Chennai Mathematical Institute

SORTING AND SELECTION IN RESTRICTED MODELS

A Thesis in
Department of Computer Science
by
Varunkumar Jayapaul

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

October 2017

Abstract

Sorting and selection problems have always been fundamental questions which have constantly been studied. In this thesis, several questions regarding sorting and selection are looked at in restricted models motivated by theoretical and practical considerations.

In Chapter 1, we look at problems of sorting, where the algorithm may not have complete freedom on what queries it can make. The adversary declares that certain queries will not be answered by it, but the algorithm may be able to deduce the result of that query using transitivity. In these circumstances, we give an algorithm which improves the previous upper bound on the queries required and we also give the first lower bound on the number of queries which are required. A prelimnary version of the results in this chapter appear in proceedings of CALDAM 2017 [1].

In Chapter 2, we look at sorting and selection problem, when the oracle cannot answer the relation between two elements, but can answer the difference in their ranks. In this scenario the problem of sorting is equivalent to the problem of graph reconstruction (where the graph is a simple path). To complicate the matter, the adversary can also decide which queries it will answer. We show that the problem has the same complexity, regardless of whether there are restrictions on pairs of elements between which a query can be made.

In Chapter 3, we look at problems of sorting and selection, where the adversary/oracle can reveal only whether the elements being compared are equal or not. In these circumstances, we study the question of finding a mode and organizing all the elements in the input, so that all the equal elements are grouped together. We give optimal lower bounds and matching upper bounds for finding a mode and for sorting in this model. We generalize a previously known technique for finding the majority element, and use it to find a mode. Results of this chapter appeared in proceedings of WADS 2015 and WALCOM 2016 [2, 3].

In Chapter 4, we look at the selection problem, when the oracle has some freedom to lie. The oracle has this freedom, only when the elements being compared have their difference below a fixed threshold. In these situations, it is known that finding a maximum element is equivalent to finding a king vertex in a tournament.

A king vertex is a vertex which can reach the remaining vertices in the graph, by a directed path of length at most two. We give an algorithm for finding such a king, in the incremental dynamic setting. We also show the hardness of verifying a king vertex. We also generalize some previously known results for finding kings and show that the adversarial strategy needs to be changed, if a better lower bound has to be shown. In other words we construct an algorithm which works specifically against this adversary and ensures that it finds a king and the algorithm's runtime matches the best known lower bound. Results of this chapter appeared in proceedings of FAW 2017 [4].

In Chapter 5, we look at the selection problem in directed graphs, undirected graphs and tournaments, which may not have any transitivity property. In this circumstances, the goal is to find vertices having a certain degree or the maximum degree. We improve the previously known results for these questions. Results of this chapter appeared in proceedings of CALDAM 2017 [5].

Table of Contents

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Venkatesh Raman for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee for their insightful comments and encouragement. Chennai Mathematical Institute gave me complete freedom in the choices I made and also ensured that my stay with the institute was a cherishable ride.

I would also like to express my gratitude to Dr. Srinivasa Rao Satti, whose unplanned visits to IMSc would ultimately end in wonderful ideas and papers. He has been a great source of inspiration for me.

My sincere thanks also goes to the Institute of Mathematical Sciences that provided me an opportunity to use their institute resources, and gave access to the library and research facilities. Without their precious support it would not be possible to conduct this research.

I would also like to express my gratitude to my colleagues Arindam Biswas, and Dishant Goyal, with whom I had the opportunity to collaborate and write papers.

I thank my fellow colleagues Niranka Banerjee, Sankardeep Chakraborty, Anish Mukherjee, for the stimulating discussions and for all the fun we have had in the last four years. I would also like to thank my juniors at Chennai Mathematical Institute.

Last but not the least, I would like to thank my parents for supporting me spiritually throughout writing this thesis and my life in general.

Dedication

To Menakha Jayapaul, Jayapaul Narayanswamy and Lord Shiva

Chapter 1

Introduction

Sorting and selection problems have always been the under constant attention of researchers across the computer science research community. These problems have been a gateway for students who indulge in the world of algorithms. Over the past several decades, various improvements have been made to the previously existing algorithms for sorting and selection. Every parameter related to the efficacy of an algorithm for sorting has been fine tuned over the past decades. They have been looked through the lens of several branches of computer science. Researchers working on parallel algorithms have designed parallel algorithms for faster sorting. People concerned with space requirements of the algorithm, have succeeded in reducing the additional workspace required for a sorting algorithm.

A significant portion of this advancement is based on certain assumptions regarding the instance of the problem or the ability of the oracle. In this thesis, we take a look at those hidden assumptions and see what happens if these assumptions may not apply to a particular sorting or selection problem. A common assumption made by most of sorting algorithms presented across all graduate level textbooks, is that the algorithm has the freedom to make a comparison between any pair of elements. But several scenarios arise where such an assumption may not be valid. Suppose there are n teams, and we need to rank the teams according to their level of skill. If all teams are willing to compete against any other team, then the problem boils down a traditional sorting problem. But suppose several teams have some internal rivalry or they are not willing to play against certain teams which they believe resort to unfair means. In such a scenario, traditional

algorithms may fail to rank them according to their skill.

Another common assumption made in regard to sorting/selection problems is that when two elements a and b are compared, the oracle is assumed to output whether $a \leq b$ or $a > b$. But there exist several scenarios where the oracle only has the ability to tell whether $a = b$ or not. In these types of problems, the goal of a sorting algorithm would be to keep all elements which are equal in value together. Consider the case of a folder which has several images. Most computer softwares are only capable of telling whether two images match or not, thus revealing whether the folder contains any duplicate images or not. These algorithms are unable to define a total ordering between two distinct images, unless given a complex method to define such an ordering to measure relative similarity.

Sometimes the oracle may not be correctly able to determine the winner if the two elements being compared have an insignificant amount of difference. Especially if the oracle resembles a human then such instances can occur quite frequently. For example, it is easier for a child to decide that a tennis ball is smaller than a basketball, but a child may not be able to decide correctly which one among a basketball and a football is bigger.

The most common assumption regarding sorting problems is the assumption that there exists a total ordering between the elements constituting the input. In the worst of the scenarios, the elements being compared may have absolutely no relative order between themselves. For example, several online multiplayer games involve players playing several games against random strangers, and the simplest way to rank them according to their skill is to keep track of the number of their victories. The assumption of comparisons being transitive can go for a toss in various such day to day scenarios.

In this thesis, we look at questions regarding these problems and improve previously known bounds and give new lower bounds with the goal to get a better understanding of these problems.

Chapter 2

Sorting Under Forbidden Comparisons with Orientation Oracle

2.1 Introduction

Comparison based sorting algorithms is one of the most studied areas in computer science. Huang, Kannan and Khanna [6] initiated the study of sorting when certain pairs of elements are forbidden to be compared and they gave a randomized $\tilde{O}(n\sqrt{n})$ algorithm [1].

A forbidden pair is a pair of elements x and y which cannot be directly compared to each other. This does not necessarily imply that the two elements are mutually incomparable, i.e. there may exist some element z, such that $x > z$ and $z > y$, thereby implying $x > y$. The input is an undirected graph $G(V, E)$, where V is the set of elements and E represents the allowed comparisons/edge queries (the two terms are used interchangeably). We call G as the comparison graph, and we assume that the algorithm is given this undirected graph (so it does not have to spend time to determine whether a given pair is comparable or not).

The comparison graph G has an underlying directed acyclic orientation realizing a poset P_G maintained by an adversary. The goal of the problem is to determine

[1] $\tilde{O}$ ignores polylogarithmic factors

the orientation of all the edges by probing the adversary for as few edges as possible (and using transitivity). We call this problem loosely as sorting the poset underlying the graph G or simply sorting the graph G. The number of queries made to the adversary is defined as the query complexity.

Poset sorting has been well studied for width bounded posets in [7]. It is known that if a poset P has width (the size of the largest anti-chain) at most w, then the information theoretic bound for the query complexity is $\Omega((w + \lg n)n)$. A query optimal algorithm n for width bounded posets whose total complexity is $O(nw^2 \lg(n/w))$ is presented in the same paper.

Let q be the number of forbidden pairs in the given graph and let w be the width of the poset P_G realized by the vertices of the graph. The parameters q and w are known to be related, since q is at least the number of incomparable pairs in P_G which is at least $\binom{w}{2}$. Hence, w is $O\left(\sqrt{q}\right)$, although $\sqrt{q}$ can be substantially larger than w. Take, for example, a total order with the comparison graph as simply a path on n vertices. The width of the graph is one, as there is no anti-chain of length more than one, however the graph is missing $q = \binom{n}{2} - n + 1$ edges.

Banerjee and Richards [8] used q as a measure of how difficult it is to sort G. When q is 0, there is no forbidden pair, and the standard comparison based sorting algorithms can determine the poset using $O(n \lg n)$ edge queries. At the other extreme, when $|E|$, the number of edges present in the graph is small, one can simply probe every edge in the graph and sort much faster. A good structure on the input graph can also help to sort much faster, regardless of how large q is, as we show later in this chapter.

Banerjee and Richards [8] showed a $O((q + n) \lg n)$ query bound for sorting a graph with q missing edges. We modify the algorithm resulting in a query complexity of $O((q + n) \lg(n^2/q))$. For $q = \Theta(n^2)$, this bound is better than the bound of Banerjee and Richards. We also improve the running time of $O(n^\omega)$ to running time of $O(n^2 \lg n)$. We also show the first lower bound by exhibiting a comparison graph with q forbidden pairs and an orientation of the edges that requires $\Omega(q + n \lg n)$ queries.

Then, we investigate the problem for some special classes of comparison graphs. The graph classes which we have considered are incomparable, in the sense that no graph class is contained by the other. A graph is *chordal* if every cycle of length

at least 4 in the graph has a chord (an edge joining two non-adjacent vertices of the cycle). We show that if the comparison graph is a chordal graph, then we can sort the graph using $O(n \lg n)$ queries using the simplicial ordering of the vertices of the graph. A graph is a *comparability* graph if its edges can be oriented such that for every triple x, y, z of vertices, if there is a directed edge from x to y, and if there is a directed edge from y to z, then there is a directed edge from x to z. We call such an orientation a comparability orientation. Note that a directed acyclic orientation is not necessarily a comparability orientation. For example, if G, the underlying undirected graph is a path on three vertices a, b and c, then the orientation $a \to b \to c$ is not a comparability orientation though it is a directed acyclic orientation. The two comparability orientations for such a path are $a \to b \leftarrow c$ and $a \leftarrow b \to c$.

We show that if the comparison graph is a comparability graph and if the adversary answers the edge queries based on a comparability orientation of the graph, then we can sort the poset underlying the graph using $O(n \lg n)$ queries. A consequence of this result is the following. Suppose we have a poset represented by a directed graph such that the directed edges of the graph represent *all* the order relations between the elements of the poset (so the missing edges represent incomparable elements of the poset). And suppose we are not given the orientation, but only the underlying undirected graph. Then we can find the poset probing the adversary for $O(n \lg n)$ edge queries (the rest of the edge directions are deduced using transitivity). This may be of independent interest.

We also look at the class of graphs known as proper circular arc graphs. A circular-arc graph is the intersection graph of a set of arcs on the circle. It has one vertex for each arc in the set, and an edge between every pair of vertices corresponding to arcs that intersect. A circular arc graph is a proper circular-arc graph [9], if there exists a corresponding arc model such that no arc properly contains another. Recognizing a proper-circular arc graph can be done in linear time [10, 11] . If the graph is a proper-circular arc graph we show that finding the poset can be done by probing $O(n \lg n)$ edges.

2.1.1 Organization of the Chapter

In Section 2.2, we give the improved upper bound for sorting a graph based on q. In Section 2.3, we give the first lower bound for sorting a comparison graph. We also give lower bounds, in case the comparison graph is guaranteed to have a total order. In Section 2.4, we outline the $O(n \lg n)$ query algorithm to sort comparability graphs, proper circular arc graphs and chordal graphs.

In Section 2.5, we give some concluding remarks on sorting a graph whose partial order is a total order.

2.1.2 Definitions and Notation

Let G be a graph. We denote by $V(G)$ (or simply V) the vertex set of G, and by $E(G)$ (or simply E) the edge set. If $|V(G)| = n$ and $|E(G)| = m$, G is said to have *order* n and *size* m.

For any $v \in V(G)$, $N(v)$ denotes the set of neighbors of v, and we define $\deg(v) = |N(v)|$ and $n(v) = n - 1 - deg(v)$ is the number of vertices that are not adjacent to v. For a subset $S \subseteq V(G)$, $G[S]$ denotes the subgraph of G induced by S and for $v \in S$, $N_S(v)$ denotes the neighbors of v in $G[S]$. Analogously, we define $\deg_S(v) = |N_S(v)|$.

A tournament is a directed graph in which there is exactly one directed edge between every pair of vertices, and a (directed or undirected) Hamiltonian path is a (directed or undirected respectively) path that visits every vertex of the graph exactly once.

An approximate median (if it exists) of G denotes a vertex v which can reach and can be reached by $\Omega(n)$ vertices, whereas a median vertex (if it exists) is a vertex which can reach $\lfloor n/2 \rfloor$ vertices and can be reached by $n-1-\lfloor n/2 \rfloor$ vertices.

To keep the notation simple, we sometimes omit ceilings and floors on fractions when we actually mean integers, typically on the sizes of the vertex subsets we handle. This does not affect the asymptotic analysis.

2.2 Sorting Under Forbidden Comparisons

2.2.1 Upper Bounds

We start by revising a known algorithm [8] with some slight modifications including some constants, which we believe gives more accurate analysis. Central to their algorithm is the construction of several subsets of vertices such that each subset induces a clique.

If $n^2 \leq 320q$, it means that the total number of edges $|E| = \binom{n}{2} - q < 160q - q$, thus $|E| < 159q$, that is the number of missing edges is asymptotically of same size as the number of edges present. In such a case, we use brute force to query all the edges in $O(q)$ time and find the total sorted order. So we focus on the case when $n^2 > 320q$.

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

Lemma 2.2.1. *If $q \leq cn$, there exists a subset $X \subseteq S$ such that $|X| \geq \frac{n}{2(4c+1)}$ and $G[X]$ is a complete graph and such a set can be found in $\mathrm{O}(n^2)$time.*

Proof. We construct X explicitly. We start with $X = \{u\}$, where u is an arbitrary vertex in S. We repeatedly pick a vertex (say t) adjacent to all vertices in X from V and add it to X and remove it from V, until no such vertex exists. Let v be the last vertex to be added to X, after which V becomes empty. Since v has at least $|S| - 4c$ neighbors, whenever we pick a neighbor of v from S to add to X we lose at most $4c + 1$ vertices (including the vertex we picked). Hence, $|X|$ is at least $\frac{|S|}{4c+1} \geq \frac{n}{2(4c+1)}$.

Clearly the above procedure runs in $\mathrm{O}(n^2)$ time and makes no comparisons. □

Now, we deal with the general case for arbitrary values of q.

We will define the sets R and S analogous to the previous Lemma. We have $R = \{v \in V |\ n(v) > 4q/n\}$. Thus, we get $|R| \leq n/2$. Similarly $S = \{v \in V |\ n(v) \leq 4q/n\}$ Thus all vertices in S do not share an edge with at most $4q/n$ vertices in G. Additionally, $|S| \geq n/2$. Now we will apply Lemma 2.2.1 successively to

construct a big-enough set $X \subseteq S$ which we will use to find an approximate median of V . For general values of q we can also show the following

Lemma 2.2.2. *There exists a subset $X \subseteq S$ such that $|X| \geq \frac{n}{2(4q/n+1)} = \frac{n^2}{2(4q+n)}$ and $G[X]$ is a complete graph and such a set can be found in $O(n^2)$time.*

Constructing several X_i

Let us define $S_i = S \setminus \bigcup_{j=1}^{i} X_j$. We construct the first clique $X_1 \subseteq S$ using the method detailed in Lemma 2.2.1.
There are two cases:
Case 1. $q \leq n$: In this case, we use Lemma 2.2.1 and hence $|X_1| \geq (n/2)/(4q/n + 1) \geq n/10$. We take the exactly $n/10$ elements from X_1 and discard the rest to be processed in the next round. Then we compute X_2 from S_1 which is of size at least $(n/2 - n/10)/5 = 4n/50$. Let $X = X_1 \cup X_2$. Thus $|X| \geq 9n/50$.

Case 2. $q \geq n$: In this case we have $|X_1| \geq \frac{n^2}{2(4q+n)} \geq n^2/10q$ by Lemma 2.2.2. We take $|X_1| = (1/10)n^2/q$ discarding some vertices if necessary.

Similarly we construct $X_2 \subset S_1$. $S_1 = |S| - n^2/10q \geq n/2 - n^2/10q$. It can be seen that $X_2 \geq |S_1|/5q/n = (n^2/10q)(1 - n/5q)$. We keep $(n^2/10q)(1 - n/5q)$ vertices in X_2 and the rest are discarded to be processed the next round. In general for the clique X_r we have $|X_r| \geq (n^2/10q)(1 - n/5q)^{r-1}$. Now we let $X = \cup_{i=1}^{r} X_i$. We will show that $|X| \geq \delta_2 n$ for some constant $\delta_2 > 0$. When $r = 5q/n + 1$, we have

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

since $q \geq n$. Hence, $|X| = \bigcup_{i=1}^{r} |X_i| \geq r|X_r| \geq (15/100)n$, giving $\delta_2 = 15/100$.

Thus, in both the cases, we can create X such that $|X| \geq 15n/100$.

In both the cases, for each $X_i (1 \leq i \leq r)$ we keep a subset Y_i of size $|X_r|$ and throw away the rest. Clearly, for each i, the induced sub-graph $G[Y_i]$ is also a clique. Let $Y = \bigcup_{i=1}^{r} Y_i$. We have $|Y| \geq 15n/100$ in both the cases.

Computing An Approximate Median Of V

Recall the definition of approximate median. We compute an approximate median with respect to all the vertices (the set V) and not just the set S. This vertex will divide the set V in constant proportions. This is done by using the set Y . For each Y_i we find its median using O($|Y_i|$) probes since $G[Y_i]$ is a complete graph using the classical median finding algorithm in a total order [12]. Let m_i be the median of Y_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 $A \subseteq V$ (m may not be a member of A) as $U(m, A) = \{a \in A |\ a > m\}$. Similarly we define the lower set $L(m, A)$. We want to compute the sets $U(m, Y)$ and $L(m, Y)$. However, m may not be neighbor of all the elements in Y. So we compute approximate upper and lower sets by probing all the edges between m and the neighbors of m in Y. These sets are denoted by $\tilde{U}(m, Y)$ and $\tilde{L}(m, Y)$ respectively. It is easy to see that there exists some $m \in M$ which divides Y 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 all be neighbors of each other hence we will approximate m using the ranks of the elements in M with respect to the set Y (which is $|\tilde{L}(m, Y)|$). We compute the sets $|\tilde{L}(m, Y)|$ and $|\tilde{U}(m, Y)|$ for all the medians in M. All elements in M are rearranged in increasing order of $|\tilde{L}(m, Y)|$. $m^* \in M$ is chosen by taken the element which is a median of this sorted set. Next we prove that the element m^* the approximate median of M, picked using the above procedure, is also an approximate median of Y .

Lemma 2.2.3. *The element m^* is an approximate median of Y.*

Proof. First we show that the median of M is an approximate median of Y. Let us take the elements in M in sorted order $(m_1, ..., m_r)$, so the median of M is $m_{\lceil r/2 \rceil}$. Now $L(m_{r/2}, Y) \geq \bigcup_{i=1}^{\lceil r/2 \rceil} L(m_i, Y_i)$. Since, the sets Y_i are disjoint and $L(m_i, Y_i) \geq |X_r|/2$, we have $|L(m_{\lceil r/2 \rceil}, Y)| \geq |X_r|r/4$ (ignoring the floor). Similarly we can show that $|U(m_{\lceil r/2 \rceil}, Y)| \geq |X_r|r/4$. Hence $m_{\lceil r/2 \rceil}$ is an approximate median of Y, since it has at least $|Y|/4$ elements less than itself and $|Y|/4$ elements greater than itself. Now we show that $||\tilde{L}(m^*, Y)| - |L(m_{\lceil r/2 \rceil}, Y)|| < 4q/n$. Consider the sorted order of elements in M according to $|\tilde{L}(m^*, Y)|$. Since each element in $m \in M$ has

at most $4q/n$ missing neighbors in Y , we have $||\tilde{L}(m,Y)| - |L(m,Y)|| < 4q/n$. 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 $|L(m,Y)|$ cannot be more than $4q/n$ apart from $m_{\lceil r/2 \rceil}$ in the sorted order of Y. Hence

$$|L(m^*,Y)| \geq |X_r|r/4 - 4q/n \geq 15n/400 - 4q/n \geq n/40$$

whenever[2] $n^2 \geq 320q$. In an identical manner we can show that $|U(m^*,Y)| \geq n/40$. Hence, m^* is an approximate median of Y . □

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

This gives the following Lemma

Lemma 2.2.4. *Every graph with $q < n^2/320$ missing edges has an approximate median vertex m computable using $\mathrm{O}(q+n)$ queries, such that m is greater than at least $n/40$ elements and less than at least $n/40$ elements. Furthermore m can not be compared with at most $\mathrm{O}(q/n)$ elements.*

The algorithm of Banerjee and Richards [8] uses this Lemma recursively to break the problems into smaller subproblems, to solve the sorting problem. Both their and our algorithms are recursive, and the main difference is that we bail out of the recursion earlier which improves the bounds. Their algorithm has a partition step to break the input into smaller subproblems and a merging step where they combine smaller solutions to solve larger subproblems. We modify the process here by only having a single partitioning step, after which the problem breaks into two disjoint problems. The output of the algorithm is an orientation of all of the edges of the graph G. This simplification also helps us improve the running time from $\mathrm{O}(n^\omega)$ (where $\omega \in [2, 2.38]$ is the exponent in the complexity of matrix multiplication) to $\mathrm{O}(n^2 \lg n)$.

[2]The constant used in [8] is 200, instead of 320. We have made the small change to factor in a calculation gap in the last equation on page 7 in [8].

Algorithm 1: *Sort* $G(V, E, q, depth)$

if $q \geq n^2/320$ *or* $depth = \lg(\frac{n^2}{q})/\lg(\frac{40}{39})$ **then**
    query every edge in E and output their orientations;
end
else
    Find an approximate median vertex m using Lemma 2.2.4;
    Compare m with all its neighbors in V and output their orientations;
    $V_l = \{v | v \in N(m), v < m\}$ and $V_h = \{v | v \in N(m), v > m\}$;
    $V_{incomp} = \{v | v \in V - N(m)\}$;
    Compare every vertex in V_{incomp} with all its neighbors in V and output their orientations;
    $\forall u, v$ such that $u \in V_l$ and $v \in V_h$ and $uv \in E$ orient edge from v to u;
    Sort $G(V_l, E_l, q_l, depth + 1)$ and Sort $G(V_h, E_h, q_h, depth + 1)$ (E_l and E_h are the set of edges in $G[V_l]$ and $G[V_h]$ respectively, while q_l and q_h are the number of missing edges in $G[V_l]$ and $G[V_h]$ respectively)
end

Theorem 2.2.5. *Sorting a comparison graph with q forbidden edges can be done in $min\{|E|, \mathrm{O}\Big((q+n)\lg(\frac{n^2}{q})\Big)\}$ edge queries and takes $\mathrm{O}(n^2 \lg(n^2/q))$ time.*

Proof. The details of the algorithm are given in Algorithm 1 which is initially called with $depth = 0$. The algorithm is essentially the same as that of [8], except that it is forced to stop the recursion, when the depth of the recursion i is $(\lg(\frac{n^2}{q}))/\lg(\frac{40}{39})$. It checks if the value of q is more than a certain threshold or the depth of the recursion is less than a threshold and then it breaks the problem into two disjoint problems using $\mathrm{O}(q+n)$ queries by Lemma 2.2.4. Suppose at level l of the recursion, the sizes of the subproblems are $n_1, n_2, n_3 ... n_t$ and the number of missing edges in these subproblems are $q_1, q_2, q_3 ... q_t$ respectively. The algorithm either breaks them into smaller subproblems or queries every edge in that subproblem (whenever $q_i \geq {n_i}^2/320$ missing edges). In the latter case, the algorithm performs at most $\binom{n_i}{2} < 160 q_i$ queries. The query cost incurred by the incomparable elements at any internal node of recursion tree is also $\mathrm{O}((q_i/n_i) * n_i) = \mathrm{O}(q_i)$. In either of the cases, the total number of queries done at this level is at most $\mathrm{O}\Big(\sum_{i=1}^{l}(q_i + n_i)\Big) = \mathrm{O}(q+n)$. Thus at any level of the recursion tree, the algo-

rithm makes at most $O(q+n)$ queries.

When the depth of the recursion i is $(\lg(\frac{n^2}{q}))/\lg(\frac{40}{39})$, the number of subproblems would be $O(n^2/q)$ and the size of each subproblem would be at most $(39/40)^i n = q/n$, since each subproblem has at most 39/40 fraction of the vertices of its parent subproblem by Lemma 2.2.4.

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

In the above theorem, we have just counted the query complexity and ignored the time it takes to find the queries to make. If the value of $q > n^2/320$, then the algorithm runs in $O(E) = O(n^2)$ time, by asking all queries using brute force. If $q < n^2/320$, constructing the sets X_i for finding the median (Step 5 of the algorithm) using Lemma 2.2.4 takes $O(n^2)$ time by Lemma 2.2.1. Another $O(|E_{lh}|)$ time is spent for finding orientations of edges across set V_l and V_h (E_{lh} is the set of edges between set V_h and V_l). Thus, one can see that the actual running time to perform each level of iteration is $O(q+n+|E_{lh}|+n^2) = O(n^2)$. Since there are $O(\lg(n^2/q)$ such iterations, the algorithm has a total running time of $O(n^2\lg(n^2/q))$, which improves the previous best running time of $O(n^\omega)$ ($\omega \in [2, 2.38]$ is the exponent in the complexity of matrix multiplication). □

2.3 Lower Bounds

2.3.1 Lower Bound for Dense Graphs

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

Lemma 2.3.1. *There exists a graph with $q \geq n^2/4$ and an orientation such that, $\Omega(|E|)$ edge queries are needed to sort the graph.*

Proof. The graph which the adversary constructs is a complete bipartite graph with A and B as the equal sized parts. Here the number of missing edges, as well as the number of edges present is, roughly $n^2/4$. The adversary orients the edges from A to B forcing the algorithm to probe every edge, as the algorithm can not deduce any of the edges using transitivity. If the algorithm fails to query an edge, the algorithm has a choice of flipping its direction. □

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

Theorem 2.3.2. *When $q < n^2/4$, there exists a graph and an orientation of the edges such that any algorithm has to probe $\Omega(q + n \lg n)$ queries to sort the graph.*

Proof. In this case, the graph constructed by the adversary consists first of a complete bipartite graph B with partitions X and Y of size roughly $\sqrt{q}$ each such that it has q edges and has q edges missing. Then it forms a complete graph K on the remaining $n - 2\sqrt{q}$ vertices and maintains a total order among those vertices. And it has all the edges between every vertex of K and every vertex of B. If a query comes between a vertex b in the bipartite graph B and a vertex c in the complete graph K, the adversary directs the edge from b to c. If the edge query is between two vertices inside the complete graph, the adversary answers consistent with the total order it maintains. If the edge query is between two elements inside the bipartite graph, the adversary answers according to the adversary in Lemma 2.3.1 with X and Y playing the role of A and B respectively.

The number of edge queries required to sort the complete graph K would be $\Omega(n \lg n)$ and the number of edge queries required to sort the bipartite graph B is at least $\Omega(q)$ from Lemma 2.3.1, which gives a lower bound of $\Omega(q + n \lg n)$ edge queries. □

2.3.2 Lower Bound for Sparse Graphs

Let us consider a comparison graph $G(V, E)$ such that $|E| = |V| = n$ and graph is an undirected cycle, and let us say that the adversary also reveals that the

underlying poset is a totally ordered set. Even in this situation it is easy to show that $n-1$ edge comparisons are necessary to sort the graph.

Theorem 2.3.3. *If the comparison graph is a simple cycle on n edges and has the guarantee that the n vertices form a total order, then $n-1$ edge queries are necessary and sufficient to sort the graph.*

Proof. Suppose $v_1, v_2...v_n$ are the vertices of G and $v_1v_2, v_2v_3...v_{n-1}v_n$ and v_nv_1 are the edges in G. Since there are n vertices and the adversary has revealed that they form a totally ordered set, there are $2n$ possible sorted orders. This is due to the fact that each vertex can be the minima of totally ordered set, and every vertex can give possibly two sorted orders. One sorted order is obtained by looking at the cycle in clockwise direction, and another is obtained by looking at the cycle in anti-clockwise direction.

This also implies that all but two vertices have one incoming and one outgoing edge. One vertex has two incoming edges (referred to as minima) and one vertex has two outgoing edges (referred to as maxima), and these two vertices share an edge in G.

The adversary has a simple strategy. It answers the first $n-2$ queries so that the edge being queried is directed in clockwise direction. The claim is that no algorithm can correctly determine the directions of the last two edges, even with the knowledge of the first $n-2$ edge queries. There are two cases regarding the remaining two edges which have not been probed.

Case 1: The two edges share a common vertex.

Suppose v_iv_j and v_jv_k are the two unprobed edges. The remaining probed edges form a directed path, since all the edges are on a cycle and the adversary is answering all the edges in clockwise direction. Let v_i be the source of the directed path and v_k be the sink of this directed path (without any loss of generality). At this point, there are two possible total orders, which are consistent with the answers given till now. One total order has v_i as the minima and v_j as the maxima, while the other total order has v_j as the minima and v_k as the maxima. Thus $n-1$ edge queries are necessary in this case.

Case 2: The two edges do not share a common vertex.

Suppose v_iv_j and v_kv_l are the two unprobed edges. The remaining probed edges

form two disconnected directed paths, since all the edges are on a cycle and the adversary is answering all the edges in clockwise direction.

Among v_i and v_j, let v_i be the vertex which recieves an incoming edge and v_j be the vertex which has an outgoing edge (without loss of generality). Similarly among v_k and v_l, let v_k be the vertex which recieves an incoming edge and v_l be the vertex which has an outgoing edge(without loss of generality).

At this point, there are two possible total orders, which are consistent with the answers given till now. One total order has v_i as the minima of the total order, and the other total order has v_k as the minima.

Thus, when two edges have not been probed the algorithm cannot correctly output the total order in the underlying graph. Similarly it is easy to see that if k edges have not been probed, then there are at least k vertices which do not have any incoming edge and hence any of the k vertices can be the maxima of the total order. Thus, the algorithm cannot figure out which of these edges is a anti-clockwise edge. Therefore, under this adversary, the algorithm cannot infer the direction of any edge until the algorithm makes $n-1$ queries.

Since finding an anti-clockwise edge, reveals the source of the total order and hence the entire total order, finding an anti-clockwise edge under this adversary is sufficient to find the total order. Finding the anti-clockwise edge is also necessary, since the anti-clockwise edge determines the source of the total order.

Thus $n-1$ edge queries are necessary in this case.

We also prove that this lower bound is tight in this case of a cycle graph. A simple $n-1$ edge query algorithm to find the directions of edges also exists for a simple cycle graph if there is the guarantee that the vertices form a total order. The algorithm picks any $n-1$ edges and queries them. One of the three possible cases can happen.

Case 1: The algorithm finds a vertex v which has two incoming edges, in which case v is the smallest element in the total order.

Let u and u' be the two neighbors of v in the cycle. This implies that one of the neighbors of v is a largest element in the total order. Since only one edge has not been queried by the algorithm, it means that one of the vertices (u and u') know that they have two outgoing edge or one of the vertices has one incoming and one outgoing edge. If u (without loss of generality) has two outgoing edges,

then u is the largest element of the total order, which defines and a unique total order, defined by directed path which can reach v from u through u'. Otherwise, if u (without loss of generality) has one incoming and one outgoing edge, then u' is the largest element of the total order, which defines and a unique total order, defined by directed path which can reach v from u' through u.

Case 2: The algorithm finds a vertex v which has two outgoing edges, in which case v is the largest element in the total order.

The proof is analogous to the proof for case 1.

Case 3: Every vertex has at most one incoming and one outgoing edge, in which there is a directed path of length n which reveals a total order, and the comparison which has not been made is the direct comparison between the smallest and largest element of the total order.

Thus $n-1$ edge queries are sufficient to find the total order in a simple cycle graph.

□

2.4 Sorting Special Comparison Graphs

In general, the number of edge queries needed to sort a poset can depend on both its size (n) and the number of forbidden pairs (q). However, when the comparison graph accompanying the poset has some additional structure, we show that the number of edge queries needed is at most $\mathrm{O}(n \lg n)$, and more importantly becomes independent of q.

In this section, we show that when the comparison graph is either *chordal*, *transitively orientable* (i.e. a comparability graph) or *proper-circular arc* graph, the graph can be sorted by making $\mathrm{O}(n \log n)$ edge queries. In fact, the algorithms presented below output linear extensions (i.e. topological orderings) of the input posets. A topological ordering $v_1, v_2 ... v_n$ of its vertices is an ordering where if (v_i, v_j) is a directed edge in the graph, then $i < j$. It is well-known that every directed acyclic graph has a topological sort [13].

Topological sorting of a general directed acyclic graph $G = (V, E)$ requires $\Omega(|V| + |E|)$ running time, while the algorithms in this section make $\mathrm{O}(n \lg n)$ queries. One can deduce the edge directions from the topological sort of the vertices

using transitivity, but they are not counted in the query complexity.

2.4.1 The General Idea

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

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

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

A binary insertion of a vertex v in the topological order $v_1, v_2, \ldots v_t$ of its neighbors is the process of finding whether $v \to v_1$ or $v_t \to v$ or some adjacent vertices v_i and v_{i+1}, $i < t$ such that $v_i \to v$ and $v \to v_{i+1}$. This is similar to the process of searching a value in a sorted array using binary search and can be performed using $\lceil \lg t \rceil + 1$ queries.

When the comparison graph is a complete graph, then any directed acyclic orientation of its edges has a unique directed Hamiltonian path (as with the orientation, the directed graph is a transitive tournament), and hence has a unique topological ordering. In this case, the above procedure is essentially the binary insertion sort to sort the graph.

What we show is that such a binary insertion type procedure works even if the comparison graph is not a complete graph. Towards that we capture the following two simple observations.

Lemma 2.4.1. *If G is a directed acyclic orientation of a complete graph (also known as tournament), then it has a unique topological order.*

Lemma 2.4.2. *Let $G(V, E)$ be a directed acyclic graph, and let $S \subseteq V$ be such that $G[S]$, the induced subgraph on S, is a tournament on s vertices. Let $v_1, v_2, \ldots v_s$ be the unique topological order of vertices of S. In any topological order of the vertices of V, the elements of S appear in the unique topological order within S.*

Proof. Let $u, v \in S$, and let u appear before v in one topological order, then (u, v) is directed from u to v ((u, v) edge is there as $G[S]$ is a tournament). Then v cannot appear before u in any other topological order of $G[V]$ by the definition of topological order. □

2.4.2 Chordal Comparison Graphs

Chordal graphs [14] form a well-studied class of graphs as they can be recognized in linear time [15], and several problems that are hard on other classes of graphs such as graph coloring can be solved in polynomial time for chordal graphs [16]. An undirected graph is chordal if every cycle of length greater than three has a chord, namely an edge connecting two non-consecutive vertices of the cycle [14].

In a graph G, a vertex v is called *simplicial* if and only if the subgraph of G induced by the vertex set $\{v\} \cup N(v)$ is a complete graph. A graph G on n vertices is said to have a perfect elimination ordering (PEO) if and only if there is an ordering $v_1, v_2 ... v_n$ of G's vertices, such that each v_i is simplicial in the subgraph induced by the vertices $v_1, v_2 ... v_i$. Every chordal graph has a perfect elimination ordering which can be found in $O(|E| + |V|)$ time [16, 15]. Our algorithm first finds a perfect elimination ordering of the given chordal graph. Note that no edge queries are made while finding a perfect elimination ordering.

Now we apply the idea outlined in Section 2.4.1 in the perfect elimination ordering. Suppose $v_1, v_2, \ldots, v_n$ is a perfect elimination ordering (PEO) for the graph.

We obtain a topological sort of the vertices of the graph, in the inductive order of the PEO. Let G_i be the induced graph on the first i vertices of the PEO, and suppose that we know all the orientations of the edges of G_i, and we have a topological sort of the orientation of G_i. Now we insert v_{i+1} using binary search as follows. We consider the projection of the topological order of vertices of G_i that are neighbors of v_{i+1}, and insert v_{i+1} using binary search among them. In particular, suppose we have vertices $v_p, v_q \in G_i$ such that $v_p \to v_{i+1} \to v_q$ where v_p and v_q are consecutive vertices in the topological order of G_i among neighbors of v_{i+1} (it is possible that one of v_p or v_q does not exist; v_p may not exist if v_q is the first vertex of G_i in the topological order among neighbors of v_{i+1} and v_q may not exist if v_p is the last vertex of G_i in the topological order among neighbors of v_{i+1}). Then we simply insert v_{i+1} after v_p (or before every vertex if v_p does not exist) in the topological order. We claim the following.

Lemma 2.4.3. *The resulting order is a topological order of the directed acyclic orientation of* G_{i+1}.

Proof. We only need to worry about edges incident on v_{i+1} as for every other pair of vertices, their relative order has not been changed by the insertion and hence the topological order property is satisfied by induction. By Lemma 2.4.2, v_{i+1} has been inserted properly in the unique topological order on the induced subgraph of v_{i+1} and its neighbors, as they form a clique (because of simplicial ordering property). □

Thus we have the following result

Theorem 2.4.4. *If the comparison graph is a chordal graph, then the poset underlying the graph on n vertices can be sorted using $O(n \lg n)$ edge queries.*

The algorithm first finds a PEO of the vertices in $O(|E| + |V|)$ time. It also spends $O(deg(v_{i+1}))$ time to infer the directions of all edges between v_{i+1} and vertices in G_i, which amounts to $O(|E|)$ time over all the vertices . Thus, the total time spent for finding the directions of all the edges is $O(|E| + |V|)$.

2.4.3 Proper Circular Arc Graphs

Circular graphs are intersection graphs of arcs on a circle. These graphs are reported to have been studied since 1964, and they have been receiving considerable attention since a series of papers by Tucker [9] in the 1970s. Various subclasses of circular-arc graphs have also been studied. Proper circular-arc graphs form a major and well studied subclass of this class of graphs [10]. A proper circular arc graph is a circular arc graph in which no arc properly contains another. Recognizing these graphs and constructing a proper arc model can both be performed in polynomial time [11].

We can show that $O(n \lg n)$ queries are enough for sorting proper circular arc graphs, for which we need the following theorem

Theorem 2.4.5 ([11])**.** *Neighborhood $N(v)$ of every vertex v in a proper circular arc graph can be split into atmost two disjoint sets N_1 and N_2 such that $N(v) = N_1 \cup N_2$ and the induced subgraphs on N_1 as well as N_2 are complete subgraphs. This split of the neighborhood for every vertex can be found in $O(|E||V|)$ time.*

We maintain a subset V_i of i arbitrary vertices such that the directions of all the edges in the induced subgraph $G(V_i)$ are known and show that the directions

of all the edges in the induced subgraph $G(V_{i+1})$ where $V_{i+1} = V_i \cup \{u\}$ (where $u \in V \backslash V_i$) can also be inferred using $O(\lg n)$ additional queries. This strategy is similar to the strategy described in 2.4.1. Initially V_i consists of a single arbitrary vertex.

Using Theorem 2.4.5, find the split of neighborhood of u in the proper circular arc graph formed by induced subgraph of $G(V_{i+1})$. This neighborhood is guaranteed to be partitioned into at most two sets of vertices N_1 and N_2, such that induced subgraph on N_1 and N_2 are both complete subgraphs.

Since N_1 and N_2 induce complete subgraphs, vertices in N_1 and N_2 have total order. This implies that u can find its relations with vertices in N_1 and N_2 using atmost $O(\lg n)$ queries. Then we spend $O(deg(u))$ time to mark the directions of edges which have not been queried, but exist in the induced subgraph over vertices in V_{i+1}, which can done using the acyclicity of the graph. Note that the time spent marking edges using the property of acyclicity of the graph, do not count for edge queries.

Thus we have the following result

Theorem 2.4.6. *If the comparison graph is a proper circular arc graph, then the poset underlying the graph can be sorted using* $O(n \lg n)$ *edge queries.*

The algorithm uses the procedure in Theorem 2.4.5 for a maximum of n times, which gives the running time to be $O(n^2|E|)$. Additionally $O(n)$ time is spent by the algorithm during i^{th} iteration to assign orientations to edges based on acylicity of the underlying graph. Thus $O(n^2)$ time is spent assigning edges. The total running time of the algorithm is $O(n^2|E|)$.

The general case of circular arc graphs seem to have a less rigid structure and thus we have been unable to find a similar result for them.

2.4.4 Comparability Comparison Graphs

An undirected graph is a *comparability graph* [14] if the edges of the graph can be oriented in a way that for every triple x, y, z of vertices, if there is a directed edge from x to y, and a directed edge from y to z, then there is a directed edge from x to z. While both comparability graphs and chordal graphs are sub-classes of the well-known class of *perfect* graphs, there is really no relation between both these classes

of graphs. In particular, in comparability graphs, we have no guarantee about the existence of a simplicial ordering. However, we argue that we can incrementally insert new vertices and get the orientation of its edges by doing the variation of binary search outlined in Section 2.4.1, provided the adversary answers the query according to a comparability orientation.

As we do not have simplicial ordering, we simply start with an arbitrary ordering of the vertices and maintain the topological sort of the subgraph induced by the initial set of vertices and incrementally insert the new vertex. Let G_i be the induced graph on the first i vertices ($i \geq 1$) in the arbitrary order, and suppose that we know all the orientations of the edges of G_i, and we have a topological sort of the orientation of G_i. Now we insert v_{i+1} using binary search as before. We consider the projection of the topological order of vertices of G_i among the neighbors of v_{i+1}, and insert v_{i+1} using binary search among them. In particular, suppose we have vertices $v_p, v_q \in G_i$ such that $v_p \to v_{i+1} \to v_q$ where v_p and v_q are consecutive vertices in the topological order of G_i among neighbors of v_{i+1}. (Here again, it is possible that one of v_p or v_q does not exist; v_p may not exist if v_q is the first vertex of G_i in the topological order among neighbors of v_{i+1} and v_q may not exist if v_p is the last vertex of G_i in the topological order among neighbors of v_{i+1}.) Then we simply insert v_{i+1} after v_p (or before every vertex if v_p does not exist) in the topological order. We claim the following.

Lemma 2.4.7. *The resulting order is a topological order of the directed acyclic orientation of G_{i+1}.*

Proof. Here again we only need to worry about edges incident on v_{i+1} as for every other pair of edges, their relative order has not been changed by the insertion and hence the topological order property is satisfied by induction. Suppose that there exists a vertex $u \in G_i$ such that u comes before v_{i+1} in the resulting topological order and we have the edge v_{i+1} to u. Then $u \neq v_p$ (as $v_p \to v_{i+1}$) and u has to come before v_p in the order (as v_p immediately preceded v_{i+1} in the order). But as $v_p \to v_{i+1} \to u$, there must be an edge from $v_p \to u$ as the orientation is a comparability orientation, which is a contradiction to the fact that we have a topological order of vertices of G_i (as u comes before v_p in the topological order of vertices of G_i). The case when there is a vertex u that comes after v_{i+1} in the topological order, is similar to the case discussed. □

Thus we have,

Theorem 2.4.8. *If the comparison graph is a comparability graph, and if the adversary answers the queries according to a comparability orientation, then the poset underlying the graph can be sorted using* $O(n \lg n)$ *edge queries.*

When v_i is added to G_i during the i^{th} iteration, $O(\lg n)$ edge queries are made by the algorithm but before starting the next iteration, the algorithm spends $O(deg(v_i))$ time to mark the orientation of the edges whose directions have been deduced using transitivity. Thus, the algorithm spends $O(E + V)$ time to mark the orientation of the edges.

Note that our proof crucially used the fact that adversary answers the query according to a comparability orientation, and this is not just the artefact of the proof. Otherwise we may not be able to sort using $O(n \lg n)$ queries. For example, a complete bipartite graph (over two partitions A and B) is a comparability graph, but if we are told that the orientation of the adversary is a comparability orientation, then only two orientations are possible. One orientation has all edges directed from vertices in A to vertices in B and the other orientation has all edges directed from vertices in B to vertices in A. In this case, we can sort with just one query. However we have shown an $\Omega(n^2)$ lower bound in Lemma 2.3.1 if the adversary is free to choose any directed acyclic orientation. While this may appear as a restriction, another way to view the theorem is as follows, which can be of independent interest.

Corollary 2.4.9. *Suppose we have a poset represented by a directed graph whose directed edges represent all the underlying relations between elements of the poset. If we are given only the underlying undirected graph, and there is an adversary that answers queries according to the partial order, then we can determine the relations of the poset using* $O(n \lg n)$ *queries to the adversary.*

The algorithms mentioned in the preceding subsection exploit the additional information about the input poset which the comparison graph provides. We remark that though these algorithms perform $O(n \lg n)$ edge queries, we did not make any attempt to optimize the running times of the algorithm.

2.5 Concluding Remarks

We have given an improved upper bound of $O((q+n)\lg(n^2/q))$ and the first lower bound of $\Omega(q + n\lg n)$ for sorting an undirected graph on n vertices and q missing edges. There is still a gap between the upper and lower bound, narrowing this gap is an interesting open problem.

We gave algorithms which make $O(n\lg n)$ edge queries when the input comparison graph is a comparability or a chordal or a proper ciruclar arc graph, an interesting open problem is to find the largest class of graphs for which an $O(n\lg n)$ query algorithm is possible.

Finally, the problem is wide open when we know that there is a total order underlying the vertices of the comparison graph. For example, in that case, the complete bipartite graph can not be oriented as shown in Lemma 2.3.1 and hence we do not know of any lower bound other than $\Omega(n\lg n)$ regardless of the number of missing edges. In particular, sorting the graph when the graph is a complete bipartite graph is the famous *nuts and bolts* problem and can be sorted using $O(n\lg n)$ queries [17]. We conjecture that sorting any undirected graph whose vertices represent an underlying total order, can be done in $O(n\lg n)$ queries. In other words, if we know that the directed acyclic orientation of the comparison graph has a directed Hamiltonian path, then we conjecture that we can find the path using $O(n\lg n)$ queries.

In fact, for total-orders in the forbidden pairs model, we do not even know how to find an element of any given rank (or the median element) in $O(n)$ queries. We only know that $O(n)$ queries are sufficient for ranks that are at a constant distance away from the extremes.

Chapter 3

Sorting and Selection under Forbidden Comparisons with Distance Oracle

3.1 Introduction

Consider the following problem. There are n servers and each pair of servers can communicate between each other directly or indirectly through other servers. The structure as to how these servers are connected and how these messages are routed is not known. The only way to know about this structure is to send a message from a vertex and measure how much time it takes for the message to be received by a destination vertex. If the message takes t units of time to reach the destination vertex, then it can be inferred that it goes through $t - 1$ distinct servers. The goal is the find the underlying routing structure of these servers, while trying to minimize the number of message requests. These kind of problems are in general referred to as graph reconstruction problems [18].

Formally, we are given n vertices which have a hidden graph H built on them. The distance oracle, when given two vertices a and b, returns the number of edges on the shortest path between these vertices in the graph H. The goal is to find the hidden graph H while trying to minimize the number of distance queries to the oracle.

Culberson and Rudnicki [19] gave an algorithm for reconstructing a d-degree tree using $O(dn \log_d n)$ distance queries. Reyzin and Srivastava [20] argued that actual number of distance queries made was $O(n^{1.5}\sqrt{d})$, and they showed an $\Omega(n^2)$ lower bound for the graph reconstruction problem on general graphs [21]. Mathieu and Zhou [18] gave randomized algorithms for the reconstruction of a degree bounded graph with query complexity $\tilde{O}(n^{1.5})$, for the reconstruction of a degree bounded outerplanar graph with query complexity $\tilde{O}(n)$ and near-optimal approximate reconstruction of a general graph.

We initiate a study of a variation of the graph reconstruction problem where the oracle has certain restrictions. In particular, the input is given as an undirected graph $G(V, E)$, and the oracle can answer distance between two vertices u and v only if $uv \in E$. This restriction poses certain challenges and we study the simplest case of reconstruction of a simple path on n vertices. More specifically, we address the following problem.

Given an undirected graph $G(V, E)$ which has an underlying Hamiltonian path H and access to a distance oracle which gives the distance between vertices u and v in H only if uv is an edge in E, find the maximum number of distance oracle queries to find H.

Another motivation for the problem comes from the notion of generalized sorting which has been described in chapter 1. The problem we study in this chapter can be seen as a version of the sorting problem, where instead of direction queries, the oracle answers the distance between the ranks of the pair of elements being asked.

Our main result is that under this oracle query, there is no difference in complexity between complete and general graphs, and in both cases $O(n)$ queries are sufficient.

3.2 Organization of the Chapter

Section 3.3 we give the necessary terminologies and notations that we use in the chapter. Section 3.4 gives the $O(n)$ algorithm when the graph is a complete graph. In Section 3.5.1, we extend the algorithm to general graphs when we know a leaf on the Hamiltonian path H. In Section 3.5.2, we give the most general algorithm

taking $O(n)$ queries. Section 3.7 concludes with some remarks and open problems.

3.3 Definitions and Notation

$G(V, E)$ denotes the simple undirected graph given as an input where V ($|V| = n$) is the set of vertices and E is the set of edges. $N(v)$ denotes the set of vertices which share an edge with v in $G(V, E)$. H denotes the unique Hamiltonian path hidden in in G. A distance oracle when given two vertices u and v, returns the number of edges between u and v on the Hamiltonian path H in G which is denoted by $d(u, v)$.

3.4 Reconstructing Hamiltonian Path in Complete Graphs

First we look at the case, when adversary reveals that the undirected graph $G(V, E)$ is a complete graph on n vertices. In other words, the distance oracle can answer distance query between any two vertices of G.

Theorem 3.4.1. *Given a complete undirected graph on n vertices, the hidden Hamiltonian path H can be found using at most $3n/2 - 2$ distance queries and in $O(n)$ time.*

Proof. Pick an arbitrary vertex v and make all the distance queries of v with all the remaining vertices. These $n - 1$ distance queries, will give at most 2 vertices with the maximum distance from v. For any vertex w, a vertex lying at maximum distance from it on H is a leaf vertex. A leaf vertex of the Hamiltonian path H has a unique distance with every other vertex and hence if we perform all the remaining $n - 2$ distance queries from a leaf vertex, we can find the locations of each of these vertices on the Hamiltonian path H correctly. Thus, $2n - 3$ oracle queries are enough to find H.

We now argue that the number of queries can be reduced to $3n/2 - 2$.

After the first step comparing of v with all other vertices, suppose that all the distances are distinct. Then there is only one vertex for any distance from v, and

thus v is a leaf and we already have the Hamiltonian path. Otherwise, let l_1 be the vertex at maximum distance from v, then clearly l_1 is a leaf vertex. Similarly, let $l \leq n/2$ be the maximum distance for which v has two vertices at that distance, and let S be set of all vertices which have positive distance $\leq l$ from v, and let $S' = V \setminus (S \cup \{v\})$. Note that $d(l_1, v) + l = n - 1$.

It is clear that vertices u in S' (vertices u at distances $d(u, v) > l$) can only lie between v and l_1. If a vertex $u \in S'$ does not lie between v and l_1, it would imply that v lies between u and l_1 and hence the distance between l_1 and $u = d(l_1, v) + d(u, v) > d(l_1, u) + l = n - 1$ which is a contradiction, since the maximum distance between any two vertices on H can be at most $n-1$. Thus, vertices $u \in S'$ can be correctly located in H without making any oracle queries, since they lie between v and l_1 and at distance $d(u, v)$ from v.

S is of even size, since S contains vertices whose distances repeat. Every vertex in S has at most two valid locations in Hamiltonian path, which are equidistant from v. For every pair of equidistant vertices u and u' in S, we need to figure out which of these vertices lie between v and l_1 and which one does not. Finding location of any one of the equidistant vertices, uniquely determines the location of the other vertex. Perform oracle query ul_1, and if $d(u, l_1) > d(v, l_1)$, then v lies between u and l_1, else v' lies between u and l_1. Thus, at most $|S|/2$ queries are enough to figure out the location of vertices in the Hamiltonian path.

When n is even, $|S|$ is of size at most $(n-2)$ (since S does not have v, it implies that $|S| \leq n - 1$ and $n - 1$ is odd). Thus, the total number of oracle queries made the algorithm, when n is even at most $n - 1 + (n - 2)/2 = 3n/2 - 2$.

If $n(= 2k + 1)$ is odd, then two possible cases happen. If $|S| \leq 2k - 2$, then at most $|S|/2 = (k-1)$ oracle queries are required. Otherwise, when $S = 2k$, both the leaf vertices are in S, and we can figure out the second leaf for free without making any oracle query and remove both leaf vertices from S, making $|S| = 2k-2$. In this case too, at most $k - 1$ oracle queries are sufficient. Thus, when n is odd, the total number of queries required are at most $n - 1 + (k - 1) = 3n/2 - 5/2 < 3n/2 - 2$.

Thus, at most $3n/2 - 2$ oracle queries are enough to find the Hamiltonian path in a complete graph. The running time of algorithm is also clearly $O(n)$. □

We conjecture that $3n/2 - 2$ queries are necessary to find the Hamiltonian path in a complete graph.

3.5 Reconstructing Hamiltonian Path in General Graphs

In general when the graph is not complete, then we are restricted by the oracle queries we can ask, and so it is not obvious whether we can find the Hamiltonian path or how we can find a leaf.

3.5.1 Assuming a Leaf Vertex is Given

We first look at a simple case when a vertex is known to be a leaf vertex of the Hamiltonian path H, but the graph is not necessarily complete. This case is simpler, because all the vertices lie on the same of side of this leaf vertex on H, and hence knowing their distance from that leaf is enough to find out where all the vertices lie on the Hamiltonian path.

The following pseudocode, which has a flavour of the standard BFS algorithm, explains the algorithm.

Algorithm 2: Hamiltonian Path in $G(V, E)$ with s as one of leaf vertices

```
1  Color all vertices except s as white and color s as grey ;
2  Maintain all grey vertices in a priority queue with priorities as distances
     from s ;
3  Initially the queue contains s with value 0 ;
4  while there exists a white vertex do
5      delete the grey vertex v with the smallest distance value from grey
         queue ;
6      for u ∈ N(v) do
7          Perform oracle query vu if u is white and let d(v, u) be its distance ;
8          Insert u into grey queue with distance d(s, u) = d(s, v) + d(v, u) ;
9          Change u's color from white to grey;
10     end
11 end
12 The distance values from s to every vertex gives the Hamiltonian path
```

It is easy to argue the query complexity and we will do that first. Whenever a distance query is made to the oracle by the algorithm, a white vertex becomes

grey, and since there are $n-1$ white vertices initially and a grey vertex never becomes white, the number of oracle queries made is $n-1$. The algorithm may actually look at all edges, as in step 7 of the pseudocode, we may need to check all neighbors of v, though we query only for white neighbors. Also in steps 5 and 8, we delete and insert a vertex into the grey priority queue. As the values of the priority queue are simply integers in the range 1 to n, we can use an integer priority queue [22] taking $O(\lg \lg n)$ time for insertion/deletion per vertex. Thus, the algorithm can be implemented in $O(m + n \lg \lg n)$ time.

To argue correctness, note that a vertex is marked grey if its distance from s is known, and is removed from the grey queue if the distances to all its neighbors in E have been known (directly or indirectly). The following Lemma shows that every vertex eventually learns its distance from s on H.

Lemma 3.5.1. *Before the i^{th} iteration of the while loop, the algorithm has correctly learnt all the vertices which lie at distance less than i from s.*

Proof. We prove this by induction on i. The claim is clearly true when $i = 1$. Every pair of vertices x and y which are adjacent in H also share an edge in E, so in the first iteration of the while loop, the algorithm learns the vertex which is at distance 1 from s, and so the claim is true for $i = 2$ as well. Assume the claim to be true up to the j^{th} iteration where $j \geq 2$.

Consider the $(j+1)^{th}$ iteration. It is possible that the vertex at distance $j+1$ from s is already in the grey queue in which case, we are already done, as the claim is true after $(j+1)^{th}$ (and before $(j+2)^{th}$) iteration. Otherwise, as the algorithm has gone through j-iterations and in each iteration a vertex (starting at distance 0) is removed from the grey queue, the shortest distance vertex being considered is the vertex at distance j from s. As the vertex at distance $j+1$ is its neighbor in the graph, the white colored vertex at distance $j+1$ from s will be probed and found in this iteration, thus proving the Lemma, which leads to the following theorem.

□

Theorem 3.5.2. *Given a vertex v which is the leaf node of the Hamiltonian path H, one can find H using $n-1$ distance queries using Algorithm 2 and $O(m + n \lg \lg n)$ time where m is the number of edges in the graph.*

Corollary 3.5.3. *Given an incomplete undirected graph on n vertices, with at least one vertex v adjacent to all other vertices, the hidden Hamiltonian path H can be found using at most $2n-2$ distance queries and $O(m + n \lg \lg n)$ time.*

Proof. Make all the $n-1$ distance oracle queries of the form vv', where $v' \in N(v)$ and find a vertex at maximum distance from v (say u). Vertex u is guaranteed to be a leaf vertex of H. Proof follows from application of Theorem 3.5.2.

□

3.5.2 When a Leaf Vertex is Not Given

In this subsection, we deal with the general case of graphs when the leaf vertex is not given as a part of input. We give an algorithm to find the Hamiltonian path without having the need to know a leaf vertex using $2(n-1)$ oracle queries.

The pseudocode for the algorithm is given in Algorithm 3. We start with a Hamiltonian path H on $2n+1$ vertices which is a placeholder for the vertices of the graph. We place vertices in H as and when we know their (relative) positions. During the execution of the algorithm, the vertices are colored with white, grey or black. White vertices are vertices which have not been placed on H'. Grey vertices are vertices which have been placed on H', but their distances with all their neighbors have not yet been learnt. Black vertices are vertices which have been placed on H' and have learned their distance from all the vertices in their neighborhood in E .

The set Q denotes the set of white vertices which have participated in exactly 1 oracle query and are not sure about their location in H'. For all $v \in Q$, I stores the result of the corresponding edge query made by v.

Now we analyze the correctness and query complexity of pseudocode in Algorithm 3.

Correctness: The algorithm always maintains the invariant that the vertices placed on H' do not contradict any distance query made upto that point. This is true initially as m was an arbitrary vertex. During the first iteration of "while" loop, steps 11-13 will label either one or two vertices on H'. If exactly one vertex is labelled, then m is a leaf vertex, as only leaf vertices have exactly one vertex at distance one. Otherwise, if two vertices are labelled, then both the vertices are

at distance 1, and the distance between them is 2. Thus at this point all vertices which have been labelled do not contradict any distance query.

We shall look at a Lemma which we need to prove the correctness and to find the query complexity.

Lemma 3.5.4. *Whenever a distance query is made by the algorithm, it is made between two vertices such that one is grey and the other is white.*

Proof. An oracle query always happens between $v \in S_u$ and the pivot vertex u. As evident from step 7, the vertex $v \in S_u$ has white color, whereas u has grey color in step 29. □

Lemma 3.5.5. *Every vertex v when it is labelled on H' has correct location with respect to the other vertices on H'.*

Proof. If v was labelled in H' from Q, two labelled vertices u and u' (since v is white, u and u' are grey by Lemma 3.5.4) in H' know their distance from v and since they are also labelled in H' the distance $d(u, u')$ in H can also be inferred from H'. These three distances $d(u, v)$, $d(u', v)$ and $d(u, u')$ define a unique unlabelled vertex in H' where v can be placed.

If v was not in Q, then labelling happens only when $d(u, v) = 1$ for a pivot u. For the first pivot m, if m was also a leaf, then there is only one vertex at distance 1, and it is correctly labelled in H'. If m is a non-leaf vertex, then both the neighbors of m are discovered and are correctly labelled without contradicting any previous labelling and distance queries, since they are the first two vertices to be labelled in H' after u. If some vertex u other than m is a pivot, then one of its neighbors is already been labelled, because the other vertex at distance 1 from u has already found its location on H' (as the other vertex was the pivot in some previous iteration of the "while" loop). Otherwise, u would not be the nearest unmarked labelled vertex from m.

In either case, if the relative positions of the labelled vertices in H' is consistent with the one in H prior to labelling of v in H', then it will also be consistent after labelling of vertex v in H', since v has only one possible location to be correctly labelled.

Thus, each stage of labelling correctly maintains the relative distances between the labelled vertices. □

Algorithm 3: Hamiltonian Path in $G(V, E)$

Create a path graph H' on $2n + 1$ unlabelled vertices ;
Pick a vertex m and label the middle vertex of H' as m ;
Color m with grey and rest of vertices with white;
Initialize Q and I to empty;
A pivot u is assigned value m;
while *n vertices are not labelled in H'* **do**
    Let S_u be neighbors of u which are assigned white color;
    for $v \in S_u$ **do**
        if $v \notin Q$ **then**
            Make oracle query uv and get $d(u, v)$;
            if $d(u, v) = 1$ **then**
                Label an unlabelled neighbour of u in H' to v;
                Color v with grey;
            end
            else
                Add v to Q;
                Add $[u, v, d(u, v)]$ to I;
            end
        end
        else
            Make oracle query uv to get $d(u, v)$;
            Find the unique tuple (say $[u', v, d(u', v)]$) in I which has the distance of v from some vertex u';
            Find the unique vertex in H', which lies at distance $d(u', v)$ from u' and $d(u, v)$ from u and label it as v;
            Remove v from Q and $[u', v, d(u', v)]$ from I;
            Color v with grey;
        end
    end
    Color u with black;
    Find a nearest grey vertex from u on H' and make it the new u;
end
Remove the remaining $n + 1$ vertices from H' which have not been labelled;
Output H';

All that needs to be shown is that every vertex in V is actually labelled in H'.

Lemma 3.5.6. *At the end of every iteration of "while" loop, the pivot knows at least one of its neighbors in H'.*

Proof. Initially, when m is pivot it knows both the vertices which are its neighbors in H' and their location in H'. During the first iteration of while loop, one of the neighbors of m is pivot (say m') , which means that m' knows one of vertices which is its neighbor on H'. Similarly at the i^{th} iteration, the pivot knows one of its neighbors on H', since it was the pivot in previous iteration of while loop. In case the previous pivot was a leaf in H', the new pivot is the nearest grey neighbor to it, and it is the grey neighbor of m (not m' as m' has become black) , and that vertex also knows one of its neighbors (m) and its location in H'. □

Lemma 3.5.7. *All vertices $v \in V$ are labelled in H' by the $(n-1)^{th}$ iteration of "while" loop in Algorithm 3.*

Proof. Every vertex (except m) is initially colored white. If v is the neighbor of current pivot of an iteration of while loop, then v is either already labelled in H' and has grey color, or lines $11-13$ ensure that v is labelled besides pivot at distance 1 correctly (as the other vertex at distance 1 from pivot is already known by Lemma 3.5.6) and hence colored grey.

Thus, after every iteration of "while" loop, at least the neighbors of pivot (in that iteration) in H' are labelled, which proves the Lemma.

□

The two lemmas, Lemma 3.5.5 and Lemma 3.5.7 together prove the correctness of the Algorithm 3.

Query Complexity: The algorithm 3 performs atmost $2(n-1)$ oracle queries to find the underlying Hamiltonian path H.

Lemma 3.5.8. *When a vertex v does not exist as a label (i.e. v is white) in H', it can engage in at most two distance queries, after which it gets a label in H' (and gets grey color).*

Proof. As evident from steps 11-13 in Algorithm 3, some vertices which do not occur as a label in H' become a label in H', after just one oracle query, if the distance oracle returns value 1.

The only other place, the algorithm labels a vertex in H' is at step 25 . This "else" case is executed, only if v exists in Q (which means it has engaged in exactly 1 distance query till this point) , and at the end of the loop, a vertex is labelled with v in H', after v engages in its 2^{nd} distance query and gets grey color. Thus, a vertex which does not exist as a label in H', can engage in at most 2 distance queries, after which it exists as a label in H'.

□

Lemma 3.5.4 shows that neither two grey vertices nor two white vertices engage in any distance query, during the execution of the algorithm 3. Thus, Lemma 3.5.4 and 3.5.8, together show that the algorithm performs at most $2(n-1)$ distance queries, (m is colored grey without making any queries) , which proves the following theorem.

Theorem 3.5.9. *Given an undirected graph $G(V, E)$, one can find H using $2(n-1)$ distance queries.*

3.6 Lower Bounds

This section gives a simple adversary strategy to show that $n-1$ distance queries are necessary to find both the leaf vertices of the Hamiltonian path, and hence gives a lower bound on number of oracle queries to find the Hamiltonian path.

The adversary maintains two sets of elements called P and S. S contains all the vertices which have not participated in any distance oracle query. P contains a collection of undirected paths $p_1, p_2...p_i$. Initially all the vertices are in set S and P is empty. Adversary strategy is as follows. If a distance oracle query (u, v) is made, three possible situations can happen.

1. $u, v \in S$:-

   Oracle answers 1 as the distance between them. Then it creates a path in P, with an undirected edge uv, removing both u and v from S.

2. $u \in S$ and $v \notin S$:-

   Oracle finds a unique path in which v belongs (say p_i). Suppose l_1 is one of the leaves of p_i and d is distance of v from l_1, then algorithm answers $d+1$ as distance between u and v. It also gives away that distance between u and l_1 is one.

3. $u, v \notin S$ and lie on same path p_i in P:-

   The adversary answers the actual distance between u and v in p_i.

4. $u, v \notin S$, and they lie on two distinct paths p_i and p_j such that u lies on p_i and v lies on p_j:-

   Let l_i be a leaf of p_i and its distance from u on p_i be d_i and l_j be a leaf of p_j and its distance from v on p_j be d_j. The adversary answers the distance between u and v to be $[d_i + d_j + 1]$. The adversary also gives a free answer between vertices l_i and l_j to be 1, which combines the two paths to become one.

Correctness and Consistency: Whenever a comparison occurs between two paths, they combine to become a single path. This means that if two vertices lie on different paths, no distance oracle query has been made between any two vertices of these paths(as shown in case 4). Whenever two paths combine to form a new path, the two paths are connected at their leaves, which doesnt contradict any answer the adversary may have given. Same can be argued when a path and a single vertex in S are combined to form a new path. Thus, the adversary ensures the consistency and correctness is maintained during the entire course of the algorithm.

Query Complexity: Suppose an algorithm terminates, while there are at least two paths in P, then the adversary has at least four distinct pairs of leaf nodes of the Hamiltonian path (and four distinct Hamiltonian paths), consistent with its previous answers. Thus a correct algorithm cannot terminate with two or more paths in P. Similarly, if an algorithm terminates while there are two or more vertices in S, the adversary has at least four distinct pairs of leaf nodes of the Hamiltonian path (and four distinct Hamiltonian paths), consistent with his previous answers.

Thus a correct algorithm can have atmost one path in P and one vertex in S, when it terminates. Suppose, when the algorithm terminates, the adversary has one path in P and one vertex in S. In that situation, the adversary can two distinct Hamiltonian paths consistent with previous answers having two different pairs of leaf nodes.

Thus, a correct algorithm has to have zero vertices in S and one path in P, when it terminates.

To create a path with p vertices, exactly $p-1$ oracle queries are forced by the adversary. Thus when a correct algorithm finally terminates, it ends with one path of n vertices, which needs $n-1$ oracle queries.

This lower bound in essence shows that finding both the leaf vertices needs at least $n-1$ queries.

It is important to note that n queries in worst case(provided that the graph is complete) to find both the leaves.

3.7 Concluding Remarks

We show that the using distance oracle, the Hamiltonian path hidden in an undirected graph can be found using $O(n)$ distance oracle queries only, irrespective of whether the input undirected graph is complete or not. It would interesting to see whether $3n/2$ oracle queries are necessary to find the Hamiltonian path in complete graphs, or the trivial lower bound of $n-1$ oracle queries can actually be achieved.

A leaf vertex can be found using $n-1$ oracle queries and finding both the leaves (or the median) can be done using n oracle queries, when the input graph is a complete graph. It is not clear whether the same number of oracle queries are enough in the case of general graphs.

Since reconstructing path graph can be done in $O(n)$ time irrespective of whether the graph $G(V, E)$ is complete or not, it is also interesting to figure out whether reconstructing d-degree trees can also be done in $\tilde{O}(n^{1.5})$ time, even when the oracle queries are restricted to edges of the input graph which is not complete.

Chapter 4

Sorting and Selection Using Equality Comparisons

4.1 Introduction

In this chapter we consider sorting and selection problems when the input sequence is not necessarily from a totally ordered set and so there is no notion of an inherent linear or total order.

The only way the relation between a pair of elements is determined in this scenario is by making equality comparisons. While this is a natural variant that occurs when dealing with heterogenous sets of elements or elements that do not have a total order (say, for example, the elements are subsets of a universe), to the best of our knowledge the only problem studied extensively in this model is the problem of determining the majority element (an element that appears at least $\lceil n/2 \rceil$ times) if it exists, and there is a classical linear time algorithm for this [23]. Exact comparison complexity including upper and lower bounds, and average case complexity of the majority problem have been studied [24, 25, 26, 27, 28].

In this chapter, we explore the natural problem of determining a mode (a most frequently occurring element), of sorting (determining the frequency of every element) and of finding a least frequent element in this model. We show that $\Theta(n^2/m)$ (equality) comparisons are necessary and sufficient to find an element that appears at least m times. This is in sharp contrast to the bound of $\Theta(n \log(n/m))$ [29]

bound in the traditional comparison model. The lack of transitivity of the inequality comparisons throws interesting challenges.

We begin the development of our mode finding algorithms in Section 4.2 with a simple algorithm that takes at most $2n^2/m$ comparisons where m is the frequency of the mode. This appears in Section 4.2.1. Then, by generalizing a classical majority algorithm due to Fischer and Salzberg [28], we improve the number of comparisons to at most $(3/2)(n^2/m) + O(n^2/m^2)$ in Section 4.2.2. Our final algorithm in Section 4.2.3 takes at most $n^2/m+n$ comparisons and can be implemented in $\tilde{O}(n^2)$ time. Section 4.4 proves asymptotically matching lower bound for finding a mode.

In Section 4.3 we discuss the sorting problem where one needs to determine all the distinct elements and their frequencies in the given input. This has applications, for example in Graph Mining, where we are given a collection of graphs, and we need to group them into those that are isomorphic to each other, and the only oracle we know is an algorithm to detect whether two graphs are isomorphic or not. We give upper bounds for sorting and finding a least frequent element, and also give lower bounds when all elements have the same frequency. Finally, Section 4.5 concludes with remarks and open problems.

4.1.1 Related Work

As mentioned earlier, we know of only the majority problem [23] studied extensively in the equality comparison model. Demaine, Lopez-Ortiz and Munro [30] studied the (lower and upper bounds on the) number of passes required to find (a superset of) elements that appear(s) at least $n/(k+1)$ times in the equality comparison model. Our first algorithm for mode in Section 4.2.1 is essentially the same algorithm though we analyze the number of comparisons (as opposed to the number of passes). In one of the earliest papers studying optimal algorithms on sets, Reingold [31] proved lower bounds for determining the intersection/union of two sets if only $=, \neq$ comparisons are allowed. Regarding mode, Munro and Spira [32] considered optimal algorithms and lower bounds to find a mode and the spectrum (the frequencies of all elements), albeit in the three way comparison model. Misra and Gries [33] gave algorithms to determine an element that appears

at least n/k times for various values of k, in the three way comparison model.

4.2 Finding All Modes (or Elements With Specific Frequency)

Given a set of elements, a *mode* of the input is defined as any element which occurs the maximum number of times in the input. The input can have several modes. A natural randomized algorithm to find a mode (given its frequency m) in this model is to pick a random element and find its frequency by comparing it with all other elements. If m is the frequency of a mode, then the probability that this algorithm picks a mode in any given round is m/n, and hence in about n/m rounds, the algorithm finds a mode with high probability. As it makes $n-1$ comparisons in each round, the expected number of comparisons is around n^2/m. A high confidence bound can then be shown for this randomized approach. We show that this bound of $O(n^2/m)$ is achievable by a deterministic algorithm even without the knowledge of m. In addition, we give an adversary argument to show that $\Omega(n^2/m)$ comparisons are necessary.

In this section, we provide three algorithms to find a mode, each subsequent algorithm improving upon the earlier one in terms of the number of comparisons performed. The first two algorithms not only use $O(n^2/m)$ comparisons, but also spend only $O(n^2/m)$ time for the rest of the operations. The first one takes at most $2n^2/m$ comparisons, while the number of comparisons made by the second one is $3n^2/2m + O(n^2/m^2)$. The first one is relatively simple to argue correctness, and the second algorithm generalizes a classical majority finding algorithm. Both these algorithms have a 'selection phase' where a candidate set of elements for a mode are selected, and a 'confirmation phase' where the candidates are confirmed and those who pass the confirmation tests are output. The third algorithm uses at most $n^2/m + n$ comparisons although our implementation uses $\tilde{O}(n^2)$ time for the other operations.

For now, we assume that m is known, and later, we explain how this assumption can be removed.

4.2.1 A Simple Mode Finding Algorithm

The first phase of the SimpleMode algorithm in this subsection is essentially the one (called FREQUENT) that appears in [30].

Let k be the smallest integer such that $\lfloor n/k \rfloor \leq m-1$, i.e., $\lfloor n/k \rfloor \leq m-1 < \lfloor n/(k-1) \rfloor$. Let $a_1, a_2, \ldots a_n$ be the given list of n elements. We give an algorithm that finds all elements with frequency more than $\lfloor n/k \rfloor$ from an input of size n. The pseudocode description is given below. Here B is a set of distinct elements with some frequencies associated with each element.

Algorithm 4: *SimpleMode*(n)

```
1  Initialize B = {a_1}; i = 1;
2  while i < n − 1 do
3      i = i + 1
4      if value of a_i already appears in B then
5          increment the frequency of the value that equals a_i in B
6      end
7      else if add a_i to B with frequency 1 then
8          if the number of distinct elements in B is k then
9              decrement the frequency of each element in B;
10             delete all elements with (the new) frequency 0.
11         end
12     end
13 end
14 Find frequency in the entire list, of all elements (if any) of B,
15 Output the elements whose frequency over entire input is at least m.
```

We show that the SimpleMode algorithm finds all elements with frequency more than $\lfloor n/k \rfloor$ using at most $2n(k-1)$ comparisons. As $k \leq n/m+1$, the total running time of the algorithm is at most $2n^2/m$.

Suppose after every decrement, one copy of each of the elements is placed in a (separate) set. Then as each set has k elements, the total number of such sets, excluding B, is at most $\lfloor n/k \rfloor < m$. Also each of the sets has distinct elements. So if an element has frequency more than $\lfloor n/k \rfloor$, then it must have a copy in the final set B. Thus all elements with frequency at least m have a copy in B.

Every new element (after the first $k-1$ distinct elements) is compared with at most $k-1$ distinct elements of B for a total of $(n-k+1)(k-1)+(k-1)k/2 = (k-1)(n+1-k/2)$ comparisons. Also finally B has at most $k-1$ distinct elements that are compared with the remaining elements for a total of at most $n(k-1)$ comparisons for the confirmation phase. This results in overall at most $2n(k-1)$ comparisons. As $m \leq n/(k-1)$, we have $k-1 \leq n/m$ and hence the number of comparisons made is at most $2n^2/m$. This calculation covers any bookkeeping costs which might be incurred during the running of the algorithm. Thus we have the following theorem.

Theorem 4.2.1. *Given a multiset of n elements and a frequency m, we can find all elements with frequency at least m using at most $2n^2/m$ comparisons and $O(n^2/m)$ time.*

4.2.2 An Improved Algorithm Generalization of the Fischer-Salzberg Majority Algorithm

Fischer and Salzberg [28] developed an algorithm to find a majority element (if it exists) in a list of n elements using at most $3n/2-2$ comparisons. (Recall that a majority element is an element that appears more than $\lfloor n/2 \rfloor$ times.) We generalize this to find a mode to improve the coefficient of n^2/m in Theorem 4.2.1 to 3/2, resulting in at most $3n/2(\lfloor n/m \rfloor) + O(n^2/m^2)$ comparisons.

As before, let k be the integer such that $\lfloor n/k \rfloor \leq m-1 < \lfloor n/(k-1) \rfloor$. We give an algorithm (FSGeneralization) to find an element with frequency at least $m \geq \lfloor n/k \rfloor + 1$ using at most $3n(k-1)/2 + O(k^2)$ comparisons and $O(nk)$ other operations. When $k=2$, the problem degenerates to the majority problem and the bound becomes at most $3n/2 + O(1)$ as in the case of Fischer and Salzberg's algorithm. The main trick beyond the SimpleMode algorithm in the previous section is, to organize the sets that are obtained by decrementing the frequency of every element in a way to save comparisons in the confirmation phase.

Let $a_1, a_2, \ldots a_n$ be the given sequence of n elements. During the preprocessing phase, the FSGeneralization algorithm maintains a list L, and an array B with the following invariants:

- For any index i, $L[i] \notin S_i$, where $S_i = \{L[j],\ 0 < |j - i| < k\}$. I.e. in L, any set of consecutive k elements have distinct values.

- B, if non-empty, contains up to $k-1$ distinct elements, all of which appear in the last $k-1$ elements of L. Each cell in B contains the value of an element x, its last location in L and a frequency f which has the following property: the frequency of x in the input sequence (up to the point we have processed) is the frequency of x in L plus f. We maintain elements of B in a queue in the increasing order of their (last occurrence) locations in L.

Initially B is empty, and L contains the first element, a_1, in the input sequence. Then it processes each element a_i $(i > 1)$ in the sequence as follows.

- If a_i equals an element in the last $k-1$ positions of L, then if a_i appears in B, find its occurrence and increment its frequency. If a_i does not appear in B, then create a new entry for in B, with a frequency of 1, and set its last occurrence location to its last occurrence in L.

- If a_i does not equal any element in the last $k-1$ positions of L, then add a_i to L, and repeat the following step until the first element of B appears in the last $k-1$ positions of L.

  - Consider the element x in the first location of B. This is an element that has the least (last) location in L among those in B. The last location in L of that element is at least k positions away from the current last position. Add x to the end of L after decrementing its frequency in B. Remove it from B if its new frequency becomes 0. Otherwise update its last location to the current location in L and move the element to the last location of B that is kept sorted in ascending order of the "last location" of its entries.

Algorithm 5: *FSGeneralization*(n)

```
1  Initialize L = a_1; i = 1; B = ∅;
2  while i ≤ n − 1 do
3      i = i + 1;
4      if a_i equals an element in the last k − 1 elements of L then
5          if a_i ∈ B then
6              Find and increment frequency of a_i in B by one ;
7          end
8          else if a_i ∉ B then
9              Create a new entry with a_i as value, frequency as 1, and its
                position as last known position of a_i in L ;
10             Insert this entry in B in increasing order of the 'position' field of
                its elements;
11         end
12     end
13     else if a_i does not equal any of the last k − 1 elements of L then
14         Add a_i to the end of L;
15         while B[first] does not appear in the last k − 1 positions of L do
16             Decrease frequency of B[first] by 1;
17             Add B[first].value to the end of list L;
18             Update B[first].location = last + 1;
19             if B[first]'s frequency is nonzero then then
20                 Move B[first] to end of B ;
21             end
22             else
23                 Remove B[first] from B;
24             end
25         end
26     end
27 end
28 Comment: Confirmation Phase
29 for all elements in the last k − 1 elements of L do
30     Find their actual frequency in the input;
31     Output the elements whose total frequency in the entire input is larger
        than or equal to m;
32 end
```

At the end of the preprocessing phase, the input sequence is partitioned into L and B; i.e. every element of the input sequence is either in L or in B. Now, we claim that only the last $k-1$ elements of L are possible candidates for a mode, and so we check the frequency of each of those elements with all elements of L and output all those with frequency m. We call the step of finding the frequency of the last $k-1$ elements of L as the 'confirmation phase'.

The pseudocode in algorithm 5 gives the details. In the initial phase, the only comparisons made for each element are to test whether it equals an element in the last $k-1$ elements of L. We can use the last location entry to find its existence (if at all) in B which involves no comparisons with the element. So at most $n(k-1)$ comparisons and $n(k-1)$ other operations are made in the first processing phase of the algorithm to construct L and B. The confirmation phase, if done naively, takes at most $(k-1)n$ comparisons for a total of $2n(k-1)$ comparisons.

In what follows, we prove the correctness of the FSGeneralization algorithm and tighten the analysis to show a bound of at most $n(k-1)/2$ for the confirmation phase resulting in an overall comparison bound of $3n(k-1)/2$.

First it is clear that the algorithm maintains the two invariants (on L and B) mentioned above after every step. The invariant on L is maintained as and when we add new elements to L (either from the input sequence or from B). B always contains at most $k-1$ elements. We add an element to B only when a new element does not appear in the last $k-1$ elements of L, and when that happens, B had less than $k-1$ elements, as the elements of B appear in the last $k-1$ elements of L, and so the addition to B does not make the number of elements of B go above $k-1$. Also when we add new elements to L, if B is non-empty, we add elements from B to L ensuring that elements of B are in the last $k-1$ elements of L, for if an element of B is not in the last $k-1$ elements of L, then that element would have been added to the last element of L.

Suppose that the size of L is divisible by k. Then every element in L can appear at most $n/k \leq m-1$ times (as every consecutive k elements are distinct). And hence B has to be non-empty (for a mode to appear m times), and the only candidates for elements appearing more than $\lfloor n/k \rfloor$ times are those in B which are anyway in the last $k-1$ elements of L by the invariant on B.

Suppose the size of L is not divisible by k. Then the only possible input

elements that appear more than $\lfloor n/k \rfloor \leq m-1$ times are those in B and those in the last $n - k(\lfloor n/k \rfloor)$ locations of L, and the last $k-1$ elements of L cover these.

This completes the correctness of the algorithm. Now we give a slight modification of the confirmation phase and give a careful calculation of the number of comparisons made in the confirmation phase and show it to be at most $n(k-1)/2 + O(k^2)$.

The Confirmation Phase

For any element in the last $k-1$ locations of L that is not in B, the confirmation phase starts by comparing it with an element that is k locations apart to the left of it (as we know that the intermediate $k-1$ elements are distinct from it). For an element in B, the confirmation phase starts with its copy in the last $k-1$ locations and continues as above. And during the confirmation phase, if we find an element which is not equal to the element being compared, then we move left by a position and continue the comparison. And if we find an element which is equal to the element being compared, then we skip $k-1$ positions to the left and continue our comparison.

Let $\ell \leq k-1$ be the number of distinct elements in B and let $f_1, f_2, \ldots f_\ell$ be their respective frequencies in B at the end of the algorithm. Let $f = \sum_{i=1}^{\ell} f_i$.

Lemma 4.2.2. *Let $\ell \leq k-1$ be the number of distinct elements in B and let $f_1, f_2, \ldots, f_\ell$ be their respective frequencies in B at the end of the FSGeneralization algorithm. Let $f = \sum_{i=1}^{\ell} f_i$. The number of comparisons done by candidates from B in the confirmation phase is at most $f(k-1) + \ell(n/k - f) + (k-1)^2$.*

Proof. Let b be an element of B with frequency f_1. Then for b to qualify as a mode, it should have at least $\lfloor n/k \rfloor - f_1$ copies in L. View L as a contiguous sequence of k-sized blocks. There are $\lceil (n-f)/k \rceil$ such blocks. Hence b may not be present in at most $\lceil (n-f)/k \rceil - 1 - (\lfloor n/k \rfloor - f_1)$ such blocks and hence may get not-equal outcomes in comparisons with elements in these blocks for a total of at most $k(\lceil (n-f)/k \rceil - 1 - \lfloor n/k \rfloor + f_1)$ comparisons with not-equal outcomes. It is easy to see that it may make at most $k-1$ more comparisons with not equal outcomes, and has to get at least $\lfloor n/k \rfloor - f_1$ equality comparisons. If the number of comparisons with not equal outcomes or equal outcomes exceeds these quantities, we can stop the confirmation phase of that element.

Now summing these comparisons for every element of B, the number of comparisons made by elements of B in the confirmation phase is at most
$\sum_{i=1}^{\ell}(\lfloor n/k \rfloor - f_i + (k-1) + k(\lceil (n-f)/k \rceil - 1 - \lfloor n/k \rfloor + f_i))$

$$\begin{aligned} &\leq \ \ell(\lfloor n/k \rfloor(1-k) + (n-f) + k - 1) + f(k-1) \\ &\leq \ f(k-1) + \ell(n/k - f + k - 1) \\ &\leq \ f(k-1) + \ell(n/k - f) + (k-1)^2 \end{aligned}$$

□

Now we continue with the analysis of the total number of comparisons in the confirmation phase.
Case 1: $f \geq n/2$.
This implies that $|L| \leq n/2$. The confirmation phase finds the frequency of each of the last $k-1$ elements of L with the other elements of L which, in this case, will take at most $(k-1)|L| \leq n(k-1)/2$ comparisons.
Case 2: $n \bmod k < f < n/2$.
In this case, $|L| \leq k\lfloor n/k \rfloor$, and hence any element of L that is not in B, appears at most $\lfloor n/k \rfloor$ times and hence they don't qualify to become a mode.

From Lemma 4.2.2, the number of comparisons made by elements of B during the confirmation phase is at most $f(k-1) + \ell(n/k - f) + (k-1)^2$.
If $f \geq n/k$, then

$$f(k-1) + \ell(n/k - f) + (k-1)^2 \leq f(k-1) + (k-1)^2 < n(k-1)/2 + (k-1)^2.$$

If $f < n/k$, then

$$\begin{aligned} f(k-1) + \ell(n/k - f) + (k-1)^2 \ &< n(k-1-\ell)/k + (\ell n)/k + (k-1)^2 \\ &= n(k-1)/k + (k-1)^2 \\ &\leq n(k-1)/2 + (k-1)^2 \end{aligned}$$

Hence the number of comparisons made in this case is at most $n(k-1)/2 + (k-1)^2$.
Case 3: $f \leq (n \bmod k)$.

Let S be the set of all the elements which occur in last $k-1$ positions of L, but which do not occur in B. Let $|S| = p$. These p elements need at least $\lfloor n/k \rfloor$ additional copies for each of them to become a candidate for mode. I.e. each of them should appear in every block of L. Hence, the verification process for each element in S ends in at most $\lfloor n/k \rfloor$ equality comparisons and possibly at most $k-1$ comparisons with not equal outcomes (as more than k not equal outcomes would imply that the concerned element is not a candidate for majority), totally making at most $pn/k + p(k-1)$ comparisons.

The remaining $\ell = (k-1-p)$ elements in the last $k-1$ positions of L have a copy in B. From Lemma 4.2.2, the number of comparisons made by these ℓ elements for verification is at most $f(k-1) + \ell(n/k - f) + (k-1)^2$.

$$\begin{aligned} f(k-1) + \ell(n/k - f) + (k-1)^2 &= f(p+\ell) + \ell(n/k - f) + (k-1)^2 \\ &= fp + (\ell n)/k + (k-1)^2 \end{aligned}$$

Thus the total number of comparisons made by all the last $(k-1)$ elements of L is

$$\begin{aligned} pn/k + p(k-1) + fp + (\ell n)/k + (k-1)^2 &= ((p+\ell))n)/k + 3(k-1)^2 \\ &\leq n(k-1)/2 + 3(k-1)^2 \end{aligned}$$

Thus, the total number of comparisons is at most $(n(k-1))/2 + 3(k-1)^2$.

Thus in all three cases, the confirmation phase takes at most $(n(k-1))/2 + O(k^2)$ comparisons. So the total number of comparisons made by the algorithm in Theorem 2 is $(3n(k-1))/2 + O(k^2)$, which is at most $3n/2\lfloor n/m \rfloor + O(n^2/m^2)$ as $k - 1 \leq \lfloor n/m \rfloor$.

Theorem 4.2.3. *Given a multiset of n elements and a frequency m, all elements with frequency more than m can be found using $(3n/2)\lfloor (n/m) \rfloor + 3(n^2/m^2)$ comparisons and $O(n^2/m)$ time.*

When m is not known The (SimpleMode and FSGeneralization) algorithms presented in Theorems 4.2.1 and 4.2.3 can be made to find a mode, even if the value of m is not known. This is achieved by guessing the values of m, in turn guessing the values of k.

We keep running the algorithm for $k = 1, 2, 4...$ till we find a power of 2 (say x, i.e. $k = 2^x$) at which the algorithm returns a non-empty set of all elements with frequency greater than n/k thereby finding all elements which have the frequency of a mode. Clearly x, the smallest power of 2 for which the algorithm returns a non-empty set of elements with frequency great than $n/2^x$, is $\lceil \lg n/m \rceil$.

The total number of comparisons performed by the algorithm in Theorem 4.2.1, when m is not known would be $S = \sum_{i=1}^{x}(2n)(2^i)$ which is $4n^2/m$ (as $(k-1) \leq n/m$) which is about twice the number of comparisons taken by the algorithm in Theorem 4.2.1 when m is known. Similarly, we can show that the algorithm in Theorem 4.2.3 takes $3n^2m+O(n^2/m^2)$ comparisons to find a mode (with frequency m) when m is not known.

4.2.3 Finding Mode using $n^2/m + n$ Comparisons

In this subsection, we find a mode faster (in terms of the number of comparisons) than the previous two algorithms, and without the need to know the value of m. We also give an implementation of the algorithm that takes $\tilde{O}(n^2)$ other operations using some interesting non-trivial data structures.

Theorem 4.2.4. *There exists an algorithm that performs at most $n^2/m+n$ (equality) comparisons to find all modes and their frequency m, in a given list of n elements.*

Proof. Let $a_1, a_2, \ldots a_n$ be the given sequence of n elements, and consider them arranged clockwise in a circular list. The algorithm repeatedly compares, in sequence, every element with the first element (in the clockwise order) with which it has not determined its (equal/not equal) relation, until an element with frequency m is found. If m is not known to the algorithm, then the algorithm performs a sequence of rounds of comparisons until it finds an element that appears at least $\lceil (n-1)/k \rceil$ times at the end of k rounds.

Algorithm 6: $Mode(n)$

```
1  Initialize r = 0;
2  M = ∅(set of modes);
3  for i ← 1 to n do
4      eq(a_i) = {i}; neq(a_i) = ∅;
5  end
6  while M = ∅ do
7      r = r + 1
8      for i ← 1 to n do
9          find the next index j if any, starting from i + 1, wrapped around
            after n if necessary
10         such that j ∉ eq(a_i) ∪ neq(a_i) ;
11         if such an index j is found then
12             if a_i = a_j then
13                 ∀x ∈ eq(a_i) ∪ eq(a_j)
14                    eq(a_x) ← eq(a_i) ∪ eq(a_j) ;
15                    neq(a_x) ← neq(a_i) ∪ neq(a_j) ;
16                 ∀x ∈ neq(a_i) ;
17                    neq(a_x) ← neq(a_x) ∪ eq(a_i) ;
18             end
19              else if a_i ≠ a_j then
20                    ∀x ∈ eq(a_i), neq(a_x) ← neq(a_x) ∪ eq(a_j);
21                    ∀y ∈ eq(a_j), neq(a_y) ← neq(a_y) ∪ eq(a_i) ;
22             end
23         end
24         else
25             Add a_i to M.
26         end
27     end
28 end
29 Output M;
```

We show that the (circular order) sequence in which the comparisons are made

achieves the desired upper bound. The pseudocode 6 describes the algorithm.

We define a "round" as a sequence of comparisons which are tracked by variable r, in which every element which does not know its actual frequency in the input has initiated exactly one comparison in clockwise order of the input. At the end of the algorithm set M is output, which has all the elements which have the maximum frequency. In the algorithm $eq(a_i)$ corresponds to the set of indices of all elements that are known to the algorithm to be equal to a_i, and similarly $neq(a_i)$ corresponds to the set of elements known to be not equal to a_i. We refer to the comparison made in each round as the one *initiated* by a_i (in line 8) and associate such a comparison with a_i (note that a_j will also, by the same token, make one such comparison in that round which will be associated with a_j). It is clear that the algorithm maintains the invariant that if the algorithm knows that $a_i = a_j$, then $eq(a_i) = eq(a_j)$ and $neq(a_i) = neq(a_j)$. Hence we could keep these two lists for each group of elements that are known to be equal as one pair of lists instead of keeping them with each element. In what follows, we will continue to assume that every element has these two lists available.

As the algorithm performs at most n comparisons in each round r, and stops in $\lceil (n-1)/m \rceil$ rounds, the algorithm performs at most $n\lceil (n-1)/m \rceil \leq n(n+m-2)/m = (n^2-2n)/m + n < n^2/m + n$ comparisons.

The rest of the proof gives the correctness of the algorithm. We show that the first time an element that appears at least $(n-1)/r$ times at the end of r rounds, it is a mode. First we show that if an element appears m times, then the element will be discovered (to have exactly m copies) in at most $\lceil (n-1)/m \rceil$ rounds. We deal with the case $m = 1$ first where we can show a slightly better bound.

Lemma 4.2.5. *When all elements are distinct (i.e. when $m = 1$), the algorithm determines this in $\lfloor n/2 \rfloor$ rounds and the number of comparisons made by the algorithm is $n(n-1)/2$.*

Proof. When n is even, each element gains information regarding two new elements in each round (one due to the comparison initiated by the element, and another due to the comparison initiated on this element). So by round $n/2-1$, all elements have discovered their relation with all other but one element of the input. So in one more round of $n/2$ comparisons, all relationships will be found.

Thus the total number of comparisons made in this case is $(n/2-1)n+n/2 = n(n-1)/2$.

Similarly when n is odd, the algorithm takes $(n-1)/2$ rounds for each element to find its relation with the rest of the elements. Thus, total number of comparisons made in this case is also $n(n-1)/2$. □

We prove the correctness for $m \geq 2$ through a series of lemmas. The first lemma follows from the fact that we maintain the 'invariant' for the sets $eq(a_i)$ and $neq(a_i)$ for each i, throughout the algorithm.

Lemma 4.2.6. *At any point in the algorithm, if $a_i = a_j$ has been discovered by the algorithm, then $eq(a_i) = eq(a_j)$ and $neq(a_i) = neq(a_j)$.*

To understand the next lemma (and hence the total runtime of the algorithm), consider the (lucky) situation where all equal elements are contiguous and so all groups have found (a lower bound for) their frequencies. Now to determine their exact frequency, all we need to do is to make *one* comparison between each pair of groups. But we may not be that lucky, as several wasteful 'not equal' comparisons may have been made between groups of (equal) elements before we even discover that a pair of elements in a group are equal. Lemma 4.2.7 says that because of the *order* in which we make the comparisons, the algorithm will not do too many wasteful comparisons.

Lemma 4.2.7. *Let a_i, a_j be two elements such that $a_i = a_j$, and let $k \notin eq(i)$. If a_i initiates a comparison with a_k in a round, then a_j will subsequently not initiate a comparison with a_k (even if $a_i = a_j$ was not determined when a_i initiated a comparison with a_k).*

Proof. If $a_i = a_j$ has already been discovered by the algorithm when a_k was directly compared with a_i, then clearly the outcome of the comparison between a_k and a_i also gives the relation between a_k and a_j (as updated by the eq and neq sets) and so the algorithm will not compare a_j with a_k. The eq and neq sets to updated everytime they obtain a new information without any delay as this is crucial for the working of this lemma.

Suppose $a_i = a_j$ has not been discovered by the algorithm when a_k was compared with a_i. Suppose $k > i$. If $j < i$, then the way the algorithm makes the

comparisons, a_j would be compared with a_i (and be found equal and hence will learn its unequal relation with a_k from a_i's *neq* set) before comparing with a_k and hence will not initiate a comparison with a_k thereafter. Hence assume that $j > i$. Now, if $j < k$, then a_i would have initiated a comparison with a_j before initiating with a_k – a contradiction to the fact that $a_i = a_j$ has not been discovered when a_i was initiating a comparison with a_k. Now if $j > k$, then again a_j would initiate a comparison (in the wrap around) with a_i before initiating a comparison with a_k. Hence a_j will not initiate a comparison with a_k.

A similar argument proves the claim if $k < i$. □

A similar mirroring lemma also holds.

Lemma 4.2.8. *Let a_i, a_j be two elements such that $a_i = a_j$, and let $k \notin eq(i)$. If a_k initiates a comparison with a_i in a round, then a_k will subsequently not initiate a comparison with a_j (even if $a_i = a_j$ was not determined when a_k initiated a comparison with a_i).*

Lemma 4.2.9. *Let X be the set of a_is whose value is x, for some value x, and let $|X| = m \geq 2$. Then all elements of X together initiate at most $n-1$ comparisons in $\lceil (n-1)/m \rceil$ rounds and know their relation with every other element.*

Proof. From Lemma 4.2.7, all elements of X initiate together at most $n - m$ comparisons with elements not in X. As the equality relation is transitive, at most $m-1$ equality comparisons will be made among themselves to determine that they are all equal. Thus together elements of X initiate at most $n-1$ comparisons. Thus at least one element of X initiates at most $(n-1)/m$ comparisons and hence knows its relation to others in this many rounds (as otherwise it would have initiated more comparisons). Note that if any element of X has determined its relationship to all other elements, then all elements of X immediately know their relationship to all other elements. So in at most $\lceil (n-1)/m \rceil$ rounds, each element of X will determine its relationship with all other elements, and the set M contains x. □

Hence, if after r rounds, we find an element(say x) which appears at least $\lceil (n-1)/r \rceil$ times, then x is a mode. For, if there is an element whose frequency is more, then from Lemma 4.2.9, that element would have been discovered in the previous rounds. □

In fact, by modifying the while condition in line 6 of the algorithm 6 from $M = \emptyset$ to $r \leq \lceil n/k \rceil$, we have by the above lemma,

Theorem 4.2.10. *Given a list of n elements and an integer k, all elements (if any) with frequency at least k can be found using at most $n^2/k + n$ comparisons.*

Now we describe how to implement Algorithm 6 so that the bookkeeping operations besides the comparisons can also be performed efficiently. We can maintain all elements that are known to be equal to each other as a single set pointed to by elements in the set in the list. Similarly for each such group, we can maintain their *neq* set as a single set as well. As before, let $a_1, a_2, \ldots, a_n$ be the input sequence. Then, if two elements a_i and a_j point to different *eq* sets, then these *eq* sets are mutually disjoint. Also, the *neq* sets themselves are disjoint union of some *eq* sets. Now after every comparison, several *eq* sets may need to be updated. Recall from line 19 of the pseudocode that when the two elements(a_i and a_j) compared are found not to be equal, then the updation involves just performing a couple of union operations, and pointing these updated *neq* sets for both values a_i and a_j. However, when two elements being compared are equal (line 12 in the pseudocode), then apart from performing the union of their *eq* and *neq* sets, we need to update the *neq* sets of several elements (specifically those that have learnt to be not equal to one of the elements compared, see line 16).

If we represent the *eq* and *neq* sets as unsorted lists, then the naive way to implement these query and update operations requires $O(n)$ time when the result of a comparison is "not equal", and requires $O(n^2)$ time when the result of a comparison is "equal" (as the equality comparison results in up to $\Theta(n)$ set union operations over sets of size $\Theta(n)$). Since the number of "equal" comparisons in atmost $n - 1$ and number of "not equal" comparisons is atmost $O(n^2/m)$, this implementation takes $(n-1)n^2 + (n^2/m)n = O(n^3)$ time.

Using a different data structure for maintaining the *eq* and *neq* sets, we show how to implement the algorithm in $\tilde{O}(n^2)$ time (still using the same $O(n^2/m)$ comparisons).

In this implementation, every element a_i is assigned a vector v_i of length n, such that $v_i[j]$ stores the information about the comparison between a_i and a_j –

one of three values d (for "don't know"), e (for "equal") or ne (for "not equal"). We initialize all the vectors v_i such that $v_i[j] = d$ for all $i \neq j$, and $v_i[i] = e$. During the execution of the algorithm, we will change $v_i[j]$ to either e or ne, if the algorithm learns that the relation between the values a_i and a_j is "equal" or "not equal", respectively. Furthermore, we update the vectors after every comparison, to ensure that $v_i[j] = v_j[i]$ for all i and j. We maintain each of these vectors using a dynamic datastructure that supports *access*, *rank*, *select* and *update* operations defined below.

- The operation $access(v_j, i)$ returns the value $v_j[i]$.
- The operation $rank(v_j, i, \alpha)$ returns the number of occurrences of α among the first i elements of v_j.
- The operation $select(v_j, i, \alpha)$ returns the position of the i-th occurrence of α in v_j.
- The operation $update(v_j, i, \alpha)$ changes $v_j[i]$ to α.

We use the following simplified and rephrased version of the data structure by Munro and Nekrich [34] to support these operations on each of the vectors.

Lemma 4.2.11 ([34])**.** *A dynamic string over a constant sized alphabet can be stored in a datastructure that uses $O(n)$ bits and supports access, rank, select and update operations in $O(\lg n / \lg \lg n)$ time.*

Note that all the vectors initially start as sequence of all but 1 d's, and some indices get changed to ne or e during the course of the algorithm's execution and once they change they don't undergo any further change. We allocate $O(n^2 \lg n / \lg \lg n)$ charge to the datastructure, as $O(\lg n / \lg \lg n)$ charge is used anytime d is changed to ne or e in the vector.

In the i-th round of the algorithm, finding the index j such that $j \notin eq(a_i) \cup neq(a_i)$ is equivalent to finding the smallest index $j > i$ (considering wrap around) in v_i such that $v_i[j] = d$. This can be implemented in $O(\lg n / \lg \lg n)$ time using the rank and select operations. In algorithm 6, whenever an inequality is discovered between two values a_i and a_j, we go through all the positions with value e in v_i (using rank and select operations to skip the others and traverse only the e

values) and update the corresponding positions in v_j to ne, and vice-versa for v_j. Since the maximum number of e's in a vector is at most m, we spend at most $O(m \lg n / \lg \lg n)$ time, during each inequality comparison. Since there are atmost n^2/m inequality comparisons, the cost incurred by these inequality comparisons is at most $O(n^2 \lg n / \lg \lg n)$.

Whenever an equality is discovered between two values a_i and a_j, we spend $O(n)$ time to simply *merge* the vectors v_i and v_j – the merged vector v_i (also v_j) is obtained by setting $v_i[k] = v_j[k]$ if $v_i[k] = d$ (and leaving the other entries as they were), for $1 \leq k \leq n$ (Note that the d values may not have changed if the other vector also has d in the corresponding location, and so we cannot charge the cost to the cost of changing d to ne or e and hence we account for the cost here to get $O(n)$ cost). After the merge, both a_i and a_j point to the merged vector.

In addition, we need to implement line 16 of algorithm 6. For that, before merging the vectors, we also find out all the indices which have ne assigned in one set and d assigned in the other set (a simple scan is enough as we anyway charge $O(n)$ for the merge). For these indices, we update their vectors to change d to ne in the corresponding location and the $O(\lg n / \lg \lg n)$ charge is consumed. If $v_i[k] = v_j[k] = ne$ for some index k, it means that the element a_k already knows all the transitivity relations induced by the current comparison prior to making it, and does not require any change in the vectors. There are atmost $n - 1$ equality comparisons, thus the total cost incurred by these equality comparisons is at most $O(n^2)$ (the $\lg n$ charge is deducted from the initial charge and is not counted here again).

Thus, the total time spent is $O(n^2 \lg n / \lg \lg n) + O(n^2) = \tilde{O}(n^2)$, which gives us the following theorem.

Theorem 4.2.12. *Given a list of n elements, all elements (if any) with frequency at least k can be found using at most $n^2/k + n$ comparisons, using $\tilde{O}(n^2)$ time.*

We leave it as an interesting open problem to implement the above algorithm to take overall $O(n^2/k)$ time.

4.3 Sorting and Finding Least Frequent Element

Sorting problem in this model is defined as finding the exact frequencies of every distinct element in the input. Consider the naive algorithm that repeatedly picks an element a_i, which is known to be different from the elements whose exact frequencies have been discovered, and finds its frequency by comparing it with all elements not in $eq(a_i)$ and $neq(a_i)$ updating these two sets after every comparison. Let c be the number of distinct elements in the sequence, and let $x_1, x_2, \ldots x_c$ be the values of the elements, and let f_i, $i = 1$ to c be the number of times x_i occurs, where $f_1 \geq f_2 \geq f_3 \ldots \geq f_c$. In the worst case this algorithm will take at most $(n-1)+(n-1-(f_c))+(n-1-(f_c+f_{c-1}))+...+(n-1-(f_c+f_{c-1}-...+f_2)))$ comparisons. This is at most $nc - c - \sum_{i=2}^{c}(i-1)f_i = \sum_{i=1}^{c}(c-i+1)f_i - c = \sum_{i=1}^{c}(c-i)f_i + n - c$ which is at most $c(n-1)+n$ comparisons. Thus we have

Theorem 4.3.1. *Let c be the number of distinct values in the sequence, and let $x_1, x_2, \ldots x_c$ be the values of the elements, and let $f_1, f_2, \ldots f_c$ be the number of times x_i occurs, where $f_1 \geq f_2 \ldots \geq f_c$. Then there exists an algorithm to sort the list using at most $\sum_{i=1}^{c}(c-i)f_i + n - c$ comparisons.*

In what follows, we analyze the number of comparisons made by our mode algorithm (Theorem 4.2.4) if we run it until all frequencies are determined. Let c be the number of distinct elements in the list. Then, it follows from Lemma 4.2.8 that an element a_k never initiates a comparison with two elements a_i and a_j that are equal to each other, but are not equal to a_k. Thus every element initiates at most $c-1$ comparisons with elements not equal to it, and hence the total number of comparisons resulting in 'not equal' answer, made by the algorithm is at most $n(c-1)$. Along with the $n-c$ equality comparisons, the total number of comparisons is at most $c(n-1)$. In the following, we give a tighter upper bound for the number of comparisons made by the algorithm.

We begin with the following corollary that follows from Lemma 4.2.7 and Lemma 4.2.8.

Corollary 4.3.2. *Let x and y be two distinct values that occur f_1 and f_2 times respectively in the sequence, then the number of comparisons made by the algorithm between elements $a_i = x$ and $a_j = y$ is at most $2\min\{f_1, f_2\}$.*

Proof. Let A be the set of indices i such that $a_i = x$ and B be the set of indices j such that $a_j = y$. Draw a bipartite graph with two parts as A and B and orient an edge from i to j if a_i initiates a comparison with a_j. Lemma 4.2.7 says that the indegree of any vertex in this bipartite graph is at most one and Lemma 4.2.8 says that the outdegree of any vertex is at most one. Thus the number of edges, that corresponds to the number of comparisons made between A and B is at most $2\min\{f_1, f_2\}$. □

Theorem 4.3.3. *Let c be the number of distinct elements in the sequence, and let $x_1, x_2, \ldots x_c$ be the values of the elements, and let $f_1, f_2, \ldots f_c$ be the number of times x_i occurs, where $f_1 \geq f_2 \ldots \geq f_c$. Then the number of comparisons made by the mode algorithm to identify the frequency of every element is at most $2(\sum_{i=1}^{c} if_i) - n - c \leq c(n-1)$.*

Proof. Consider the number of comparisons made together by the set of all elements that are equal to x_c with all the elements outside this set. By Corollary 4.3.2, this number is at most $2f_c(c-1)$. In general, the number of comparisons made together by the set of elements that equal x_i with elements that equal x_j for all $j < i$, is at most $2f_i(i-1)$. Thus the total number of comparisons made by the algorithm is at most $2\sum_{i=1}^{c} f_i(i-1) = 2\sum_{i=1}^{c}(f_i i - f_i) = 2\sum_{i=1}^{c} if_i - 2n$. As the equality comparisons are transitive, the number of equality comparisons (made within each group) is at most $\sum_{i=1}^{c}(f_i - 1)$ which is at most $n - c$.

Thus the total number of comparisons made by the algorithm is at most $2\sum_{i=1}^{c} if_i - 2n + n - c = 2(\sum_{i=1}^{c} if_i) - n - c$

To show that $2(\sum_{i=1}^{c} if_i) - n - c \leq c(n-1)$, it suffices to show $2\sum_{i=1}^{c} if_i - 2n \leq n(c-1)$ or $\sum_{i=1}^{c} if_i \leq n(c+1)/2$.

Suppose c is odd. Then

$$\begin{aligned} n((c+1)/2) - \sum_{i=1}^{c} if_i &= \sum_{i=1}^{c}((c+1)/2 - i)f_i \\ &= \sum_{i=1}^{(c+1)/2-1} f_i((c+1)/2 - i) - \sum_{i=(c+1)/2+1}^{c} f_i(i - (c+1)/2) \\ &= \sum_{i=1}^{(c-1)/2} f_i((c+1)/2 - i) - \sum_{j=1}^{(c-1)/2} j(f_{j+(c+1)/2}) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{(c-1)/2} i(f_{(c+1)/2-i}) - \sum_{i=1}^{(c-1)/2} i(f_{i+(c+1)/2}) \\
&= \sum_{i=1}^{(c-1)/2} i[f_{(c+1)/2-i} - f_{(c+1)/2+i}] \geq 0
\end{aligned}
$$

The last inequality is true since every term in the summand is nonnegative as f_i's are in nondecreasing order. This proves the claim.

Suppose c is even. Then

$$
\begin{aligned}
n(c+1)/2 - \sum_{i=1}^{c} if_i &= \sum_{i=1}^{c}((c+1)/2) - i)f_i \\
&= \sum_{i=1}^{c/2} f_i((c+1)/2 - i) - \sum_{i=c/2+1}^{c} f_i(i - (c+1)/2) \\
&= \sum_{i=1}^{c/2} f_i((c+1)/2 - i) - \sum_{i=1}^{c/2} f_{i+c/2}(i - 1/2) \\
&= \sum_{i=1}^{c/2} f_{c/2-i+1}(i - 1/2) - \sum_{i=1}^{c/2} f_{i+c/2}(i - 1/2) \\
&= \sum_{i=1}^{c/2}(i - 1/2)(f_{c/2-i+1} - f_{c/2+i}) \geq 0
\end{aligned}
$$

This proves the theorem. □

Corollary 4.3.4. *The least frequent element in a sequence of n elements can be found in at most n^2/ℓ equality comparisons where ℓ is the frequency of the least frequent element.*

Proof. If ℓ is the frequency of the least frequent element, then every element appears at least ℓ times and hence the number c of distinct elements is at most n/ℓ. So if we apply our sorting algorithm (Theorem 4.3.3), we can sort, and hence find the least frequent element in at most $n(n-1)/\ell$ comparisons. □

4.4 Lower Bounds

In this section, we give lower bounds for finding a mode, the least frequent element and sorting with equality comparisons. The lower bounds are proved by an adversary argument. The adversary first models the input elements as vertices of a graph. Then for every comparison made by the algorithm, the adversary answers equal/not equal and constructs edges (based on the structure of the graph it has constructed till then) appropriately. The adversary answers in a way that it can instantiate the input elements, consistent with its answers.

4.4.1 Lower Bound for Finding a Mode

For giving a lower bound for finding a mode, we use Turán's theorem stated below.

Theorem 4.4.1 (see Theorem 1.1 in Chapter VI in [35])**.** *Let G be any graph with n vertices such that G has no K_{m+1}, the complete graph on $m+1$ vertices. Then the number of edges in G is at most $n^2/2 - n^2/2m$.*

Theorem 4.4.2. *At least $n^2/2m - n/2$ equality comparisons are necessary for any algorithm to determine a mode with frequency m from a given list of n elements, even if the algorithm knows m.*

Proof. The proof is by an adversary argument. The adversary models the n elements as n vertices of an undirected graph G. Whenever the algorithm makes a comparison between a pair of elements, the adversary will answer 'not equal' and draws an edge between the pair of vertices, if after the addition of this edge, there is still an independent set in the graph of size $m+1$ or more. Otherwise, the adversary will answer 'equal'.

As long as the modeled graph has an independent set of size $m+1$, the algorithm cannot determine a mode. For, the adversary can make any subset of the elements of the independent set of size $m+1$ as being equal to a mode.

Now when the algorithm gets the first 'equality' answer, the graph G has no independent set of size $m+1$. Hence $\bar{G}$, the complement of G has no clique of size $m+1$. Hence by Theorem 4.4.1, the number of edges in $\bar{G}$ is at most $n^2/2 - n^2/2m$. Hence the number of edges in G, that corresponds to the number of comparisons made by the algorithm, is at least $\binom{n}{2} - n^2/2 + n^2/2m = n^2/2m - n/2$. □

The frequency distribution in the above adversary strategy is the same as the one used to prove a lower bound in the one pass algorithm to find a mode in [30]. However appealing to Turán's theorem gives a simple argument for the number of comparisons.

Recall the classical (textbook) algorithm [23] to determine the majority, if it exists, of a list of n elements using at most $2n$ equality comparisons. Suppose the list has no majority element, one could ask for a pair of elements whose combined frequency is at least $\lceil n/2 \rceil$. We show, by an adversary argument, using Theorem 4.4.2 that finding such a pair requires $\Omega(n^2)$ comparisons.

The adversary sets up the input in such a way that one element appears exactly $n/2-2$ times (and hence not a majority) and every other element appears once or twice. The adversary gives away the element that appears exactly $n/2-2$ times. So the algorithm's task is to determine whether there is an element that appears at least twice among the remaining elements. The adversary answers the comparisons between the remaining elements as in the proof of Theorem 4.4.2 (with $m = 2$) forcing the algorithm to perform $\Omega(n^2)$ comparisons. Thus we have

Theorem 4.4.3. *Given a list of n elements, $\Omega(n^2)$ comparisons are necessary to determine whether there exists a pair of elements that together appear at least $n/2$ times.*

4.4.2 Lower Bound for Finding a Least Frequent Element

It is easy to see that any algorithm to determine whether the given n elements are distinct, using equality comparisons, requires $\Omega(n^2)$ comparisons. The adversary simply has to answer 'not equal' until the last comparison, and if the algorithm does not make any comparison between a pair of elements, the adversary has the option of making that pair equal.

However, this adversary or the one in the proof of Theorem 4.4.2 does not work well to prove a lower bound for finding the least frequent element, as an algorithm can pick an element and compare it with every other element receiving 'not equal' answers and can declare it to be the least frequent element using only $n-1$ comparisons. To obtain a better lower bound for the least frequent element, we resort to a different adversary.

Lemma 4.4.4. *In an n element list, $\Omega(n^2)$ comparisons are needed to find a least frequent element. Furthermore $\Omega(n^2)$ comparisons are required to sort a list of n elements even if the algorithm knows that every element appears exactly twice.*

Proof. Assume that n is a multiple of 4. The adversary first works with a list where every element appears twice. The adversary maintains a graph on n vertices where each vertex corresponds to an element and an edge between a and b corresponds to a *possibility* that a and b are equal. The adversary starts with a complete graph and whenever it answers 'not equal', it removes that edge. When it answers equal between a pair, it removes the pair of nodes (and their incident edges) from the graph and it also reveals that these two vertices are not equal to any of the remaining vertices in the graph.

The adversary strategy is to answer 'not equal to' for the comparisons as long as it can maintain at least a Hamiltonian cycle in the resulting graph (which will give at least two perfect matchings on the remaining elements). The adversary ensures this (we will prove this later) by answering 'not equal to' for the first $n/4 - 1$ comparisons any element is involved in. For a comparison involving a pair of elements where at least one element has been involved in $n/4 - 1$ other comparisons, it answers 'equal to' and deletes the pair of elements. Adversary follows this strategy until it has deleted $n/4$ pairs of elements. After that, the adversary reveals a perfect matching on the remaining vertices (for free). If the algorithm declares any of the deleted pairs of elements as a least frequent element, the adversary has the option of making the remaining elements distinct thereby proving the algorithm wrong. This completes the adversary strategy.

Now we make the following two claims to complete the proof.
Claim 1 This adversary strategy ensures that there is always a Hamiltonian cycle in the resulting graph (until $n/4$ pairs of vertices are deleted).

Proof. Dirac's theorem [36] states that if every vertex of a graph on $|V|$ vertices has degree at least $|V|/2$, then the graph has a Hamiltonian cycle. We show that, in the graph adversary maintains, the hypothesis of Dirac's theorem is maintained.

Initially when the graph is complete this is clearly true. After the first equality comparison is given by the adversary, the resulting graph has $n - 2$ vertices and every other vertex has been involved in at most $n/4 - 1$ comparisons, and so their

degree (in the entire graph) is at least $3n/4$. In the resulting graph, a vertex might have lost two degrees to those eliminated vertices, thus each resulting vertex has degree at least $3n/4 - 2 \geq (n-2)/2$ as $n \geq 4$.

Similarly when $i \leq n/4$ pairs of vertices are removed from the graph, the resulting graph has $n - 2i$ vertices and each of the vertices in the remaining graph has degree at least $3n/4 - 2i \geq (n - 2i)/2$ which impies that the graph has a Hamiltonian cycle. □

A corollary of Claim 1 is that the algorithm has no way of figuring out the matching until $n/4$ pairs are deleted (as the adversary has two choices of matching for the remaining elements).

Claim 2 The number of comparisons made by the algorithm is at least $n^2/16$.

Proof. The adversary gives an equality comparison and eliminates a pair of elements only when at least one of the paired vertices has engaged in $n/4$ comparisons. So to eliminate $n/4$ pairs of elements, the adversary forces $n^2/16$ comparisons. □

Claims 1 and 2 prove the theorem.

□

We generalize Lemma 4.4.4 to show the following.

Theorem 4.4.5. *In an n elements list, $\Omega(n^2/\ell^2)$ equality comparisons are necessary for any algorithm to determine a least frequent element with frequency ℓ even if the algorithm knows ℓ.*

Proof. Let n be a multiple of ℓ. The adversary gives away n/ℓ sets of size ℓ each and reveals that all the elements in each set are equal to each other. Now the algorithm's task is to determine whether any element of any of the sets has copies elsewhere. The adversary answers as in the proof of Lemma 4.4.4 treating each group as an element until it eliminates $n/4\ell$ pairs of groups. After that it declares that the remaining groups have no other equal element thereby making any element in any of the groups as a least frequent element with frequency ℓ. The algorithm cannot declare any element as a least frequent element before $n/4\ell$ pairs of the groups have been eliminated as the adversary has choice of 'matching' any group with another till that point. It follows from Lemma 4.4.4 that $\Omega(n^2/\ell^2)$ comparisons have been forced. □

Note that this bounds falls short of $O(n^2/l)$ upper bound in Corollary 4.3.4.

In case that the input consists of n/m distinct elements each occuring with m frequency, then finding the m copies of each of the distinct elements also needs at least $\Omega(n^2/m^2)$ comparisons, even when the adversary reveals this information to the algorithm.

Theorem 4.4.6. *In an n elements list, $\Omega(n^2/m^2)$ equality comparisons are necessary for any algorithm to find all modes or to sort if every element is a mode appearing m times even if the algorithm knows m.*

Proof. The adversary makes $2n/m$ groups of $m/2$ elements each and reveals that each group contains elements which are equal to each other. Given the hypothesis of the theorem, the $2n/m$ groups form n/m pairs of groups where all elements in each pair are equal. Now the algorithm's task is to determine the matching pair group for every group. The adversary answers like the adversary in Lemma 4.4.4 (treating each group as an element) and forces $\Omega(n^2/m^2)$ comparisons.

□

We note that this bound falls short of our upper bound of $O(n^2/m)$ proved in Theorem 4.2.4 to find all modes or to sort when all elements appear m times.The lower bounds in Theorem 4.4.5 and Theorem 4.4.6, have been improved to match with our upper bounds by Devanny, Goodrich and Jetviroj in [37].

4.5 Conclusions

We have determined (up to constant factors) the comparison complexity of finding a mode in a given list of n elements using only equality comparisons. There is a gap of a factor of 2 between upper and lower bounds (Theorem 4.2.4 and Theorem 4.4.2) and closing the gap is an interesting open problem. With respect to finding a least frequent element the gap between our upper and lower bounds is wider, but the lower bound has since been improved by Devanny, Goodrich and Jetviroj [37] to match with our upper bound asymptotically. Also, for sorting, when all elements have equal frequency, our lower bound of Theorem 4.4.6 has been improved in [37] to match with our upper bound.

Munro and Spira [32] gave a lower bound for sorting in the model of three way comparisons model, for sorting elements with different frequencies. Proving a lower bound in the equality comparisons model, extending the lower bound of [37] for the case when elements have different frequencies, to possibly match with our upper bound in Theorem 4.3.1 is another interesting direction.

Another open problem is to explore data structures to implement the algorithm in our best mode finding algorithm so that the rest of the operations also take $O(n^2/m)$ time.

Chapter 5

Selection in Tournaments

5.1 Introduction and Motivation

A *tournament* is a directed graph in which there is exactly one directed edge between every pair of vertices. As they model many practical scenarios (game tournaments, voting strategies), tournaments are well-studied in structural and algorithmic graph theory. Various structural and algorithmic properties of tournaments are known and Moon's [38] early monograph on this subject lead to much subsequent work in the area.

It is well-known that every tournament has a 'centre' or a *king* vertex, i.e. a vertex from which every other vertex can be reached by a directed path of length at most 2 (a maximum outdegree vertex of the tournament is one such vertex). A king is also called a 2-king. In this chapter, we look at the complexity of finding a king in a tournament by probing the adjacency matrix of the tournament.

This problem also arises in the context of selection problem, when the oracle that answers the comparison is faulty [39]. The oracle is faulty, only when the elements being compared have their difference below a fixed threshold $\delta > 0$. In real life, if the two teams playing against each other have similar level of skill, then either of the teams can win the match. This oracle is inspired by both imprecision in human judgment of values and also by bounded but potentially adversarial errors in the outcomes of sports championships. In this model, it has been shown[39] that it is not always possible to find the correct maximum element, and the only guarantee which can be given, is that there is a way to find an element which is

at most 2δ smaller than the correct maximum element. So instead of finding the maximum element, we try to find an element which is at most 2δ smaller than the maximum element. In other words, we try find to an element x such that $x > y - 2\delta$ for all the remaining elements y which belong to the input.

When our faulty oracle returns that $x > y$, we can say with certainity that $x > y - \delta$. Hence, when oracle answers that $x > y$ and $y > z$, it actually implies that $x > z - 2\delta$. If we map the elements to be compared as vertices of a graph, and we map the oracle's output $x > y$ as an edge from x to y, then the problem of finding an element which is at most 2δ smaller than the maximum element, transforms into the problem of finding a vertex in a tournament, which can reach the rest of the vertices by a directed path of length at most two.

There are two different papers written several years apart with almost identical upper bound and lower bound strategies for finding a king [39, 40]. Here basically one counts the number of probes (or edge-queries) to the adjacency matrix of the tournament. Both these papers show an upper bound of $O(n\sqrt{n})$ time and a lower bound of $\Omega\left(n^{4/3}\right)$ time to find a king in a tournament. Closing the gap between the upper and lower bound has been mentioned as an interesting open problem. We address this problem in the chapter. While we don't quite close the gap, we make several advances.

Our first result is that to improve the lower bound, we need a different adversary than used in these papers. The lower bound adversary essentially answers an edge query by favoring the vertex (i.e. increasing its outdegree) with lower outdegree till that time. We give an $O\left(n^{4/3}\right)$ algorithm against this adversary. As the adversary can answer arbitrarily when the outdegree between the two vertices are the same, we need to play against this adversary carefully to prove our upper bound.

Ajtai et al. [39] generalized the notion of a king to a d-king (for any $d \geq 2$). A d-king is a vertex from which every vertex is reachable by a path of length at most d. Ajtai et.al showed that $\Omega\left(n^{1+1/(2^d-1)}\right)$ probes are necessary and $O\left(k(n/k)^{1+1/(3(2^{d-2}-1)}\right)$ are sufficient to find a d-king in an n vertex tournament. Our algorithm against the adversary for finding a 2-king also works to give an optimal algorithm against the same adversary for finding a d-king. More generally we show that if a 2-king can be found in $O\left(n^{4/3}\right)$ time, then a d-king can be found in optimal $O\left(n^{1+1/(2^d-1)}\right)$ time.

Then we address the complexity of finding not just one king, but a small subset of vertices such that every other vertex is reachable by a path of length at most 2 from one of these vertices. More generally, a d-cover is a set of vertices S such that every other vertex in the tournament can be reached from some vertex in S by a directed path of length at most d. A 1-cover is simply a *dominating set*, and d-covers are referred to as *distance-d* dominating sets.

It is well-known that every tournament has a dominating set of size at most $\lceil \lg n \rceil$ and can be found in $O(n^2)$ time. In fact, one can guarantee existence of a slightly smaller dominating set that can also be found in the same time. There exists a dominating set of size $g(n) = \lg n - \lg \lg n + 2$ in a tournament on n vertices [41] (see also Section 5.2.2). Thus, the minimum dominating set can be found in $n^{O(\lg n)}$ time and hence it is unlikely to be NP-complete. Papadimitriou and Yannakakis [42] show that finding the smallest dominating set in a tournament is complete for a complexity class $LOGSNP$. It is also known that there are tournaments where the minimum dominating set size (also called the *domination number*) is $\Omega(\lg n)$ [43], [44] (see also [45]) using which it has been shown that finding a k-sized dominating set is complete for the parameterized complexity class $W[2]$ (where the parameter is the solution size) and hence, even an $f(k)n^{O(1)}$ time algorithm is unlikely for any computable function f [46].

We consider the complexity of finding dominating sets of size more than $\lg n - \lg \lg n + 2$ and more generally, the complexity of finding d-covers. Using the lower bound adversary that favors the lower degree vertex, we can show a lower bound for finding k-sized d-covers. We also provide a matching upper bound for for finding such d-covers for $k \in \Omega(\lg n)$. More specifically, we show that $\lg n - \lg \lg n + 2 + k$ sized d-covers can be found in $O\Big(k(n/k)^{1+1/(2^d-1)}\Big)$ time for any $d \geq 1$ and $k \geq 1$. We also prove a matching adversary based lower bound to show that this bound is optimal for $k > \epsilon \lg n$ for any $\epsilon > 0$. This, in particular, implies that a dominating set of size $k > \lg n$ can be found in optimal $O(n^2/k)$ time.

We also discuss the complexity of finding a special king vertex called a Banks vertex which has applications in social choice theory [47]. A Banks vertex is a source of a transitive subtournament of vertices, who dominate the entire tournament. I.e. let S be a subset of vertices of the tournament such that the induced tournament on S is acyclic, and S is a dominating set of the tournament. Clearly

the source vertex of the induced subtournament on S is a king, and that is called a Banks vertex. We observe in this chapter that the known $O(n\sqrt{n})$ algorithm to find a king, actually finds a Banks vertex.

In social choice theory, the set of all kings in a tournament is referred to as an *uncovered set.* It is known that finding the set of all kings or Banks points in a tournament requires $\Omega(n^2)$ probes [48]. A similar, but a slightly different kind of result has been known for finding a sorted sequence of kings. A sorted sequence of kings is in a tournament on n vertices is a sequence of vertices $v_1, v_2 \ldots v_n$, such that v_i dominates v_{i+1} and v_i is a king in sub-tournament $T_{v_i} = \{v_i, v_{i+1} \ldots v_{v_n}\}$. It has been known [40] that to find a sorted sequence of kings, $\Theta(n\sqrt{n})$ probes are necessary and sufficient.

Finally we discuss the problem of finding a king or a Banks vertex in an incremental setting. I.e. given a tournament we can maintain some invariants of the tournament so that when a new vertex is added to it, we can find a king or a Banks vertex in $O(\sqrt{n})$ time.

Organization of the chapter In Section 5.2, we describe the necessary terminology and some basic results on tournaments that we use in the chapter. In Section 5.3, we describe our optimal algorithm against the known lower bound adversary . In Section 5.4, we look at d-covers and describe several upper and lower bounds for finding d-kings and d-covers in a tournament. Section 5.5 shows an adversarial strategy which forces $\Omega(n^2)$ time to verify whether a vertex is a king. In Section 5.6, we give an incremental (vertices can be added to the tournament, but not deleted) dynamic algorithm to find a king or a Banks point in tournament. We conclude with pointers to some open problems in Section 5.7.

5.2 Preliminaries

5.2.1 Definitions and Notation

Let $T = (V, E)$ be a tournament (complete directed graph). A *dominator* in T is a vertex $u \in V$ such that for any other vertex $v \in S$, $(u, v) \in E$. A *king* in T is a vertex $u \in V$ such that for any other vertex $v \in V$ there is a directed path of length at most 2 from u to v. We denote by $V(T)$ the vertex set of T, and by

$E(T)$, the edge set of T. $|T|$ denotes the order of T, i.e. the size of $V(T)$, and for a subset $S \subseteq V(T)$, $T[S]$ denotes the subtournament of T induced by S. A tournament or a subtournament is transitive if it has no directed cycle; i.e. there is an ordering of the vertices such that every edge is going from a smaller vertex to a larger vertex.

Let $d \in \mathbb{N}$ and $S \subseteq V$. S is called a *d-cover* of T if for every vertex $v \in V \setminus S$, there is vertex $u \in S$ such that there is a directed path of length at most d from u to v.

A 1-cover is also called a *dominating set*. A king by itself is a 2-cover. If a d-cover is of size 1, we call the unique element in the set a d-king. Let $u \in V$ be a vertex. An out-neighbor of u is a vertex $v \in V$ such that $(u, v) \in E$; u is said to *dominate* v. Similarly, an in-neighbor of u is a vertex $v \in V$ such that $(v, u) \in E$. For any $v \in V(T)$, $N^+(v)$ denotes the set of out-neighbors of v, and $N^-(v)$ denotes the set of its in-neighbors. We also define $\deg^+(v) = |N^+(v)|$, and $\deg^-(v) = |N^-(v)|$.

Given a subset S of vertices, *performing a round robin tournament* on S refers to the process of querying all possibles edges between every pair of vertices in S to know their orientations. Using the same tournament analogy, we sometimes refer to the out-neighbors of a vertex v as vertices that *lose* to v, and to its in-neighbors as vertices that *win* against it. In other words, if there is an edge from u to v, we say that u wins against v, and v loses to u.

We assume that the vertex set of the tournament is $\{1, 2, \ldots n\}$ and is given in the form of an adjacency matrix where the (i, j)-th entry denotes the direction of the edge between i and j. By an edge query or an edge probe, we mean a probe to the adjacency matrix.

To keep the notation simple, we sometimes omit ceilings and floors on fractions when we actually mean integers, typically on the sizes of the vertex subsets we handle. This does not affect the asymptotic analysis.

5.2.2 Some Known Results Which We Have Used

In this section, we capture some elementary results about the nature and the complexity of finding kings, that are used later in the chapter.

Lemma 5.2.1 ([49]). *Any tournament has a king vertex (a vertex from which every other vertex is reachable by a path of length at most 2). In particular, a maximum degree vertex of the tournament is a king.*

Lemma 5.2.2 ([49]). *Let T be a tournament and $v \in V(T)$. If a vertex u in $N^-(v)$ is a king in $T[N^-(v)]$, then u is a king in T.*

Proof. The vertex u reaches every vertex in $N^+(v)$ through v. Every other vertex in T is reachable from u by a directed path of length at most 2 as it is a king in $T[N^-(v)]$.

□

The above lemma can be used to give a simple algorithm to find a king.

A simple algorithm Pick any vertex $v \in V$ and find all its out-neighbors in V. Remove v, as well as all its out-neighbors in V from V. Repeat this process till V becomes empty, in which case the last picked vertex is a king. The algorithm takes $O(n^2)$ time.

Lemma 5.2.3. *For a tournament of order n a king can be found in $O(n^2)$ time.*

Shen et al. [40] (see also [39]) gave a better algorithm taking $O(n\sqrt{n})$ time which uses the following simple lemma. The lemma follows from the fact that in a tournament, the sum of the outdegrees is $n(n-1)/2$.

Lemma 5.2.4 ([41]). *In any tournament T of order n, there is a vertex with outdegree at least $(n-1)/2$ and a vertex with indegree at least $(n-1)/2$ and such vertices can be found in $O(n^2)$ time.*

Lemma 5.2.5 ([39, 40]). *For a tournament of order n, a king can be found in $O(n\sqrt{n})$ time.*

Proof. The algorithm first picks an arbitrary set of vertices of size $\sqrt{n}$ and performs a round-robin tournament on it. Then, it picks a vertex (say v) of maximum outdegree (which is at least $\sqrt{n}/2$ by Lemma 5.2.4) in the set, and finds its out-neighbors in the entire tournament. Next, it removes v and all its outneighbors from V, and repeats the process until V has less than $\sqrt{n}$ vertices in it. At this

point, the algorithm uses the routine of Lemma 5.2.3 and finds a king in $O(n)$ time. It is easy to see that the king in the final set is a king for the entire tournament by Lemma 5.2.2 and that the algorithm takes $O(n\sqrt{n})$ time.

□

Banks Point

A Banks point is a king vertex with some special properties and has been studied extensively in social choice theory, where they are useful in determining winners in certain types of competition [47].

Definition 5.2.1 (Banks Point, Banks Set [47])**.** Let $T = (V, E)$ be a tournament. A vertex $v \in V$ is called a *Banks* point if there is a sequence of vertices $\{v_1, v_2, \ldots, v_k = v\}$ such that

1. v_i dominates v_j, for all $1 \leq j < i \leq k$, and
2. $\{v_1, \ldots, v_k\}$ is a dominating set for G.

The Banks set for T is the set of all Banks points in T.

From the above definition, it is clear that $S = \{v_1, \ldots, v_k\}$ induces a transitive subtournament of T. Also, $v_k = v$ dominates all other vertices in S. Since S is a dominating set, all vertices in V are reachable from v_k by a directed path of length at most 2. Thus, v_k is a king. However, not every king is a Banks point [50].

Note that the algorithm for finding a king in Lemma 5.2.3 actually finds a Banks vertex as the last vertex dominates all the vertices picked up in the previous iterations and together will all of them, it dominates the entire tournament. It is also not hard to see that Lemma 5.2.5 also actually finds a Banks point as the sequence of vertices found in the phases of elimination form a transitive subtournament. These vertices which form the transitive tournament are of size $O(\sqrt{n})$. Then in the last subtournament (which is also of size $O(\sqrt{n})$), where every vertex dominates the vertices picked earlier, we apply Lemma 5.2.3 to find a king vertex, which actually finds a Banks vertex, which gives us the following lemma.

Lemma 5.2.6. *For a tournament of order n, a Banks vertex can be found in $O(n\sqrt{n})$ time, and the number of vertices forming the transitive tournament is $O(\sqrt{n})$.*

Dominating Sets

The following result is immediate from Lemma 5.2.4. We simply find a vertex in the tournament that dominates at least $(n-1)/2$ other vertices, include it in the dominating set and recurse on the in-neighbors of the vertex.

Theorem 5.2.7 ([45])**.** *For any tournament of order n, a dominating set of size at most $\lceil \lg n \rceil$ can be found in time $O(n^2)$.*

There is also a tighter upper bound which is probably not well-known [41], we give the proof here for completeness.

Theorem 5.2.8. *In any tournament $T = (V, E)$ of order n, a dominating set of size at most $\lg n - \lg\lg n + 2$ exists, and can be found in time $O(n^2)$.*

Proof. We proceed as in the case of Theorem 5.2.7, but bail out of the recursion after $\lceil \lg n - \lg\lg n \rceil + 1$ steps. At this point, the partial dominating set D found has at most $\lceil \lg n - \lg\lg n \rceil + 1$ vertices, and the remaining subtournament R (whose vertices are yet to be dominated) has at most $n/2^{\lg n - \lg\lg n + 1} < (\lg n)/2$ vertices. By definition, R dominates all of D.

Now, by looking at the out-neighborhood of $V(R)$ in $V \setminus D$ we check (in $O(n^2)$ time) whether $V(R)$ dominates all of $V \setminus D$. If so, we output $V(R)$, whose size is at most $(\lg n)/2$. Otherwise, there is a vertex x in $V \setminus \{D \cup V(R)\}$ that dominates the vertices of R. In this case, we output $D \cup \{x\}$ as the dominating set. Either way, the size of the dominating set output is at most $\lg n - \lg\lg n + 2$.

□

Central to the lower bound result in [40] and [39] for finding a king, is, the adversary that answers according to the pro-low strategy defined below.

Definition 5.2.2 (Pro-Low Strategy)**.** Let T be a tournament whose edge directions are determined by the following adversary strategy for an algorithm that queries edges: when the edge uv is queried, it is assigned the direction $u \to v$ if $\deg^+(u) \leq \deg^+(v)$, and $u \leftarrow v$ otherwise. Here $\deg^+(u)$ and $\deg^+(v)$ denote the outdegree of u and v before the edge query is made.

The following lemma was used in the lower bound arguments.

Lemma 5.2.9. *[40, 39] Let T be a tournament and $S \subseteq V(T)$. Suppose an adversary answers edge queries involving vertices $u, v \in S$ using the pro-low strategy, i.e., it compares out-degrees within $T[S]$. If a vertex v in S achieves $\deg_S^+(v) = t$, then at least $t(t+1)/2$ edge queries must have been made by the algorithm.*

5.3 Finding a King Against a Pro-Low Adversary

As mentioned in the introduction, it is known that $O(n^{3/2})$ probes are sufficient and $\Omega(n^{4/3})$ probes are necessary to find a king in a tournament on n vertices. Narrowing this gap between upper and lower bound has been mentioned as an open problem in literature [39, 40]. In this subsection, we take a look at the adversary strategy used in [39, 40] and show that there exists an algorithm which works specifically against their strategy and can find a king using $O(n^{4/3})$ edge queries, matching the lower bound given by their adversary.

Theorem 5.3.1. *If the adversary follows a pro-low strategy as described in Definition 5.2.2 then we can find a king in $O(n^{4/3})$ time.*

Proof. Take a sample of $2n^{1/3}$ vertices from the input V and find a vertex v that dominates at least half of these vertices (such a vertex exists from Lemma 5.2.4) using a round-robin tournament on the sample. Remove v and all vertices that lose to v (in that sample) from V, and add v to a set D. Repeat this process of sampling $2n^{1/3}$ elements and discarding elements, and adding elements to D, till there are at least $2n^{2/3}+1$ and at most $3n^{2/3}$ elements remaining which we call R. The number of edge queries made up to this point is $O(n^{4/3})$.

Notice that the vertices in D form a dominating set for all vertices in the input, except those in R. Let $|D| = f$ and $|R| = r$. Clearly $f < n^{2/3} < r/2$. Arrange the vertices of D in non-decreasing order of their out-degrees. Let $v_1, v_2, \ldots v_f$ be the vertices of D in that order. We shall probe edges between all vertices in D and all vertices in R in a certain order which shall be described below. This step also uses $O(n^{4/3})$ queries, since D and R are both of size $O(n^{2/3})$. The aim is to demonstrate a vertex in R that wins against all vertices in D. Once we find such a vertex, we will be done as the following lemma shows.

Lemma 5.3.2. *Let C be a non-empty set of vertices in R that dominate all vertices in D. Then a king of $T[C]$, the induced subtournament on C, is a king for the entire tournament.*

Proof. Let x be a king of $T[C]$. Clearly x can reach all vertices of C in at most two steps. As x dominates every vertex of D, x can reach every vertex of $V \setminus R$ in at most two steps. Now consider a vertex v in $R \setminus C$. As $v \notin C$, v loses to some vertex y of D. So x can reach v by a path of length 2 by passing through y. □ □

So the rest of the algorithm probes edges between R and D in such a way that the pro-low adversary is made to give a vertex in R that dominates all vertices in D.

Recall that $v_1, v_2, v_3 \ldots v_f$ is the set of vertices in D in non-decreasing order of their degrees. As we probe edges between vertices in D and those of R, some of the degrees of D will change, and if that happens, we rearrange the vertices so that the vertices continue to be in non-decreasing order of their degrees.

We define a *free win point* in D as the maximum integer value p such that for all $i \leq p$, $deg(v_i) \geq i$. By this definition, we can deduce the following.

Lemma 5.3.3. *If p is the free win point, then $deg(v_{p+1}) = p$.*

Proof. As the vertices in D are in non-decreasing order of their degrees, $deg(v_{p+1}) \geq deg(v_p) \geq p$ which implies that $deg(v_{p+1}) \geq p$. But $p+1$ is not a free win point and so $deg(v_{p+1}) < p+1$ from which it follows that $deg(v_{p+1}) = p$.

□

The following claim gives the importance of the notion of free win point.

Lemma 5.3.4. *If p is the free win point, then any vertex v in R will, by default, win at least up to the vertex v_p (if queried in that order) when played against a pro-low adversary.*

Proof. When $i = 1$, it is true, since all vertices in D have outdegree at least $n^{1/3} \geq 1$ and v has 0 wins before querying with v_1. When edge $(v, v_i), i < p$ is probed v has exactly $i-1$ wins since it has won all the $i-1$ probes up to this point, and $deg(v_i) \geq i$. Hence, $\forall i \leq p$, v will dominate v_i. □

The following lemma is immediate from the lemma above.

Lemma 5.3.5. *If the free win point p is $|D|$, then all vertices in R which have not queried their edges with any vertices in D will win all vertices of D whenever the queries are made.*

We probe every vertex of D with every vertex in R. After completing with one vertex of R, we simply choose another vertex of R arbitrarily. However, we probe that vertex with vertices in D in a careful order. Now we explain the order in D in which edges are probed between a vertex v in R. We first probe v up to the free win point p, thus making v win against all these vertices and obtain a degree of p. The queries after this point, depend on which of the following cases happen.

- There exists a vertex v' in D such that $degree(v') > degree(v_{p+1}) = p$.

  In this case, we play v with v' to get degree $p+1$ (as degree of v is p and degree of v' is more than p). Then v plays with all vertices with degree p and v would lose to all of them while all of them including v_{p+1} will attain a degree of $p+1$. Then v plays with all the remaining vertices in D.

- The vertex v_{p+1} is a maximum degree vertex in D. In this case, we simply query v' with v_{p+1} and all other vertices with the same degree. Note that as $degree(v)$ would also be p when the first (and possibly even subsequent) queries are made, the adversary may answer either way as the adversary answers arbitrarily in case of ties in the degrees.

Now we show the following about the sequence in D by which we probe vertices of R.

Lemma 5.3.6. *Let $x_1, x_2, \ldots x_r$ be the vertices in R in the order in which they are probed against vertices in D. If the free win point is p before x_i has started probing with vertices in D, then by the time x_{i+1} has completed its round, the free win point will be at least $p+1$ or we would have found a vertex that dominates all vertices of D.*

Proof. The vertex x_i will win with all vertices up to $v_p \in R$ by Lemma 5.3.4. If there is no vertex beyond v_p, then x_i is the vertex that dominates all of D.

Otherwise, we argue based on various cases. If there is a vertex whose degree is more than $deg(v_{p+1}) = p$, then the algorithm probes with such a higher degree

vertex, wins (due to the pro-low adversary) and then plays with all those with degree p and loses to them (as it has a higher degree when it started playing with them). So all vertices with degree p including v_{p+1} gains a degree and hence the free win point moves to at least $p+1$ as $deg(v_{p+1}) = p+1 \geq p+1$ now.

If v_{p+1} is the only maximum degree vertex, then one of two cases arises as the adversary can answer arbitrarily as there is a tie in the degree of x_i and v_{p+1}. If x_i wins, then it is a vertex that dominates all vertices of D, and otherwise v_{p+1} gains an extra degree moving the free win point.

So the only case we are left to argue is that $degree(v_{p+1})$ has the maximum degree but there is a set M of $m > 1$ vertices with the maximum degree. Here also the adversary can answer arbitrarily due to the tie between the degree of x_i and of the m vertices. If x_i loses to all of them, then all vertices with degree p in D now have degree $p+1$. Thus $degree(v_{p+1})$ changes to $p+1$ and thus the free win point also changes to $p+1$. Otherwise, observe that if x_i wins any one of the m vertices, then it gains a higher degree and hence will lose to subsequent vertices of M due to the pro-low strategy. Hence x_i can win at most one vertex of M, and since we have assumed that x_i loses to at least one of the m vertices, it will lose to exactly one. After x_i completes its round, one of the m vertices of M will gain degree $p+1$ and all others remain with degree p. Now as there is a vertex with degree more than p, due to the cases argued above, x_{i+1} from R will play with such a higher degree vertex before playing vertices with degree p, and hence those with degree p including v_{p+1} will gain a degree and the free win point will advance to at least $p+1$.

□

The following claim follows from Lemma 5.3.5 and 5.3.6 and the fact that $|R| > 2|D|$ and every vertex in D starts off with outdegree at least $n^{1/3}$ and every vertex in R starts with outdegree 0.

Lemma 5.3.7. *There exists a vertex in R that wins against all vertices of D.*

Now the algorithm simply probes all edges between verties in R and vertices in D, identifies the set of vertices that dominate all of D (and the set is non-empty by Lemma 5.3.7) and returns a king among them. The fact that the returned vertex

is the king for the entire tournament follows from Lemma 5.3.2. The total number of edge queries made during the entire course of the algorithm is clearly $O\left(n^{4/3}\right)$.

This concludes the proof of Theorem 5.3.1 □

5.4 Finding d-Kings and d-Covers

As described before a set $S \subseteq V$ is called a *d-cover* of T if for every vertex $v \in V \setminus S$, there is vertex $u \in S$ such that there is a directed path of length at most d from u to v.

While we showed in the last section that the pro-low adversary is weak to provide an improved lower bound for finding a king, here we show that it is still useful to find a lower bound for finding a set of vertices such that there is a directed path of length at most 2 (or more generally d) to every other vertex from one of these vertices. We complement the lower buond by providing an algorithm that achieves this bound for finding $\Omega(\lg n)$ sized d-cover.

5.4.1 Lower Bounds

Generalizing Lemma 5.2.9, Ajtai et al [39] show that

Lemma 5.4.1 ([39])**.** *When playing against a pro-low adversary, if an algorithm finds a vertex that reaches at least t vertices by a path of length at most d for $d \geq 1$, then at least $\Omega\left(t^{1+1/(2^d-1)}\right)$ edge queries must have been made.*

Setting $t = n$ gives the lower bound for finding a d-king. We generalize the above lemma to show

Lemma 5.4.2. *If an algorithm playing against the pro-low adversary finds a k-sized d-cover for some $k, d \geq 1$, then it must have made at least $k(n/k)^{1+1/(2^d-1)}$ edge queries.*

Proof. Suppose the algorithm returns a set of vertices $C = \{v_1, v_2, v_3 \ldots, v_k\}$ as the k sized d-cover of the entire tournament. Perform a breadth-first search from the vertices of C with all vertices in C at level 0, those that have direct edges from them at level 1 etc. The breadth-first search tree has d levels. Now, for each vertex in level i (from d to 1) assign a unique vertex at level $i-1$ from which

there is a direct edge to it. This partitions the vertices into k sets, where the i-th set contains the vertex v_i, and all vertices assigned to it from level 1, and all vertices assigned to those level 1 vertices from level 2 and so on. Thus we have k subtournaments T_1, T_2, $\ldots T_k$ induced on the partition, and let s_i be the number of vertices in the sub-tournament T_i. Note that v_i has an at most d-length path to every vertex in T_i and hence by Lemma 5.4.1, the number of edge queries made in the tournament T_i is at least $(s_i)^{1+1/(2^d-1)}$. Thus the total number of edge queries done by all the vertices is at least $\sum_{i=1}^{k}[(s_i)^{1+1/(2^d-1)}]$. As $\sum_{i=1}^{k} s_i = n$, we have that the number of edge queries made is at least $k(n/k)^{1+1/(2^d-1)}$ using Jensen's inequalities (as $\sum_{i=1}^{k}[(s_i)^{1+1/(2^d-1)}]$ attains its minimum when all the s_i's are the same as their average value which is n/k).

□

For $d = 1$, we give a slightly different argument which maybe of independent interest.

Theorem 5.4.3. *Let $1 \leq k \leq n$. If an algorithm finds a dominating set of size at most k in a tournament of order n, then it makes $\Omega(n^2/k)$ edge queries to its input in the worst case.*

Proof. We prove the result by an adversary argument. The adversary dynamically builds the input instance $T = (V, E)$, a tournament of order n. At all times, it maintains a partition $V = S \cup S'$ of the vertex set.

Initially $S = V$, $S' = \emptyset$ and $\deg^+(v) = 0$ for all $v \in V$. Consider the following invariant:

- For each $v \in V$, $\deg^+_S(v) < (n/3k) - 1$.

Suppose the above invariant holds.

Lemma 5.4.4. *If $|S| \geq n/2$ and $D \subseteq V$ is a dominating set of T, then $|D| > k$.*

Let $v \in D$. Then, by the invariant, v dominates at most $n/3k$ vertices in S (as a vertex in S also dominates itself). Since $|S| \geq n/2$, we have

$$|D| \geq \frac{n/2}{n/3k} > 3k/2 > k.$$

This proves the claim.

The adversary uses the following strategy. When a new edge uv is queried, if $u, v \in S$, the adversary uses the pro-low strategy and directs the edge from u to v if $\deg_S^+(u) \leq \deg_S^+(v)$, and from v to u otherwise. If $u, v \in S'$, the adversary arbitrarily directs the edge. If $u \in S$ (resp. $v \in S$) and $v \in S'$ (resp. $u \in S'$), then the adversary directs the edge from u to v (respectively from v to u).

After a query is answered, if a vertex $v \in S$ satisfies $\deg_S^+(v) = \lceil n/3k \rceil - 1$, v is moved with all its out-neighbors into S'. This ensures that the invariants always hold. It now suffices to show that to reach a stage where $|S| < n/2$, $\Omega(n^2/k)$ queries must be made. Note that $|S|$ shrinks in steps of $\lceil n/3k \rceil$, i.e., its size only halves after $\mathrm{O}(3k)$ steps. At each step, a vertex v with $\deg_S^+(v) = \lceil n/3k \rceil - 1$ is moved into S'. For a vertex in S to acquire a degree (in S) of $\lceil n/3k \rceil$, $\Omega\big((n/3k)^2\big)$ queries must be made (by Lemma 5.2.9 as we use pro-low strategy inside S). So the number of queries needed to halve the size of S is $\Omega\big(3k(n/3k)^2\big) = \Omega(n^2/k)$.

Therefore in the worst case, any algorithm that finds dominating sets of size at most k makes $\Omega(n^2/k)$ edge queries.

□

Corollary 5.4.5. *If an algorithm finds dominating sets of size* $\mathrm{O}(\log n)$ *in tournaments of order* n*, then it makes* $\Omega(n^2/\log n)$ *edge queries to its input in the worst case.*

Theorem 5.4.6. *Any algorithm that finds minimum dominating sets in tournaments makes* $\Omega(n^2/\log n)$ *edge queries to its input in the worst case.*

Proof. Since a minimum dominating set has the smallest size among all dominating sets, any algorithm that finds minimum dominating sets in tournaments finds dominating sets of size atmost $\log n$, because of Lemma 5.2.7. By Corollary 5.4.5, this implies that the algorithm makes $\Omega(n^2/\log n)$ edge queries in the worst case.

□

5.4.2 Upper Bounds

In this subsection we provide algorithms to find d-covers that match the lower bounds proved in the last section, as long as the size of the d-covers is $\Omega(\lg n)$.

Finding a dominating set (1-cover) of size $(k + \lg n - \lg\lg n + 2)$

We start with the following result for 1-cover (dominating set).

Theorem 5.4.7. *A* $(k + \lg n - \lg\lg n + 2)$ *sized 1-cover in a tournament on* n *vertices can be found in* $O(n^2/k)$ *time.*

Proof. Let V be the input set of vertices. If $k \leq 2$ apply Theorem 5.2.8. Otherwise, pick a subset of vertices of size $2(n/k)+1$ and find a vertex (say u) which dominates at least (n/k) vertices inside this subset using Lemma 5.2.4. This vertex can be found using $O(n^2/k^2)$ time. Delete u's out-neighbors add u to the dominating set. Now recurse on the remaining tournament as long as $|V| \geq 2n/k + 1$. Once $|V| \leq 2n/k$, apply Theorem 5.2.8 and find a 1-cover of size $\lg n - \lg\lg n + 2$ for the remaining tournament in $O((n/k)^2)$ time, and add these vertices to the output cover. The output cover is a 1-cover for the tournament, of size at most $k+\lg n-\lg\lg n+2$. The total time taken is $O\big((k-2)(n/k)^2 + (n/k)^2\big) = O(n^2/k)$.

□

Finding d**-covers** $(d \geq 2)$ **of size** $(k + \lg n - \lg\lg n + 2)$

For $d \geq 2$, the lower bound on the size of the d-cover is 1. We first make the following simple generalization of the $O\Big(n^{1+1/(3*2^{d-2}-1)}\Big)$ algorithm [39] to find a d-king to show the following.

Theorem 5.4.8. *In any tournament of order* n*, a* d*-cover of size* k *can be found in* $O\Big(k(n/k)^{1+1/(3(2^{d-2}-1)}\Big)$ *time.*

Proof. Partition the vertex set into k parts to get k sub-tournaments on roughly n/k vertices each. Now use the d-king finding algorithm in [39] in each sub-tournament to find a d-king. The set of all these d-kings together form a k-sized d-cover. Time taken to find a d-king in n/k size tournament is $O\Big((n/k)^{1+1/(3*2^{d-2}-1)}\Big)$, thus the time taken to find a d-cover of size k is $O\Big(k(n/k)^{1+1/(3*2^{d-2}-1)}\Big)$.

□

In what follows, we give an improved algorithm for finding d-covers of size more than $\lceil \lg n \rceil$. We start with a lemma similar to Lemma 5.2.4 for 2-covers.

Lemma 5.4.9. *In any tournament of order n, a vertex that reaches at least $(n-1)/2$ other vertices via directed paths of length at most 2 can be found in time $\mathrm{O}\big(n^{4/3}\big)$.*

Proof. Let V be the vertex set of the tournament. Pick a subset of vertices of size $2n^{1/3}+1$ and find a vertex (say u) which dominates at least $n^{1/3}$ of those vertices. Take exactly $n^{1/3}$ of u's losers and remove them from V. Also add u to a set of vertices called C. Repeat this process on the resulting V as long as $|V| \geq 2n^{1/3}+1$. Once $|V| < 2n^{1/3}+1$, perform a round-robin tournament on all the vertices in C and find a vertex k that dominates at least $(|C|-1)/2$ vertices using Lemma 5.2.4. The size of C is at least $n^{2/3} - n^{1/3} - 1$.

Thus vertex k can reach at least $n^{1/3}(n^{2/3} - n^{1/3} - 1)/2 = n/2 - (n^{2/3} - n^{1/3})/2$ vertices with path of length exactly two, and can reach $(n^{2/3} - n^{1/3} - 1)/2$ vertices with path of length 1. Thus the vertex k is king of at least $(n-1)/2$ vertices.

The total number of edge queries made is $\mathrm{O}\big(n^{4/3}\big)$, since round robin on set C costs $\mathrm{O}\big(n^{4/3}\big)$ edge queries, as well as the time taken to create the set C is $\mathrm{O}\big(n^{2/3}(n^{1/3})^2\big) = \mathrm{O}\big(n^{4/3}\big)$.

□

It is interesting to note that Yao [51] conjectured that finding a partial order S_k^m (an element which has k elements less than itself and m elements greater than itself) in a total order of size n cannot be done faster even if extra elements are given. Lemma 5.4.9 shows that finding a king of n elements becomes substantially easier if $2n$ elements are given. As we don't see any connection at the problem level, we are not sure whether our algorithm provides any insight to address Yao's conjecture.

Corollary 5.4.10. *A $(\lg n - \lg\lg n + 2)$ sized 2-cover in a tournament on n vertices can be found in $\mathrm{O}\big(n^{4/3}\big)$ time.*

Proof. Use Lemma 5.4.9 to find a vertex $v \in V$ which has a path of length at most 2, to at least $(n-1)/2$ vertices (say to the set C). Remove v and the vertices in C from V after adding v to the 2-cover set S and adding C to S', and repeat this

process on the remaining vertices, till the number of vertices in S is $\lg n - \lg\lg n + 1$. At this point the number of vertices remaining in V is at most $(\lg n)/2$. Find the relation of all vertices with these $(\lg n)/2$ with the vertices in S' using $O(n \lg n)$ queries. One of the two cases happen. Either some vertex v' in S' dominates all vertices in V, in which case we add v' to S and output it as the 2-cover, or no vertex can dominate all vertices in V, in which case V forms a dominating set of size $(\lg n)/2$ and we output as 2-cover, since any 1-cover is a valid 2-cover. This would give a $\lg n - \lg\lg n + 2$ sized 2-cover.

□

Theorem 5.4.11. *A $(k + \lg n - \lg\lg n + 2)$ sized 2-cover in a tournament on n vertices can be found in $O\Big(k(n/k)^{4/3}\Big)$ time.*

Proof. Take a subset of input vertices of size $2n/k$ from V. Using Lemma 5.4.9 find a 2-king u of at least half of these vertices using $O\big((n/k)^{4/3}\big)$ edge queries. Remove u from V and add it to the output set C, and remove all vertices in V that can be reached by a path of length at most 2 from u. Recurse on the remaining tournament as long as $|V| \geq 2n/k$. When $|V| < 2n/k$, find a $\lg n - \lg\lg n + 2$ sized 2-cover, by using $O\big((n/k)^{4/3}\big)$ edge queries using Corollary 5.4.10, and add this 2-cover to the set C. The set C is a 2-cover for all the vertices in the tournament of size at most $k + \lg n - \lg\lg n + 2$. The total time spent is $O\big(k(n/k)^{4/3} + (n/k)^{4/3}\big)$ $= O\big(k(n/k)^{4/3}\big)$.

□

Now we generalize the above theorem for $d \geq 2$.

Lemma 5.4.12. *Let $d \geq 2$ and $k \geq 1$ be an integer. If a k-sized 2-cover can be found in $O\Big(k(n/k)^{4/3}\Big)$ time, then a k-sized d-cover can be found in $O\Big(k(n/k)^{1+1/(2^d-1)}\Big)$ time.*

Proof. We prove this by induction on d. For $d = 2$, there is nothing to prove. Let $d \geq 3$. Assume that the lemma is true for all integers from 2 to $d - 1$.

Let V be the set of vertices and $s = (n/k)^{1/(2^d-1)}$. Since $d \geq 3$, $n/s = n^{1-1/(2^d-1)}k^{1/(2^d-1)} > 3 + \lg n$ (for large enough n), and so using Theorem 5.4.7 find a (n/s)-sized 1-cover of V using $O(n^2/(n/s)) = O(ns)$ time. Now find a k-sized $(d-1)$-cover for these n/s vertices using $O\Big(k(n/ks)^{1+1/(2^{d-1}-1)}\Big) = O(ns)$

time using induction hypothesis. The resulting set is a k-sized d-cover of entire input, which is found in $O(ns) = O\Big(k(n/k)^{1+1/(2^d-1)}\Big)$ time.

□

Thus we have from Theorem 5.4.11, Lemma 5.4.12 and Theorem 5.4.7,

Theorem 5.4.13. *A $(k + \lg n - \lg\lg n + 2)$-sized d-cover in a tournament on n vertices can be found in $O\Big(k(n/k)^{1+1/(2^d-1)}\Big)$ time for any $k \geq 1$ and $d \geq 1$.*

By setting $k = 1$ in Lemma 5.4.12, we have

Corollary 5.4.14. *If a 2-king can be found in $O\big(n^{4/3}\big)$ time, then we can find a d-king in $O\Big(n^{1+1/(2^d-1)}\Big)$ time.*

The above corollary was mentioned as a likely possibility (without proof) in the conclusion of [39].

The following corollary follows from Lemma 5.4.12 and Theorem 5.3.1.

Corollary 5.4.15. *If the adversary follows a pro-low strategy as described in Definition 5.2.2 then we can find a d-king in $O\Big(n^{1+1/(2^d-1)}\Big)$ time.*

The above lemma implies that we need a completely different adversary strategy to improve the lower bounds for finding d-kings.

5.5 Verification of Kings

In this section, we deal with problem of verifying whether a given vertex is a king or not. One can easily verify in $O(n^2)$ time whether a given vertex is a king by running a standard breadth-first-search starting at the vertex and seeig whether every vertex is reachable by a path of length at most 2. We give an $\Omega(n^2)$ lower bound for this problem. It is interesting to note that it is NP-complete [52] to verify whether a given vertex is a Banks point.

Lemma 5.5.1. *Let T be a tournament of order n. Given a vertex $v \in V(T)$, it takes $\Omega(n^2)$ edge queries to decide whether v is a king in T.*

Proof. Consider the following adversary strategy. The adversary arbitrarily fixes subsets $A, B \subseteq V(T) \setminus \{v\}$ such that $|A| = \lfloor (n-1)/2 \rfloor$ and $|B| = \lceil (n-1)/2 \rceil$, and then assigns edge directions such that $N^+(v) = B$ and $N^-(v) = A$. The remaining edge directions are assigned dynamically.

Suppose $u \in A, w \in B$, and the edge uw is queried. The following cases arise.

- There are other non-queried u–B edges: the adversary assigns the direction $u \to w$.
- All other u–B edges have been queried: the adversary assigns the direction $w \to u$.

For edges uw where $u \in A$ and $w \in A$ or $u \in B$ and $w \in B$, the adversary answers arbitrarily. Note that v directly reaches each vertex in $N^+(v)$ by a directed edge, and it can only reach vertices in $N^-(v)$ through vertices in $N^+(v)$. For any vertex $u \in A$, an algorithm cannot decide whether v can reach u until it queries all u–B edges, since the status of v depends on how the adversary answers when the last u–B edge query is made. By the same token, the algorithm cannot decide whether v can reach all vertices of A unless it asks all edges between A and B.

Thus for every $u \in A$, $\lceil (n-1)/2 \rceil$ queries must be made, i.e. a total of $\Omega(n^2)$ queries must be made to determine whether v is a king.

□

5.6 Finding Kings in the Incremental Dynamic Setting

In the incremental dynamic setting, we start with a tournament, and the adversary provides a vertex at a time, and the task is to find a king of this new tournament after a new vertex is added to it without starting from scratch. We give an $O(\sqrt{n})$ algorithm to determine a king under vertex additions. We assume that no vertex is deleted.

The algorithm maintains the following invariants/data:

1. A transitive subtournament on a subset $D = \{v_1, v_2, \ldots v_k\}$ of vertices. For all $j > i$, v_j dominates v_i. Let V' be the set consisting of D and all vertices

dominated by D. Then vertex v_k is a Banks point of $T[V']$.

2. Let $R = V \setminus V'$, and by definition of V', every vertex in R dominates all vertices of D.

3. $|R| \leq 2|D| + 1$.

4. The outdegree of every vertex in R in the subtournament $T[R]$.

When a new vertex v is added, it is probed with vertices in D, and if any vertex in D dominates v, then v is added to V' and we return the king declared earlier as the king for this new tournament too. Otherwise, v is added to R after probing its edges with every vertex in R and updating the degree of every vertex in R including that of v. If R is non-empty, then we output a maximum degree vertex in R as a king, otherwise, we return v_k as the king. The correctness is obvious as a maximum degree vertex x in R reaches all vertices in R by a path of length at most two by lemma 5.2.1. The vertex x also dominates all vertices of D and as D is a dominating set of V', x reaches all vertices in V' by a path of length at most two, and hence x is a king of the entire tournament. If R is empty, then as D is a dominating set and as v_k is the source of D, clearly v_k is a king. Since every vertex in R dominates every vertex in D, we maintain the transitive order when x is added as the last vertex in D.

Clearly the first two invariants on D, V' and R are maintained but we need to maintain the (third) size constraint on R. So when $|R| = 2|D| + 1$, we delete a maximum degree vertex x in R from R and add it to D, make it as v_{k+1} and moving all vertices dominated by x from R to V'. This step of moving vertices from R to D reduces the size of R by at least $|D|$ and increases the size of D by 1, thereby ensuring that the third invariant is always maintained.

Now to analyze the number of probes made, we argue that $|D|$ is maintained as $O(\sqrt{n})$. As $|R| \leq 2|D|+1$ and as the probes of the new vertex are made only with vertices in D and R, the claim that the number of probes made by the algorithm is $O(\sqrt{n})$ will follow.

Lemma 5.6.1. $|D| \leq 2\sqrt{n} + 1$.

Proof. Let $D = \{v_1, v_2, \ldots v_k\}$ be the vertices ordered according to the order at which they are added to D. We first claim that v_k dominates at least $k-1$ vertices (other than itself).

We prove this by induction on k. This is true for v_1. Assume that $k \geq 2$ and the claim is true for values up to $k-1$. Noticing the manner in which the algorithm proceeds, when v_k was added, $|R| = 2(k-1)+1$ and v_k is a maximum degree vertex in R and hence by Lemma 5.2.4, v_k dominates at least $(k-1)$ vertices other than itself and the claim follows.

Thus $(n-k) \geq |V'| \geq \sum_{i=1}^{k}(i-1)$ from which it follows that $(n-k) \geq (k-1)k/2$ from which it follows that $k \leq 2\sqrt{n}+1$.

□

As a preprocessing step, we can simply find a Banks vertex using the algorithm of Lemma 5.2.6 and store the dominating set D of V.

Now we count the time for other bookkeeping operations (beyond the number of edge probes). We notice that we find a maximum outdegree vertex in R whenever R is non-empty. But as we maintain the outdegrees of every vertex in R, this can be found in $\mathrm{O}(|R|) = \mathrm{O}(\sqrt{n})$ time.

When $|R| = 2|D|+1$, we delete a maximum outdegree vertex from R to D and delete from R all vertices dominated by that vertex. Note that $\Omega(\sqrt{n})$ vertices are deleted from R and due to that the outdegree of every other vertex in R may need to be updated. This bookkeeping step could take $\mathrm{O}(n)$ time as every vertex in R now needs to recount its degree in R. Notice that immediately after the step, $|R|$ decreases by at least half its size while $|D|$ increases by 1. Hence, this expensive step which takes $\mathrm{O}(n)$ time will not happen for another $\Omega(\sqrt{n})$ steps. No new edges are queried in R when this split happens, and only the running time gets affected. Thus, the overall amortized time over a long sequence of insertions for updating a king is $\mathrm{O}(\sqrt{n})$. Thus we have

Theorem 5.6.2. *Given a tournament on n vertices, we can maintain a data structure in $\mathrm{O}(n\sqrt{n})$ time so that when a new vertex v is added to the tournament, we can output a king of the new tournament using $\mathrm{O}(\sqrt{n})$ edge probes in the worst case and $\mathrm{O}(\sqrt{n})$ other bookkeeping operations in the amortized case.*

Note that the algorithm may not return a Banks vertex, as the maximum degree vertex of R (which is output as a king) may not be a Banks point of R.

In what follows, we give a modification to the above algorithm where one can actually find a king in the new tournament in $O(\sqrt{n})$ *worst* case time using some standard tricks. Basically when R is split, instead of moving the highest degree vertex to D immediately, we store that in a temporary set, and move it over roughly $\sqrt{n}$ steps by which time, we will be able to update the outdegrees in the remaining vertices in R.

As before the algorithm maintains

1. a Banks vertex v_k of a subtournament $T[V']$ ($V' \subseteq V$) of the given tournament, along with the vertices in the transitive subtournament $v_1, v_2, \ldots v_k$ realizing the Banks vertex where v_k is the source vertex, and the set $D = \{v_1, v_2, \ldots v_k\}$ that dominates V'.

2. We partition $R = V \setminus V'$ into three disjoint sets R_1, R_2 and R_3 and maintain the invariant that $|R| \leq 3|D| + 1$ and $|R_1| \leq 2|D| + 1$.

3. Whenever $R_2 \neq \emptyset$, $|R_2| \geq |D| + 1$ and it contains a vertex m that dominates all other vertices in R_2. Whenever $R_3 \neq \emptyset$, every vertex of R_3, dominates m, the dominator vertex of R_2.

4. We maintain the outdegree of every vertex in R with respect to the subtournament on vertices of R.

5. We maintain the outdegree of every vertex in R_3 with respect to the subtournament on vertices of R_3.

Initially $R = R_1$ and $R_2 = R_3 = \emptyset$.

When a new vertex v is added to the existing tournament, it is probed as before with vertices in D, and if any vertex in D dominates v, then v is added to V', in which case, the king of the tournament just prior to addition of v in the tournament also is a king of the tournament after v is added and the algorithm terminates.

Otherwise, if v dominates all vertices in D, then v is probed with all vertices in R and the count of outdegrees of all vertices in R (with respect to tournament on R) is updated.

If v is added to R it is added to exactly one of the sets R_1, R_2 or R_3 using the following rules

- If $R_2 = \emptyset$ and $R_1 < 2|D|+1$, then v is added to R_1. The outdegree of vertices in R is updated.

- If $R_2 = \emptyset$ and $|R_1| = 2|D| + 1$, then we find the maximum degree vertex in R_1 and label it as m. Move m and all vertices dominated by m from R_1 to R_2, making R_1 of size at most $|D|$ (and thereby making R_2 of size at least $|D| + 1$). If the new vertex v dominates m, v is added to R_2, else it is added to R_3. If $R_1 \neq \emptyset$, an arbitrary vertex is moved from R_1 to R_3. The outdegree of vertices in R is updated. The outdegree of vertices in R_3 (with respect to tournament on R_3) is maintained.

- If $R_2 \neq \emptyset$ and $R_1 \neq \emptyset$, then we probe v with m. If m dominates v, then we add v to R_2, otherwise we add v to R_3 after updating the outdegree of all vertices of R_3 within the subtournament induced on R_3. Regardless of whether we add v to R_2 or R_3), we also remove an arbitrary vertex of R_1 and move it to R_3 and update the outdegrees of vertices in R_3(with respect to tournament on R_3).

- If $R_2 \neq \emptyset$ and $R_1 = \emptyset$, then we add m to D as $v_{|D|+1}$, move all other vertices in R_2 to V' and move all vertices in R_3 to R_1. Now R contains only the vertices which were earlier in R_3. The outdegrees of R_3 were already maintained anyway, and thus the outdegrees of every vertex in R is also maintained.

It is easy to see that the invariants $1, 3, 4, 5$ are maintained. Now we prove that the invariant 2 is also maintained.

Notice that whenever $R_2 = \emptyset$, $|R_1| < 2|D| + 1$ and R_3 are empty. Thus $R \leq 3|D| + 1$ and $|R_1| \leq 2|D| + 1$.

When $|R_1| = 2|D| + 1$ vertices, R_1 gets split and at least $|D| + 1$ vertices move from R_1 to R_2, and a new vertex is added to either R_2 or R_3. Suppose $d \leq |D|$ is size of R_1 at this point. Thus, in this case also $R \leq 3|D| + 1$ and $|R_1| \leq 2|D| + 1$.

When $R_1, R_2 \neq \emptyset$, it means that R_1 has been split, and everytime a new vertex is added to R, it is either added to R_2/R_3 and exactly one vertex in R_1 moves

to R_2/R_3. Before R_1 becomes empty, at most d vertices have moved from R_1 to R_3 and d new vertices are added to R. Thus, whenever $R_2 \neq \emptyset$ and $R_1 = \emptyset$, the maximum size of R is (d(number of new vertices added to R)+d (number of vertices moved from R_1 to R_2/R_3)+$(2|D| + 1 - d)$ (number of vertices in $R_2 \cup R_3$ when R_1 was last split))$= 2|D| + 1 + d$. Since $d \leq |D|$, the maximum size of R is $3|D| + 1$. No vertices are added to R_1 whenever $R_1, R_2 \neq \emptyset$. Thus $|R_1| \leq d \leq |D|$. When $R_2 \neq \emptyset$ and $R_1 = \emptyset$, at least $|D| + 1$ vertices are moved from R to V', which ensures that $|R| < 3|D| + 1$.

When $R_2 \neq \emptyset$ and $R_1 = \emptyset$, all vertices of R_2 are moved to V', and then vertices in R_3 are moved to R_1. Thus at least $|D| + 1$ vertices are moved from R to V', thus $R \leq 2|D|$, and then the sets R_2, R_3 become empty. So $|R| \leq 3|D| + 1$, and $R \leq 2|D| + 1$.

This proves that invariant 2 is always maintained.

Now to output a king vertex of the updated tournament, the algorithm checks if R is empty or not. Whenever R is empty, the Banks vertex of V' is the Banks point of entire tournament and thus it is also a king. Whenever R is non-empty, a maximum degree vertex in R (which happens to be a king of R) is also a king of entire tournament, which is always maintained by the invariant 4. This proves the correctness of the algorithm.

The running time (including the number of queries made) for incrementally maintaining a king is clearly $O(|D|) = O(\sqrt{n})$ in the worst case. Thus, we have

Theorem 5.6.3. *Given a tournament on n vertices, we can build a data structure in $O(n\sqrt{n})$ time so that when a new vertex v is added to the tournament, we can output a king of the new tournament in $O(\sqrt{n})$ time in the worst case.*

The algorithms in Theorem 5.6.2 and Theorem 5.6.3 always return a king, but they do not necessarily return a Banks point at every step. This is because the maximum outdegree vertex of R may not dominate all vertices in R. We could maintain a Banks point within R and return such a Banks point whenever R is non-empty. And when the size invariant on R is violated, it makes sense to move the Banks point along with the dominated vertices to D, but then we move too many vertices to D and this requires care.

First recall that a Banks point is simply the source vertex of a transitive subtournament whose vertices dominate the entire tournament. So if we simply main-

tain the transitive subtournament (which maybe of large size) and compare the newly added vertex to the vertices of the subtournament, we can find a Banks point. If the new vertex dominates all vertices of the transitive subtournament, then it is a Banks point and otherwise the earlier Banks point is the Banks point for the new tournament. But as the size of the transitive subtournament can be $\Theta(n)$ this results in a linear time algorithm to report a new Banks point.

Lemma 5.6.4. *Given a tournament on n vertices, we can build a data structure in* $O(n^2)$ *time so that when a new vertex v is added to the tournament, we can output a Banks point of the new tournament in* $O(n)$ *time in the worst case.*

We shall now use this lemma together with the strategy in Theorem 5.6.3 to improve the bound for finding a Banks point to $O(\sqrt{n})$. In particular, we continue to maintain the invariants $1, 2$ and 3 of Theorem 5.6.3. The invariants 4 and 5 are changed to the following conditions.

- Maintain a Banks point over all vertices in R in addition to maintaining the relation between every pair of vertices in R.
- Maintain a Banks point over vertices R_3 with respect to the subtournament on vertices of R_3.

First, we notice that the algorithm in Theorem 5.6.3 always returns a Banks point whenever the set R is empty. Whenever R is non-empty it only contains vertices which dominate all vertices in D. Thus any Banks point of R is a Banks point of entire tournament.

Initially $R = R_1$ and $R_2 = R_3 = \emptyset$.

When a new vertex v is added to the existing tournament, it is probed with vertices in D, and if any vertex in D dominates v, then v is added to V', in which case, the Banks point of the tournament just prior to addition of v in the tournament also is declared as a Banks point.

Otherwise, if v dominates all vertices in D, then v is probed with all vertices in R and a Banks point of all vertices in R (with respect to tournament on R) is updated in $O(|R|) = O(\sqrt{n})$ time using lemma 5.6.4.

If v is added to R it is added to exactly one of the sets R_1, R_2 or R_3 using the exact same rules as used in Theorem 5.6.3. But there is one more step, which is

performed in addition to the procedure followed in Theorem 5.6.3 while a vertex is added to R. Whenever R and R_3 are not empty, a Banks point with respect to R and a Banks point with respect to R_3 are also maintained. This takes an additional $O(|R| + |R_3|) = O(\sqrt{n})$ time/edge queries by the use of lemma 5.6.4.

The arguments for maintaining invariants, correctness and query complexity are similar to the one showed in Theorem 5.6.3, which gives us the following theorem.

Theorem 5.6.5. *Given a tournament on n vertices, we can build a data structure in $O(n\sqrt{n})$ time so that when a new vertex v is added to the tournament, we can output a Banks point of the new tournament in $O(\sqrt{n})$ time in the worst case.*

5.7 Conclusions and Open Problems

We have investigated the complexity of finding Kings, Banks points and d-dominating sets in tournaments. While our algorithms are not very involved, they are strengthened by the fact that the algorithms to find above $\Omega(\lg n)$ sized d-dominating sets are provably optimal. We have also provided some additional insights into the complexity of finding a d-king which may help narrow the gap between upper and lower bound for the complexity of the problem. We believe that our work will spur further work on improving the bounds for finding a king in tournament. We end with some specific problems from the work on tournaments.

- There is still a gap in the complexity of finding a d-king for $d \geq 2$. We have shown in Corollary 5.4.14 that to improve the upper bound, it is sufficient to improve the upper bound for finding 2-kings. Is the converse true? I.e. can we improve the upper bound for finding d-kings ($d > 2$) even if a 2-king cannot be found in $o(n\sqrt{n})$ time?

- Are the bounds in Theorem 5.4.13 optimal for $k < \epsilon \lg n$, for $d \geq 2$? For $d = 1$, we know that finding a k-sized dominating set is $W[2]$-complete.

- For vertices u and v, let $b(u, v)$ denote the number of vertices through which u can reach v by a directed path of length 2. Then, a king z is strong [53] if the following condition is satisfied: $b(z, v) > b(v, z)$ whenever v dominates z. I.e. if v dominates z, the number of ways to reach v from z is more than

the number of ways to reach x from v through a directed path of length at most 2. It can be shown that a maximum degree vertex in a tournament is a strong king. Can a strong king be found in $o(n^2)$ time? It is also known [54] that we need $\Omega(n^2)$ time to find a maximum degree vertex in a tournament.

- Can we show that the lower bound for finding a Banks point is $\Omega(n\sqrt{n})$?

Chapter 6

Elusiveness of Finding Degrees

6.1 Introduction

In this chapter, we look at the selection problem in directed graphs, undirected graphs and tournaments, which may not have any transitivity property. In this circumstances, the goal is to find vertices having a certain degree or the maximum degree. The topic of this chapter is on a query model which was quite popular based on a conjecture from 70s which is yet unproven. We consider simple (directed or undirected) graphs, where there are no loops and there is at most one (directed or undirected appropriately) edge between any pair of vertices, which can be represented using $\binom{n}{2}$ entries of a matrix. A graph property is said to be elusive (or evasive) [55] if any algorithm to determine the property requires probing all $\binom{n}{2}$ entries of the adjacency matrix in the worst case. In this model, only the number of queries is only counted, and the algorithm is allowed to do any other operations for free.

Several graph properties are known to be elusive, e.g having a clique of size k or a coloring with k colors [56] or having atleast one edge in the graph. A property is monotone if it remains true when edges are added. For example a non-empty graph stays nonempty even after adding an edge. A.L. Rosenberg [55] conjectured that any deterministic algorithm must query at least a constant fraction of entries in the adjacency matrix in the worst case, to determine if the graph has a given non-trivial monotone graph property. This was proven by Rivest and Vuillemin [57]. A stronger version of the conjecture called Aanderaa-Karp-Rosenberg conjecture was

also formulated [55], which stated that exactly $\binom{n}{2}$ probes are needed to determine whether a monotone property is elusive. The stronger Aanderaa-Karp-Rosenberg conjecture remains unproven. See [58] for more information on recent developments.

One can also define elusiveness for other problems on graphs (not necessarily properties, for example, finding a vertex with maximum degree) and for properties of directed graphs [59]. There are also non-monotone properties that are elusive. Hougardy and Wagler [60] showed that the property of being perfect, though not a monotone property, is still an elusive property. In this chapter, we show that another non-monotone property, the graph having a vertex of outdegree $k \leq (n+1)/2$ in directed graphs, is elusive. This improves an earlier lower bound of $n(n-1-k)/2$ [54] for $k > 1$. For $k = (n-1)/2$, for example, the earlier bound was $n(n-1)/4$ whereas we improve the bound to $\binom{n}{2}$ for all values of $k \leq (n+1)/2$. Existence of a vertex with degree k is not a monotone property, since adding an edge in the graph may make the graph to lose all its vertices with degree exactly k.

We also address the complexity of finding a vertex of degree k in an undirected graph. We show that determining whether an undirected graph has a vertex of degree 0 or 1 is elusive, while for larger k, we could show a lower bound of $.42n^2$ improving the previous lower bound of $.25n^2$. We also show that finding the maximum degree in an undirected graph or a directed graph requires $\binom{n}{2}$ queries to the adjacency matrix.

It is known [54] that finding a maximum outdegree vertex in a tournament requires $\binom{n}{2} - 2n + 3$ queries, but it was not known whether one needs $\binom{n}{2}$ probes. We show that one can find a maximum outdegree vertex in $\binom{n}{2} - 1$ queries, and improve the lower bound to $\binom{n}{2} - 2$ when n is odd and $\binom{n}{2} - n/2 - 2$ when n is even. All our lower bounds are shown using simple, but subtle adversary arguments.

6.1.1 Organization of the Chapter

In Section 6.2, we give some definitions and conventions and some results we use. In Section 6.3, we show results for finding degree k vertices in directed or undirected graphs. In Section 6.4, we show results for finding the maximum degree as well as a

maximum degree vertex in directed or undirected graphs. In Section 6.5, we show improved lower bounds and upper bounds for finding a maximum degree vertex in a tournament. Section 6.6 lists some interesting open problems.

6.2 Definitions and Conventions

All our graphs (directed or undirected) are simple graphs that have no loops and have at most one edge between any pair of vertices. The outdegree of a vertex u in a directed graph is the number of vertices v such that there is a directed edge from u to v, and the indegree of vertex is defined analogously. Sometimes we say degree to mean outdegree when dealing with directed graphs. For a subset S of vertices of a graph, the induced subgraph on S ($G[S]$) is the graph with vertex set S, and the edge set that contains pairs of vertices of S that are edges in the graph G. In a directed graph/tournament, when we say that a vertex u wins vertex v, we mean that there is an edge directed from u to v. By the same token, we say that v looses to u. A tournament is an orientation of a complete graph, i.e. a tournament is a directed graph in which there is exactly one directed edge between every pair of vertices. A round-robin over the tournament (or a subtournament) involves finding the direction of all the edges in the (sub)tournament. A regular tournament is a tournament in which every vertex has the same indegree and the same outdegree. The following result is well known.

Lemma 6.2.1 ([38])**.** *If a tournament on n vertices is regular, then n is odd. Furthermore, for every odd integer n, there exists a regular tournament on n vertices.*

The adjacency matrix of a graph on n vertices is an n by n matrix that has its (i, j)-th entry 1 whenever there is an edge (i, j) and 0 otherwise. In the case of directed graphs if (i, j) is a (directed) edge, then the (i, j)-th entry has 1 and the (j, i)-th entry has a -1. Clearly, in undirected graphs, the matrix is symmetric, and in directed graphs, the matrix is anti-symmetric.

All our proofs use an adversary argument. Here an adversary maintains the adjacency matrix, and the algorithm probes the adversary for entries of the matrix. The adversary has a fixed strategy and creates entries in the matrix, when they are queried by the algorithm. Our argument to prove a lower bound of $f(n)$ typically

has the form, 'if the algorithm does not probe $f(n)$ entries, then the adversary has enough options on the rest of the graph to prove the algorithm wrong, whatever the algorithm answers'.

6.3 Finding Vertices of Degree k

6.3.1 Directed Graphs

Earlier known lower bound [54] for finding a outdegree k vertex in a directed graph is $n(n-1-k)/2$ for any $0 < k \leq n-1$. For $0 < k \leq (n+1)/2$, we improve the bound to show the following.

Theorem 6.3.1. *Any algorithm to determine whether a given directed graph has a vertex with outdegree k $(0 < k \leq (n+1)/2)$ on n vertices, requires $\binom{n}{2}$ probes to the adjacency matrix.*

Proof. The adversary constructs the digraph in following manner depending on the value of k:

1. *k is even and at least* 2 The adversary partitions the graph into two sets of vertices A and B. A contains $2(k-1)+1$ vertices. B contains $n-2k+1$ vertices. Adversary maintains a regular tournament (that exists by Lemma 6.2.1) on A (each vertex has outdegree and indegree exactly $(k-1)$) and the induced subgraph on B is empty (edgeless). There is no edge directed from A towards B. For every vertex $x \in B$, the adversary would add directed edges from x to the first $k-1$ vertices of A it is probed with. This way, it is clear that there is no vertex in the graph with outdegree k.

   In order to determine that no vertex in A is a outdegree-k vertex, the algorithm is required to probe at least $n-k$ non-outdegree edges of each vertex in A. This would make any algorithm to probe the entire subtournament on A and every edge between A and B. For otherwise, the adversary can flip/add an edge out of that vertex to make that vertex of degree k consistent with its answers for all vertices. To prove that no vertex in B is an outdegree k vertex, the algorithm has to probe all pairs within B, as otherwise, the adversary can add an edge to make a vertex of outdegree k.

2. *k is odd and $k \geq 3$* The adversary partitions the graph into two sets A and B by taking $2k-2$ vertices in A and the rest in B. Adversary maintains a regular subtournament A' inside A on $2k-3$ vertices (each vertex within A' has indegree and outdegree $(k-2)$) and the remaining 1 vertex z of A will have all edges directed to the vertices of the regular subtournament. The rest of the construction for edges across A and B and within B are the same as in the previous case.

   In order to claim that no vertex in A' is a degree-k vertex, the algorithm is required to probe at least $n-k$ non-outdegree edges of each vertex in A'. This would make the algorithm to probe the entire subtournament on A' and every edge between A' and $B \cup \{z\}$. Although z already may have degree more than k, queries with z are required to verify whether any vertex in B or A' can get degree k or not.

   As in the previous case, the algorithm has to probe all pairs within B to prove that no vertex in B is an outdegree k vertex.

3. $k = 1$ The adversary maintains an empty (edgeless) graph and answers the queries according to it. If the algorithm leaves some edge unprobed between a pair of vertices, say u and v, then following two cases happen:

   (a) If the algorithm declares any vertex other than u and v as 1-outdegree vertex, then adversary would trivially show that the algorithm is wrong, as it has never given any edge, till this point.

   (b) If the algorithm declares u (without loss of generality) as a 1-outdegree vertex, then adversary would add an edge $v \to u$, making vertex v the unique vertex of outdegree 1 proving the algorithm wrong. □

□

For $k = 0$, we improve the earlier known lower bound [54] of $n^2/4$ to show the following.

Theorem 6.3.2. *Determining whether a given directed graph on n vertices has a vertex with outdegree 0 requires $\binom{n}{2}$ probes to the adjacency matrix.*

Proof. For $n = 2$, it is straightforward to see that the only edge in the graph has to be probed, to determine whether any of the two vertices has outdegree 0. For $n \geq 3$, the adversary strategy is as follows: For the first query between two vertices, say x and y, the adversary answers that the graph has an edge $y \to x$ and adds y to a set W which is a set maintained by the adversary, that contains all vertices with outdegree at least 1. The vertex x is now marked as a sink vertex s. The label of the sink vertex is not fixed and may change over the course of the algorithm's execution.

If the algorithm asks a query (u, w) for a vertex $w \in W$, and $u \notin W$, and $u \neq s$, the adversary answers that there is no edge unless w is the only vertex of W with which u has not been probed, in which case it directs the edge from u to w and adds u to W. For a query involving a pair of vertices (u, v), $u, v \notin W \cup \{s\}$, the adversary answers that there is no edge.

For a query between sink s and y, the adversary answers that there is no edge unless y is the last vertex (not necessarily in W) with which sink s has not been probed, in which case the adversary gives an edge from s to y and adds s to W. If in the process y gets indegree 1 (and outdegree 0) then y plays the role of new sink and marked as s, otherwise there is no sink. This strategy allows the adversary to maintain the following invariants:

- The set W contains vertices whose outdegree so far is exactly one.

  Proof. Each vertex gets its first outdegree while entering W, after which point it only get indegrees. □

- Every probe between vertices in W has been already made by the algorithm.

  Proof. If there is an unprobed edge between two vertices u and v, where $v \in W$, then u will not get an outdegree edge, and thus it would not be present in W. □

- There is at most one vertex outside W, referred to as sink s (if it exists) that has indegree exactly one.

- Vertices other than those in $W \cup \{s\}$ have outdegree and indegree 0.

It is clear that all the above invariants are maintained by the answers of the adversary. The adversary never gives an edge, if the edge between two vertices both of which are outside W is made. Thus, to confirm whether a vertex has 0 outdegree or not, it has to compare with vertices in W. The only time when the algorithm can discard a vertex for a candidate of 0-degree vertex, is when it is added to W, after it gets its first outdegree.

Suppose an algorithm has not queried an edge between vertices u and v, and it answers that $w(\neq u, v)$ is a 0-outdegree vertex. If $w \notin W$, then there is at least one vertex (say w') with which it has not made a query, and its outdegree thus far is 0. So the adversary can give an edge $w \to w'$ and prove the algorithm wrong. If $w \in W$, then the algorithm is wrong, since all vertices in W have at least one outdegree.

If the algorithm outputs u as a 0-outdegree vertex (without loss of generality), then the adversary can give the edge $u \to v$ and contradict the algorithm. Thus all pairs of queries have to be made, to determine whether there is any vertex with outdegree 0. □ □

For $k > (n+1)/2$, we improve the known lower bound of $n(n-k-1)/2$ to show the following.

Theorem 6.3.3. *For $k > (n+1)/2$, determining whether there is a vertex with outdegree k in a directed graph on n vertices requires at least $(n-k-1)(n+k)/2$ probes to the adjacency matrix.*

Proof. The adversary maintains two sets A and B. A contains $k+1$ vertices, with no edge between any pair of vertices within A. B contains $n-k-1$ vertices, with no edge between any pair of vertices within B. For any vertex in B, the adversary will direct the first $k-1$ vertices of A with which it is probed towards A, and will answer that there is no edge with the remaining two vertices of A (if probed). The algorithm is required to probe all edges out of each vertex of B to prove that none of them is of degree k (as otherwise, the adversary can create edges to prove the algorithm wrong). The number of queries probed is $(k+1)(n-k-1) + (n-k-1)(n-k-2)/2 = (n-k-1)(n+k)/2$. □ □

6.3.2 Undirected Graphs

Similar to the adversary for determining outdegree 1 vertex in a directed, graph, we can show the following by maintaining an adversary that does not give any edge between pairs of vertices. If some edge is unprobed, then then the adversary can keep the option of making both of them a degree 1 vertex.

Theorem 6.3.4. *Determining whether a given undirected graph has a degree* 1 *vertex is an elusive property.*

For degree 0, the adversary is similar to the adversary for outdegree 0 in the case of directed graphs. We show the following theorem.

Theorem 6.3.5. *Determining whether an undirected graph on n vertices has a degree* 0 *vertex requires all $n(n-1)/2$ probes to the adjacency matrix of the graph.*

Proof. For the first query between two vertices, adversary would add an undirected edge and place both of them in a set W. After that, if the algorithm asks a query (u, w) for a vertex $w \in W$, and $u \notin W$, the adversary answers that there is no edge unless w is the only vertex of W with which u has not been probed, in which case it gives an edge between u and w and adds u to W. For a query involving a pair of vertices (u, v), $u, v \notin W$, the adversary answers that there is no edge.

Thus the adversary maintains that

- vertices in W have degree at least 1,
- all pairs of vertices in W have been probed, and
- all vertices outside W have degree 0.

So the algorithm cannot declare any vertex of W as one of degree 0 as it will be wrong otherwise. If $V(G) \setminus W \neq \emptyset$, and if the algorithm declares a vertex of $V(G) \setminus W$ as a vertex of degree 0, then the algorithm can make the degree of that vertex at least 1, as that vertex has still some edge unprobed (as otherwise it will be in W). If the algorithm declares that there is no vertex of degree 0, then $W = V$, and in that case, all edges between pairs of vertices in $W = V$ have been probed due to the invariant maintained by adversary.□

□

For $k > 1$, we are unable to show whether the property is elusive or not, but we improve the known lower bound from $n^2/4$ [54] to show the following.

Theorem 6.3.6. *Finding a degree k vertex in an undirected graph on n vertices requires at least $(0.42)n^2$ probes to the adjacency matrix of the graph.*

Proof. Without loss of generality we can assume that $k \leq \lfloor (n-1)/2 \rfloor$. For $k \geq (n-1)/2$, one can simply find a vertex with degree $n-1-k$ in the complement graph which can be obtained online from the adjacency matrix by interpreting $1s$ as $0s$ and vice versa.
Adversary has a choice to answer in one of following two ways, which it decides based on the value of k:

1. $k \leq (3-\sqrt{5})n/2$: Adversary constructs the graph as follows:
   The vertices are divided into two sets A and B. A consists of $k+2$ vertices. B consists of $n-k-2$ vertices. Each vertex of B has an edge with every vertex of A. The subgraph of the set B contains no edge and the subgraph of the set A is complete and hence every vertex of A has degree at least $k+1$. For each vertex in B, adversary would give edges to the first $k-1$ vertices of A with which it is probed, and answer the last three edges with set A as empty, to make each vertex of B of degree $k-1$. To prove that no k degree vertex exists in B, the algorithm would probe every edge of subgraph B.
   Hence the algorithm has to probe all edges incident on vertices of B which is $(n-k-2)(k+2)+(n-k-2)(n-k-3)/2 = (n-k-2)(n+k+1)/2$ probes.

2. $k > (3-\sqrt{5})n/2$:-Adversary constructs the graph as follows:
   The graph is divided into two sets of vertices A and B. A contains $n-k+1$ vertices and B contains $k-1$ vertices. The subgraph on each of the sets A and B is complete. Each vertex of B has degree at least $k-2$ and each vertex of A has degree of at least $k+1$ (because $k \leq \lfloor\ (n-1)/2\ \rfloor$).
   In order to claim that some vertex $x \in B$ is of degree $< k$, the algorithm is required to probe between x and at least $n-k$ vertices of A. Adversary would add an edge between x and the last 3 vertices of A it is probed with. This would make x a degree $k+1$ vertex. Adversary would do the same for

each such vertex of B. Now, the algorithm has to probe the whole subgraph on B, to prove that none of the vertex in B is a degree k vertex.
In order to prove that none of vertex in A is a degree k vertex, the algorithm would probe at least $k+1$ edges of each vertex of A. As there can be at most $3k-3$ edges between A and B, the algorithm is required to probe the subgraph on A for the remaining edges. Algorithm would stop, when each vertex of A has degree of $k+1$. Suppose e is the number of edges that algorithm is forced to probe inside the set A, to make each vertex of A degree $k+1$. Then, $2e+3(k-1) \geq (n-k+1)*(k+1)$ and so $e \geq (n-k+1)(k+1)/2 - 3(k-1)/2$. Thus the algorithm would probe at least $(k-1)(n-k+1)+(k-1)(k-2)/2+(n-k+1)(k+1)/2-(3k-3)/2 = (k(3n-2k-2)-n+4)/2$ probes.

The minimum value of complexity of both strategies is when $k = (3-\sqrt{5})n/2$, which gives a lower bound of $.42n^2$ probes. □ □

6.4 Finding the Maximum Degree

6.4.1 Directed Graphs

The adversary for $k=1$ in Theorem 6.3.1 also gives the following

Theorem 6.4.1. *For a directed graph on n vertices, any algorithm would require to probe at least $n(n-1)/2$ queries to the adjacency matrix to find a maximum outdegree vertex.*

The same adversary implies that finding the maximum outdegree (not just a vertex with the maximum outdegree) requires $\binom{n}{2}$ probes.

Corollary 6.4.2. *Finding the maximum outdegree in a directed graph on n vertices, requires all $n(n-1)/2$ queries to the adjacency matrix.*

6.4.2 Undirected Graphs

The adversary strategy used to show that finding a degree 1 vertex in an undirected graph is an elusive property (Theorem 6.3.4) shows the following.

Theorem 6.4.3. *Finding the maximum degree in an undirected graph requires all* $n(n-1)/2$ *edge queries in the worst case.*

However, if we don't care about the maximum degree, and is sufficient to find a vertex with maximum degree, then, we can show the following.

Theorem 6.4.4. *Finding a maximum degree vertex in an undirected graph on* n *vertices, has a lower and upper bound of* $n(n-1)/2 - 1$.

Proof. Lower bound: Whenever a query comes for any pair of vertices x and y, the adversary answers that there is no edge between x and y, except when all of the remaining unprobed edges share the same vertex, in which case the adversary would work as follows: If the query between vertices u and v is the last query, after which all queries will have a common vertex x, then adversary would add an edge (u, v) to the graph, and not give any more edges to x. Now, the algorithm has to probe all the edges of x except the last one, to prove that vertex u or v is indeed a maximum degree vertex. For, if the algorithm omits two edges out of x, the adversary can make x the unique maximum degree vertex.
If such a case does not happen i.e, if there exist two unprobed edges that do not share any vertex, say u_1v_1 and u_2v_2. Then the following two cases arise:
Case 1: If the algorithm declares any vertex x other than u_1,v_1,u_2 and v_2 as a maximum outdegree vertex, then adversary would add an edge u_1v_1 to make vertex u_1 as the maximum degree vertex, contradicting the algorithm's claim.
Case 2: Without loss of generality, if the algorithm declares vertex u_1 as maximum degree vertex, then the adversary would add edge u_2v_2 to make vertex u_2 of degree 1, contradicting the algorithm's claim.

Upper bound:- The algorithm would probe all edges except the last edge, say between u and v. If there exists some vertex x with degree larger than that of any other vertex including u and v, then the algorithm declares x as a maximum degree vertex, as both u and v cannot get degree more than that of x, even if edge uv exists.

Without loss of generality, if u has degree greater or equal to that of any other vertex, then the algorithm would declare u as a maximum degree vertex (since presence or absence of an edge between u and v won't affect the fact that u has

maximum degree). If both u and v are of the largest degree, then either of u and v can be output by algorithm as a maximum degree vertex. □ □

6.5 Tournaments

A tournament is a directed graph in which there is exactly one directed edge between every pair of vertices. Balasubramanian et al.[54] gave a $2kn$ lower bound and $4kn$ upper bound for finding a vertex of outdegree k $(0 < k \leq (n-1)/2)$. Bridging this gap is still open for general value of k. In this section we deal with finding the maximum outdegree vertex in a tournament. The previous known lower bound for finding a maximum outdegree vertex in tournament was $\binom{n}{2} - 2n + 3$ [54]. We give an improved lower bound and also show that such a vertex can be found without probing all the edges in the tournament.

6.5.1 Lower Bound for Finding Maximum Outdegrees

Theorem 6.5.1. *Finding a vertex with the maximum outdegree in a tournament on n vertices requires $\binom{n}{2} - 2$ edge-probes if n is odd and $\binom{n}{2} - n/2 - 2$ edge-probes if n is even.*

Proof. When n is odd, the adversary starts by maintaining a regular tournament on n vertices where each vertex has outdegree and indegree $(n-1)/2$. Our first claim is the following.

Claim 6.5.2. *Suppose the algorithm declares a vertex x as one with maximum outdegree. Then the algorithm must have ensured that the remaining $(n-1)$ vertices have indegree at least $(n-1)/2$.*

Otherwise, the adversary can flip an unprobed edge coming into one of these vertices, say y, and make y the unique vertex with maximum outdegree proving the algorithm wrong.

This claim already shows a lower bound of $(n-1)(n-1)/2$. We improve the bound further by modifying the adversary as follows. The adversary keeps track of vertices with indegree $(n-1)/2$, and when the $(n-1)^{th}$ vertex z is about to get indegree $(n-1)/2$, the adversary flips the edge coming into z. Let y be

the n^{th} vertex, now every vertex other than y and z are known to have indegree $(n-1)/2$ and z has indegree $(n-1)/2-1$. Now the adversary will continue to answer according to its initial choice of orientations (of the regular tournament) making z the vertex with the unique maximum outdegree. But to rule out y as a candidate, the algorithm has to ensure that y has indegree at least $(n-1)/2-1$ (as otherwise it can flip an edge coming into y to make it a unique vertex with maximum outdegree). All these indegree edges are disjoint, but it is possible that the edge that was going to come into z that got flipped was actually from y counting for the indegree of y. Hence, the algorithm has to perform at least $(n-1)(n-1)/2+(n-1)/2-2=(n-1)n/2-2$ comparisons.

Suppose n is even. The adversary constructs a tournament that has two sets of vertices: X is a set of $n/2$ vertices with outdegree $n/2$ each and Y is a set of $n/2$ vertices with outdegree $n/2-1$ each. Before declaring v which is a vertex of X as a vertex with maximum outdegree, the algorithm has to ensure that every other vertex has indegree at least $n/2-1$ as otherwise the adversary can make one of those vertices to have outdegree $n/2+1$ by flipping an unprobed incoming edge to one of them, thus contradicting the algorithm. This proves a lower bound of $(n-1)(n/2-1)$. As before, we further improve the lower bound as follows. The algorithm flips the edge coming into the $(n-1)^{th}$ vertex (say u) that is about to get indegree $n/2-1$ so that it and the n^{th} vertex (say v) are candidates for maximum outdegree. Now the adversary answers in such a way that both u and v dont get more than $n/2+1$ outdegree, thus the algorithm has to ensure that both get indegree at least $n/2-2$. This, as before, results in a lower bound of $(n-2)(n/2-1)+2(n/2-2)=n(n-1)/2-n/2-2$. □ □

Theorem 6.5.3. *Finding the maximum outdegree (along with a vertex with the maximum outdegree) in a tournament on n vertices requires at least $\binom{n}{2}$ comparisons if n is odd, and $\binom{n}{2}-n/2$ comparisons if n is even.*

Proof. It is evident that one cannot find out the maximum outdegree of a graph, without actually knowing which vertex has the maximum outdegree, although the converse is not true. If n is odd, the adversary works with a regular tournament, and if some edge is not probed, it can flip the edge and change the maximum outdegree.

If n is even, the adversary works with a tournament where $n/2$ vertices have outdegree $n/2$, and other $n/2$ vertices have outdegree $n/2-1$. Any algorithm which declares the maximum outdegree(which is $n/2$ in this case) along with the vertex which has that degree (say u) has to ensure that all the vertices including u have indegree at least $n/2-1$, because if a vertex, say v, with indegree less than $n/2-1$ exists, then the adversary will make v have outdegree $n/2+1$, and thus contradict the algorithm. □ □

Corollary 6.5.4. *Any algorithm to determine all vertices with the maximum outdegree requires* $\binom{n}{2}$ *edge-probes.*

Proof. For odd sized graph, it follows from Theorem 6.5.3. For even sized tournaments, the adversary is similar to the one shown in Theorem 6.5.3. There are $n/2$ maximum degree vertices and each of the $n/2$ maximum outdegree vertices must have their all edges probed, otherwise the adversary could alter any edge to prove the algorithm wrong. Also, the algorithm needs to probe all indegrees of the remaining $n/2$ vertices to prove that none of them is a max-outdegree vertex. Thus algorithm has to probe all indegree edges of each vertex of the tournament. □ □

6.5.2 Upper Bound for Finding a Vertex with Maximum Outdegree

Theorem 6.5.5. *Finding a vertex with the maximum outdegree in a tournament on n vertices, requires at most* $\binom{n}{2}-1$ *edge-probes for $n>2$.*

Proof. Consider the subtournament T on some $n-1$ vertices and perfom a round robin tournament by making all possible comparisons between them taking $(n-1)(n-2)/2$ edge-probes. Let x be the remaining vertex. Two possible cases arise for T.

1. T is a regular subtournament: In this case, every vertex in T has outdegree/indegree $(n-2)/2$.

   Choose an arbitrary vertex y of T. Probe x with all vertices of T except y. We will argue that we have sufficient information to determine a vertex with

the maximum outdegree. Let z be a vertex in $T - \{y\}$, which is a vertex with the maximum degree among vertices in T. At this point,

(a) if z lost to x, then x is a maximum outdegree vertex.

(b) if z won against x and outdegree of x is less than the outdegree of z, then z is a maximum outdegree vertex.

(c) if z won against x and outdegree of x is at least the outdegree of z, then output x as the maximum outdegree vertex.

2. T is not a regular subtournament:- Let y be a vertex of T with minimum outdegree in T, which is at most $n/2 - 2$. Probe x with all vertices of T except y. Let z be a vertex in $T - \{y\}$, which is the vertex with the maximum outdegree among vertices of $T - \{y\}$. At this point,

(a) if outdegree of x is less than outdegree of z, then output z as the maximum outdegree vertex.

(b) if outdegree of x is the same or more than outdegree of z, then output x as the maximum outdegree vertex.

The correctness of the algorithm is clear, and in both cases, we have probed at most $n(n-1)/2 - 1$ edges. □ □

Note that the gap between the current upper bound (Theorem 6.5.5) and the lower bound (Theorem 6.5.1) for finding a maximum degree vertex is 1 when n is odd and is $n/2 + 2$ when n is even.

6.6 Conclusions and Open Problems

We have addressed the complexity of finding vertices of degree k or maximum degree in a directed or an undirected graph or a tournament in a model where only probes to the adjacency matrix are counted. While our adversaries are simple, they are still non-trivial and they narrow the gap between upper and lower bounds substantially. We end with the following open problems.

1. Is $n(n-1)/2-1$ the optimal bound for finding a maximum degree vertex in an odd sized tournament and is $n(n-1)/2-n/2-1$ the optimal bound for finding a maximum degree vertex in an even sized tournament? We believe that that the answer is yes when n is odd and no for n is even. For example, when $n=4$, we can find a vertex with the maximum outdegree in three probes (simply by declaring the winner of a knock-out tournament). Extending this argument/bound for larger even sized tournaments would be interesting.

2. Is determining whether an undirected graph has a degree k vertex an elusive property for $k>1$?

Bibliography

[1] Biswas, A., V. Jayapaul, and V. Raman (2017) "Improved Bounds for Poset Sorting in the Forbidden-Comparison Regime," in *Algorithms and Discrete Applied Mathematics - Third International Conference, CALDAM 2017, Sancoale, Goa, India, February 16-18, 2017, Proceedings*, pp. 50–59.

[2] Jayapaul, V., J. I. Munro, V. Raman, and S. R. Satti (2015) "Sorting and Selection with Equality Comparisons," in *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, pp. 434–445.

[3] Jayapaul, V., V. Raman, and S. R. Satti (2016) "Finding Mode Using Equality Comparisons," in *WALCOM: Algorithms and Computation - 10th International Workshop, WALCOM 2016, Kathmandu, Nepal, March 29-31, 2016, Proceedings*, pp. 351–360.

[4] Biswas, A., V. Jayapaul, V. Raman, and S. R. Satti (2017) "The complexity of finding (approximate sized) distance d dominating sets in tournaments," in *Frontiers in Algorithmics - 11th International Workshop, FAW 2017, June 23-25, 2017 at Chengdu, Sichuan, China.*

[5] Goyal, D., V. Jayapaul, and V. Raman (2017) "Elusiveness of Finding Degrees," in *Algorithms and Discrete Applied Mathematics - Third International Conference, CALDAM 2017, Sancoale, Goa, India, February 16-18, 2017, Proceedings*, pp. 242–253.

[6] Huang, Z., S. Kannan, and S. Khanna (2011) "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*, pp. 738–747.

[7] Daskalakis, C., R. M. Karp, E. Mossel, S. J. Riesenfeld, and E. Verbin (2011) "Sorting and Selection in Posets," *SIAM Journal on Computing*, **40**(3), pp. 597–622.

[8] Banerjee, I. and D. Richards (2016) "Sorting Under Forbidden Comparisons," in *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, vol. 53 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 22:1–22:13.

[9] Tucker, A. (1974) "Structure theorems for some circular-arc graphs," *Discrete Mathematics*, **7**(1-2), pp. 167–195.

[10] Durán, G., L. N. Grippo, and M. D. Safe (2014) "Structural results on circular-arc graphs and circle graphs: A survey and the main open problems," *Discrete Applied Mathematics*, **164**, pp. 427–443.

[11] Hell, P., J. Bang-Jensen, and J. Huang (1990) "Local Tournaments and Proper Circular Arc Gaphs," in *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, pp. 101–108.

[12] Blum, M., R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan (1973) "Time Bounds for Selection," *J. Comput. Syst. Sci.*, **7**(4), pp. 448–461.

[13] Kahn, A. B. (1962) "Topological sorting of large networks," *Commun. ACM*, **5**(11), pp. 558–562.

[14] Golumbic, M. C. and W. Rheinboldt (2014) *Algorithmic Graph Theory and Perfect Graphs*, Computer science and applied mathematics, Elsevier Science.

[15] Tarjan, R. E. and M. Yannakakis (1984) "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs," *SIAM Journal on Computing*, **13**(3), pp. 566–579.

[16] Rose, D. J., R. E. Tarjan, and G. S. Lueker (1976) "Algorithmic Aspects of Vertex Elimination on Graphs," *SIAM Journal on Computing*, **5**(2), pp. 266–283.

[17] Komlós, J., Y. Ma, and E. Szemerédi (1998) "Matching Nuts and Bolts in O(n log n) Time," *SIAM J. Discrete Math.*, **11**(3), pp. 347–372.

[18] Mathieu, C. and H. Zhou (2013) "Graph Reconstruction via Distance Oracles," in *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, pp. 733–744.

[19] Culberson, J. C. and P. Rudnicki (1989) "A Fast Algorithm for Constructing Trees from Distance Matrices," *Inf. Process. Lett.*, **30**(4), pp. 215–220.

[20] Reyzin, L. and N. Srivastava (2007) "On the longest path algorithm for reconstructing trees from distance matrices," *Inf. Process. Lett.*, **101**(3), pp. 98–100.

[21] ——— (2007) "Learning and Verifying Graphs Using Queries with a Focus on Edge Counting," in *Algorithmic Learning Theory, 18th International Conference, ALT 2007, Sendai, Japan, October 1-4, 2007, Proceedings*, pp. 285–297.

[22] Thorup, M. (2003) "Integer priority queues with decrease key in constant time and the single source shortest paths problem," in *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pp. 149–158.

[23] Boyer, R. S. and J. S. Moore (1991) "MJRTY: A Fast Majority Vote Algorithm," in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pp. 105–118.

[24] Alonso, L. and E. M. Reingold (2013) "Analysis of Boyer and Moore's MJRTY algorithm," *Inf. Process. Lett.*, **113**(13), pp. 495–497.

[25] Alonso, L., E. M. Reingold, and R. Schott (1997) "The Average-Case Complexity of Determining the Majority," *SIAM J. Comput.*, **26**(1), pp. 1–14.

[26] ——— (1993) "Determining the Majority," *Inf. Process. Lett.*, **47**(5), pp. 253–255.

[27] Saks, M. E. and M. Werman (1991) "On computing majority by comparisons," *Combinatorica*, **11**(4), pp. 383–387.

[28] Fischer, M. J. and S. L. Salzberg (1982) "Finding a Majority Among n Votes: Solution to Problem 81-5 (Journal of Algorithms, June 1981)," *Journal of Algorithms*, **3**(4), pp. 362–380.

[29] Dobkin, D. P. and J. I. Munro (1980) "Determining the Mode," *Theor. Comput. Sci.*, **12**, pp. 255–263.

[30] Demaine, E. D., A. López-Ortiz, and J. I. Munro (2002) "Frequency Estimation of Internet Packet Streams with Limited Space," in *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, pp. 348–360.

[31] Reingold, E. M. (1972) "On the Optimality of Some Set Algorithms," *J. ACM*, **19**(4), pp. 649–659.

[32] Munro, J. I. and P. M. Spira (1976) "Sorting and Searching in Multisets," *SIAM J. Comput.*, **5**(1), pp. 1–8.

[33] Misra, J. and D. Gries (1982) "Finding Repeated Elements," *Sci. Comput. Program.*, **2**(2), pp. 143–152.

[34] Munro, J. I. and Y. Nekrich (2015) "Compressed Data Structures for Dynamic Sequences," in *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pp. 891–902.

[35] Bollobás, B. (1978) *Extremal graph theory*, Academic Press.

[36] Dirac, G. A. (1952) "Some Theorems on Abstract Graphs," *Proceedings of the London Mathematical Society*, **s3-2**(1), pp. 69–81.

[37] Devanny, W. E., M. T. Goodrich, and K. Jetviroj (2016) "Parallel Equivalence Class Sorting: Algorithms, Lower Bounds, and Distribution-Based Analysis," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pp. 265–274.

[38] Moon, J. (1968) *Topics on tournaments*, Athena series: Selected topics in mathematics, Holt, Rinehart and Winston.

[39] Ajtai, M., V. Feldman, A. Hassidim, and J. Nelson (2016) "Sorting and Selection with Imprecise Comparisons," *ACM Trans. Algorithms*, **12**(2), pp. 19:1–19:19.

[40] Shen, J., L. Sheng, and J. Wu (2003) "Searching for Sorted Sequences of Kings in Tournaments," *SIAM Journal on Computing*, **32**(5), pp. 1201–1209.

[41] Lu, X., D. Wang, and C. K. Wong (2000) "On the bounded domination number of tournaments," *Discrete Mathematics*, **220**(1-3), pp. 257–261.

[42] Papadimitriou, C. H. and M. Yannakakis (1996) "On Limited Nondeterminism and the Complexity of the V-C Dimension," *J. Comput. Syst. Sci.*, **53**(2), pp. 161–170.

[43] Graham, R. L. and J. H. Spencer (1971) "A constructive solution to a tournament problem," *Canad. Math. Bull.*, **14**, pp. 45–48.

[44] Alon, N. and J. Spencer (1992) *The Probabilistic Method*, John Wiley.

[45] Megiddo, N. and U. Vishkin (1988) "On Finding a Minimum Dominating Set in a Tournament," *Theor. Comput. Sci.*, **61**, pp. 307–316.

[46] Downey, R. and M. Fellows (1999) *Parameterized Complexity*, Springer New York.

[47] Brandt, F., F. A. Fischer, and P. Harrenstein (2009) "The Computational Complexity of Choice Sets," *Math. Log. Q.*, **55**(4), pp. 444–459.

[48] Dey, P. (2017) "Query Complexity of Tournament Solutions," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pp. 2992–2998.

[49] Maurer, S. B. (1980) "The King Chicken Theorems," *Mathematics Magazine*, **53**(2), pp. 67–80.

[50] Penn, E. M. (2006) "The Banks Set in Infinite Spaces," *Social Choice and Welfare*, **27**(3), pp. 531–543.

[51] Yao, F. F. (1974) *ON LOWER BOUNDS FOR SELECTION PROBLEMS*, *Tech. rep.*, Cambridge, MA, USA.

[52] Brandt, F., F. A. Fischer, and P. Harrenstein (2007) "Recognizing Members of the Tournament Equilibrium Set is NP-hard," *CoRR*, **abs/0711.2961**.

[53] Chen, A.-H., J.-M. Chang, Y. Cheng, and Y.-L. Wang (2008) "The existence and uniqueness of strong kings in tournaments," *Discrete Mathematics*, **308**(12), pp. 2629 – 2633.

[54] Balasubramanian, R., V. Raman, and G. Srinivasaragavan (1997) "Finding Scores in Tournaments," *J. Algorithms*, **24**(2), pp. 380–394.

[55] Rosenberg, A. L. (1973) "On the Time Required to Recognize Properties of Graphs: A Problem," *SIGACT News*, **5**(4), pp. 15–16.

[56] Bollobás, B. (1976) "Complete subgraphs are elusive," *J. Comb. Theory, Ser. B*, **21**(1), pp. 1–7.

[57] Rivest, R. L. and J. Vuillemin (1976) "On Recognizing Graph Properties from Adjacency Matrices," *Theor. Comput. Sci.*, **3**(3), pp. 371–384.

[58] Miller, C. A. (2013) "Evasiveness of Graph Properties and Topological Fixed-Point Theorems," *Foundations and Trends in Theoretical Computer Science*, **7**(4), pp. 337–415.

[59] King, V. (1990) "A lower bound for the recognition of digraph properties," *Combinatorica*, **10**(1), pp. 53–59.

[60] Hougardy, S. and A. Wagler (2004) "Perfectness is an Elusive Graph Property," *SIAM J. Comput.*, **34**(1), pp. 109–117.