimal sorting algorithm for mesh connected computers," in *Proceedings of the eighteenth annual ACM symposium on Theory of computing.* ACM, 1986, pp. 255–263.

[57] M. Kunde, "Optimal sorting on multi-dimensionally mesh-connected computers," in *Annual Symposium on Theoretical Aspects of Computer Science.* Springer, 1987, pp. 408–419.

[58] O. Angel and I. Shinkar, "A tight upper bound on acquaintance time of graphs," *Graphs and Combinatorics*, vol. 32, no. 5, pp. 1667—-1673, 2016, arXiv:1307.6029.

[59] I. Shinkar, "Private communications," 2016.

[60] L. Babai and M. Szegedy, "Local expansion of symmetrical graphs," *Combinatorics, Probability and Computing*, vol. 1, no. 01, pp. 1–11, 1992.

[61] F. T. Leighton, *Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes.* Elsevier, 2014.

Curriculum Vitae

Avah Banerjee graduated with honors from Bengal Engineering College & Model School, Howrah, India in 2004. He went on to study Electrical Engineering at National Institute of Technology, Durgapur, India, where he was awarded a Bachelor of Technology (Hons.) in 2009. After graduation he worked primarily in the field of computer vision. He was employed at Videonetics Technology Pvt. Ltd. and subsequently to TechBLA Corp. Pvt. Ltd. from 2009 to 2012 as a software engineer. During this time he worked with Dr. Prasun Das (SQC-OR) and Dr. Sanghamitra Bandyopadhyay (Machine Learning Unit) at Indian Statistical Institute (Kolkata) on evolutionary multi-objective optimization. He received his Masters in Computer Science from George Mason University in 2015 and subsequently his PhD in 2018. His research focuses on graph algorithms and computational geometry.