DIMACS Technical Report 98-05
January 1998

The Set-Maxima Problem: an Overview

by

Richard Desper[1,2]

[1]DIMACS Graduate Fellow
[2]Department of Mathematics, Rutgers University, Piscataway, NJ 08903, USA. (*desper@math.rutgers.edu*) Supported in part by NSF grant 9215293 and in part by grant NSF BIR 94-12594.

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

Sorting problems have long been one of the foundations of theoretical computer science. Sorting problems attempt to learn properties of an unknown total order of a known set. We test the order by comparing pairs of elements, and through repeated tests deduce some order structure on the set.

The set-maxima problem is: given a family S of subsets of a set X, produce the maximal element of each element of S. Local sorting is a sub-problem of set-maxima, when $S \subseteq \binom{X}{2}$, i.e. there exists a graph G with vertex set X and edge set S. We compare algorithms by estimating the number of comparisons needed, as a function of $n = |X|$, and $m = |S|$.

In this paper, we review the information-theory lower bounds for the set-maxima and local-sorting problems. We review deterministic algorithms which have optimally solved the set-maxima problem, as a function of m, n, in settings where extra assumptions about S have been made. Also, we review randomized algorithms for local sorting and set-maxima which achieve an optimal expected number of comparisons.

1 Introduction

This paper provides an exposition of work done on the following two closely related problems:

The set maxima problem: Given a family S of m subsets of a totally ordered set $X = \{x_1, \ldots, x_n\}$ use comparisons between elements of X to find the maximum in each subset.

The local sorting problem: Given a graph $G = (V, E)$ with an underlying total order on V, use comparisons between elements of V to orient the elements of E in accordance with the total order.

Note that the local sorting problem is a specialized case of the set-maxima problem, where each of the subsets has exactly two elements. We will analyze these problems under the comparison-tree model, where algorithms are judged solely by the number of comparisons they use between the elements of the underlying set. Any other computations are free, under this model.

If m is kept fixed, the set-maxima problem is easily solved in linear time. One partitions X into blocks with x_i and x_j in the same block if they are members of exactly the same sets of S. Then one uses fewer than n comparisons to find the maximum of each block, and a constant number of additional comparisons to solve the problem.

Unfortunately, the constant cost implied by such a method increases exponentially with m. If $m = o(\log n)$, such an approach will still be better than the $n \log n$ cost of sorting X.

On the other hand, for $m = \Theta(n^2)$ the complexity of the problem may force $n \log n$ comparisons. For example, if X is the underlying set of the complete graph on n vertices, and S is the set of all edges in the complete graph, then solving the set maxima problem is equivalent to sorting the underlying set. In this case, the best possible algorithm requires $\Omega(n \log n)$ comparisons.

The complexity of the set-maxima problem is easily seen to be $\Omega(n)$, yet $O(n \log n)$. This is about all that is known, excepting the specialized cases where fast algorithms have been found. Komlós found a linear time algorithm in the setting of verifying the minimality of a spanning tree of a graph. Bar-Noy, Motwani, and Naor solved the problem in two specialized settings: in the case where the sets are hyperplanes in a projective geometry, and in the case where set membership is random. A general randomized algorithm for set-maxima and for local sorting was provided by Goddard, Kenyon, King, and Schulman. The expected running time of the algorithm is linear.

This paper starts with a discussion of the solution sets to these problems. Analysis of the sizes of these sets yields the information-theory lower bound for the complexity of the problems. We will compare these lower bounds to the calculated running speeds of our algorithms. In many cases, we will show that the algorithm runs at a speed which is the same order of magnitude as the information-theory lower bound.

The second section includes a discussion of the k-selection algorithm, a simple, yet powerful algorithm to find the k-th largest element of a set, using only $O(n)$ comparisons. The k-selection algorithm, when applied repeatedly to a set, can be used to partition the set into intervals, whose endpoints are of specified ranks. This application is referred to as *comput-*

ing a rank-sequence. The importance and utility of the k-selection algorithm justifies its complete exposition.

The third and fourth sections provide expositions of deterministic algorithms for a few specific instances of the set-maxima problem, including the minimal spanning-tree verification algorithm of Komlós and the rank-sequence algorithm of Bar-Noy, Motwani, and Naor. These algorithms are applicable for a few specific cases, in settings which currently appear to be more tractable that the general case.

The final section presents the randomized algorithms of Goddard, Kenyon, King, and Schulman. To understand the GKKS set-maxima algorithm, one needs to first study their local-sorting algorithm, as the former makes calls to the latter. The local-sorting algorithm is illustrated in the limited-degree case, and in the general case. The set-maxima algorithm is presented as a specific case of a powerful randomized algorithm, which sorts the t largest elements of a set. Bounds are provided for the expected number of comparisons. These bounds are shown to be optimal.

2 Information-Theory Lower Bounds

Suppose we have an algorithm which, to solve a problem, is choosing one solution from a set of solutions, $\mathcal{S}$. An algorithm may be represented as a rooted tree. Namely, each step of the algorithm is represented as a node of the tree, with children corresponding to each of the possible next steps. Each leaf of the tree represents an element of the set of solutions, though one may have many leaves corresponding to the same solution. The complexity of the algorithm is the height of the tree, which represents the largest number of steps taken before a solution is reached. If h denotes the height of a tree, the number of leaves is at most 2^h. This observation implies the converse inequality: if our algorithm is choosing an element of the set $\mathcal{S}$, then its complexity must be at least $\log|\mathcal{S}|$. This quantity is referred to as the *information-theory lower bound.*

2.1 Feasible Vectors

Research on the set-maxima problem was initiated by the following proof that the information theory lower bound is low. This result of Fredman is taken from [GYY].

Let $v = (v_1, .., v_m)$ be called a *feasible vector* on (X, S) if there is some ordering of the underlying set X such that $v_i = \max S_i$ for all i. Let V be the set of all feasible vectors on (X, S). Let $L = (x_{j_1}, x_{j_2}, \dots, x_{j_m})$ be called a *list of representatives* with respect to (X, S) if for all i, $x_{j_i} \in S_i$. For a fixed list of representatives, L, let r_k denote the number of i such that $j_i = k$. The vector $(r_1, \dots, r_n)$ is called the *signature* of L. The list L is *s-unique* if no other list L' has the same signature as L.

Theorem 2.1 *A list of representatives $L = (x_{j_1}, \dots, x_{j_m})$ is feasible iff it is s-unique.*

Corollary 2.1 $|V| < 2^{m+n}$.

Proof of Theorem: Necessity: Suppose L is a feasible vector which is not s-unique. Let $L' = (x_{j'_1}, \ldots, x_{j'_m})$ be another vector with the same signature as L. Assume X is ordered so as to realize L, i.e. $x_{j_i} = \max(S_i)$ for all i.

The map $j_i \to j'_i$ induces an automorphism of the multiset $\{x_{j_1}, \ldots, x_{j_m}\}$ which is non-increasing, and strictly decreasing at least once (if non-trivial), an impossibility. Thus if L is a feasible vector, it is s-unique.

Sufficiency: Suppose L is not feasible. Construct the directed graph G whose vertex set is X and whose edges are the edges (x_{j_i}, x_k) such that $x_k \in S_i$ and $k \neq j_i$. The fact that L is not feasible implies that G contains a directed cycle, say $((x_{l_1}, x_{l_2}), \ldots, (x_{l_{d-1}}, x_{l_d}), (x_{l_d}, x_{l_1}))$. By definition of G, there exist distinct $h_1, h_2, \ldots, h_d$ such that $x_{l_u}, x_{l_{u+1}} \in S_{h_u}$ $(1 \leq u < d)$, $x_{l_d}, x_{l_1} \in S_{h_d}$, and x_{l_u} is the representative chosen from S_{h_u} for $1 \leq u \leq d$. However, from the sets S_{h_u} we can pick instead the representatives $x_{l_{u+1}}$ if $u < d$, x_{l_1} if $u = d$. Picking the same representatives from the remaining S_k as before we obtain a different list of representatives with the same signature. Thus L is not s-unique. □

Proof of Corollary: As the number of distinct signatures is at most $\binom{m+n-1}{n-1}$, which is less than 2^{m+n}, the corollary follows. □

The information theory lower bound for the complexity of the set-maxima algorithm is $\log |V|$. The corollary shows that $\log |V| < m + n$, which is too low to be of interest as a lower bound.

2.2 Acyclic Orientations

Solving the local sorting problem for a graph G is equivalent to choosing an acyclic orientation of G. The total number of such acyclic orientations is given the unfortunate designation $\alpha(G)$ by [GKKS]. To avoid confusing this parameter with the independence number, we will use the label $acorns(G)$ or $a(G)$ to denote the number of acyclic orientations. The term "acorns," has been used by L. Schulman in this context.

For our first proposition, we shall use *depth-first searches,* which we formalize with the following construction. Given a directed graph $G = (V, E)$, we fix an ordering of $V = \{v_i | 1 \leq i \leq n\}$. We will create an n-tuple $(y_1, y_2, \ldots, y_n)$, where y_i is the ith vertex visited by a certain depth-first search. This search starts at v_1, and moves through the graph along the path taken by a standard depth-first search, breaking ties by using the vertex numbering.

To start, let $y_1 = v_1$. Our search begins at this vertex. We iterate the following for $i = 1, 2, \ldots$. Let $Y_i = \{y_j | 1 \leq j \leq i\}$. Then for $j = 1, \ldots, i$, let $N_{ij} = \{v_k \in V \setminus Y_i | (y_j, v_k) \in E(G)\}$. The set N_{ij} is composed of vertices the search could go to from y_j at the ith stage of iteration. If for all j, $N_{ij} = \emptyset$, let $k_i = \min\{k | v_k \in V \setminus Y_i\}$. In this case, our search moves to a new vertex and starts a new search. Otherwise, let $j(i) = \max\{j \leq i | N_{ij} \neq \emptyset\}$, and let $k(i) = \min\{k | v_k \in N_{ij(i)}\}$. Let $y_{i+1} = v_{k(i)}$. The vertex $v_{k(i)}$ is the next vertex visited by our search.

We call the n-tuple $(y_1, \ldots, y_n)$ the *depth-first search list* associated to the ordering of V. This list generates a unique rooted spanning forest of G, namely the forest F whose edges are traversed along the path of the search. Its edge set is $E(F) = \{(y_i, y_j) | i < j, i = \max\{k \leq j | (y_k, y_j) \in E(G)\}\}$. Note that a depth-first search of F will coincide with the search of G which generated F.

Let $f(x) = (x!)^{\frac{1}{x}}$. In studying the graph G, some parameters of interest are $F(G) = \prod_{v \in G} f(d_v + 1)$, $D(G) = \prod_{v \in G}(d_v + 1)$, $a(G)$ and $For(G)$, where $For(G)$ denotes the number of rooted spanning forests on G.

Proposition:

The following relationships hold between these four parameters:

$$\sqrt{D(G)} \leq F(G) \leq a(G) \leq For(G) \leq D(G).$$

Proof:

1. $\sqrt{D(G)} \leq F(G)$. This inequality follows directly from the fact that for any positive, integral value of x, $\sqrt{x+1} \leq f(x+1)$.

2. $F(G) \leq a(G)$. This proof, due to [GKKS], uses induction on $|G|$. Let δ denote the minimal degree of G. The case $\delta = 0$ is done by induction. Assuming $\delta \geq 1$, let v be a vertex of minimal degree. Then

   $$a(G) \geq (\delta + 1)a(G - v),$$

   as any acyclic orientation of $G - v$ may be extended in at least $\delta + 1$ ways. Let N denote the set of neighbors of v in G. Then

   $$F(G) = F(G - v)f(\delta + 1)(\prod_{w \in N} \frac{f(d_w + 1)}{f(d_w)}),$$

   and since $f(x+1)/f(x)$ is monotonically decreasing,

   $$F(G) \leq F(G - v)f(\delta + 1)(\frac{f(\delta + 1)}{f(\delta)})^{\delta}.$$

   Applying the induction hypothesis yields the inequality $F(G) \leq a(G)$, our desired result.

3. $a(G) \leq For(G)$. (This proof is from [MT].) Index V as $V = \{v_1, v_2, \ldots, v_n\}$. For each acyclic orientation O of G we use a depth-first search based on this ordering to define a forest, $F(O)$, following the method described earlier.

   We show that this assignment is one-to-one. To reverse the assignment, suppose that $F = F(O)$ for some acyclic orientation O, of the graph G. Let $\leq_F$ denote the partial order on V induced by F, where $(v, w) \in F$ implies $w \leq_F v$. From F and the vertex numbering of V, we create the depth-first search list $(y_1, y_2, \ldots, y_n)$ by searching F. As noted above, this list is identical to the list generated by the depth-first search of G as oriented by O. We use this list and $\leq_F$ to reconstruct O.

(a) If the edge $y_i y_j$ is in the unoriented graph G, and $y_i \leq_F y_j$, then $y_i \leq_O y_j$. All edges of F are oriented by O just as they are by F, and any edge of G not in F, but whose orientation can be inferred from the partial order on F, is oriented in agreement with this partial order.

(b) For $i < j$, if the edge $y_i y_j$ is in G, and y_i and y_j are incomparable under $\leq_F$, then $y_i <_O y_j$, as $y_i >_O y_j$ would imply that the depth-first search would force $y_i >_F y_j$, a contradiction.

Thus the assignment of rooted spanning forest to acyclic orientation is 1-1.

4. $For(G) \leq D(G)$. To each rooted spanning forest F on G, assign the function $a : G \to G$, where $a(v) = v$ if v is a root in F, otherwise $a(v) = w$, where w is the parent of v in F. The number of such functions is at most $D(G)$ and the assignment of functions is clearly 1-1, so we have the inequality. □

In particular, the proposition says that the information-theory lower bound on the complexity of the local-sorting problem is simply expressed as

$$\log a(G) = \Theta(\log(D(G))).$$

Thus, for example, if each vertex has degree polylogarithmic in n, then one may hope for a local-sorting algorithm which requires no more than $O(n \log\log n)$ comparisons.

Concrete lower bounds for the complexity of the set-maxima problem, when $m > n$, can be seen as follows:

Let $f(m, n)$ denote the largest number of comparisons needed to solve the set maxima problem over all sets of m subsets of a set of size n. It is clear that $f(m, n) = \Omega(n)$ as finding the maximum of a single set of size n requires $n - 1$ comparisons.

For values of m in the range $m = \omega(n)$, and $m = O(n^2)$, we observe a lower bound of $f(m, n) = \Omega(n \log(m/n))$. Assume that $2m/n$ and $n^2/(2m+n)$ are integers. If we partition the universe into $n^2/(2m+n)$ blocks of size $(2m+n)/n$, and take as our sets the edges of the complete graph on each block, then finding the maxima of the m edges will force the complete sorting of each block. Thus the complexity of the algorithm must be at least

$$(\frac{n^2}{2m+n})(\frac{2m+n}{n})\log(\frac{2m+n}{n}) = \Omega(n \log(m/n))$$

comparisons. A slightly modified argument will show this bound for all values of m, n in the aforementioned range.

For larger values of m, in the range $m = \Omega(n^2)$, we note that $m \geq \binom{n}{2}$ implies that the collection of subsets could include all of the edges of a complete graph on n vertices. Finding the maxima of these sets would force a complete sorting of the underlying set, which requires $\Omega(n \log n)$ comparisons. This cost is an upper bound on the complexity of set-maxima, as a complete sort of the underlying set will solve set-maxima.

3 The k-selection algorithm

Given a set X, with an underlying total order, for each $x \in X$, let $r(x) = |\{y | y \geq x\}|$, referred to as the *rank* of x. If $r(x) = k$, we call x the element of rank k. The k-selection problem requires that we use comparisons on X to determine the element of rank k. For $k = 1$, this problem is equivalent to the problem of finding the maximum of a set, which requires $n - 1$ comparisons.

The discussions throughout this section, including the historical background and the algorithms, are taken from [Kn].

For values of k greater than one, the k-selection problem dates back to the efforts of the reknowned author and mathematician, the Rev. C. L. Dodgson (better known as Lewis Carroll), who recognized the inadequacy of the standard tennis tournament in determining the second- and third- best players of the tournament. In the standard single-elimination tournament, the runner-up has only proven his superiority over half of the players of the tournament. Carroll devised a method to determine the second-best player by adding more matches to the tournament. His method could not be described as an algorithm, as the plan was restricted to thirty-two player tournaments. Furthermore, his method did not minimize number of matches needed to make such a determination.

In 1929-30, Steinhaus formalized the *tennis tournament problem:* What is the minimum number of matches required to determine the best and second best players in a tournament of n players, for $n \geq 2$. This problem translates to a problem in algorithms. We suppose that there is an underlying order to the set of players, namely their ability level, and that each match is a comparison between players, which reflects the underlying order. We wish to study how many matches/comparisons are needed to answer this problem. In the following arguments, I will use the words "matches" and "comparisons" interchangeably, and likewise for "players" and "elements."

In 1932, Schreier produced an algorithm which uses $n - 2 + \lceil \log n \rceil$ matches to answer this question. His algorithm first uses a standard single-elimination tournament to determine the best player. Such a tournament is represented mathematically as a binary tree of height $\lceil \log n \rceil$, where the leaves represent the players, and the internal vertices represent the matches between the players. In such a tournament, there may be as many as $\lceil \log n \rceil - 1$ players who have lost to the champion.

To find the second best player, we run a tournament among the players who lost to the champion. The player who wins the second tournament is the second-best player. This second tournament requires only $\lceil \log n \rceil - 1$ comparisons. When added to the $n - 1$ comparisons of the original tournament, this yields a total cost of $\lceil \log n \rceil + n - 2$ comparisons. Schreier provided a faulty proof for the optimality of his approach. It was not until 1962 that Kislitsyn proved that Schreier's algorithm was optimal.

Let $V_t(n)$ denote the minimum number of comparisons needed for an algorithm to determine, with worst-cast input, the t-th largest of n elements, for $1 \leq t \leq n$.

Theorem 3.1 *With the above terminology,* $V_2(n) \geq n - 2 + \lceil \log n \rceil$.

Proof: Assume that T is an algorithm which optimally finds the second-best player in a set of players, X, where $|X| = n$. Under the comparison-tree model, it is impossible to determine the second-greatest element without also determining the maximum. We will choose an ordering of the set, which will force T to use at least $n - 2 + \lceil \log n \rceil$ comparisons.

We associate the following parameters with T, for purposes of cost analysis. For $i \geq 1$, let a_i denote the number of players of X which lose at least i matches as T is conducted. Since T determines a champion, $a_1 = n - 1$, as every player other than the champion must lose at least one match. It suffices to show that we can force $a_2 \geq \lceil \log n \rceil - 1$.

Let c denote the number of players the champion defeats head-to-head. Then one of these c players is the second-best. The others must lose at least one other time, as T determines their inferiority to the second-greatest player. Thus $a_2 \geq c - 1$.

Suppose an oracle decides the outcomes of matches according to the rules: in a match between A and B

- If A is undefeated at the time of the match, but B is not undefeated, then A defeats B.
- If A and B are both undefeated, but A has won more matches before the current match than B has, then A defeats B.
- Other matches are decided arbitrarily.

We conduct T, using the oracle to determine the outcomes of matches. In doing so, we define a partial order, $\geq_F$, by saying $A \geq_F B$ if A is the first player to defeat B, and closing transitively. We use the notation $S(A) = \{B | A \geq_F B\}$ for all players A. Let $S(A, q)$ be the natural analogue of $S(A)$ to the first q matches of T.

Let C denote the maximum of X. Then for all $A \in X$, $C \geq_F A$. This is easily shown, using induction on the rank of A.

Lemma 3.1 *For all q, if at most p of the first q matches involve A, then $|S(A, q)| \leq 2^p$.*

Proof of Lemma: We use induction on p. For $p = 0$, the statement is trivial. For $p \geq 1$, let $A' \in X$ be the p-th opponent of A. The interesting case is when A defeats A'; let us assume this happens. Then $S(A, q) \subseteq S(A, q-1) \cup (S(A', q-1) \cap S(A, q))$. By our inductive hypothesis, $|S(A, q-1)| \leq 2^{p-1}$. Note $S(A', q-1) \cap S(A, q) \neq \emptyset$ only if A is the first player to defeat A'. But, since the oracle is determining matches, such a result could only happen if A' had been involved in at most $p - 1$ matches before his match with A. In such a case, $|S(A', q-1)| \leq 2^{p-1}$, by induction, so $|S(A, q)| \leq 2^p$. □

Thus, since $C \geq_F A$ for all $A \in X$, it follows that $c \geq \log n$. This implies that $a_2 \geq \lceil \log n \rceil - 1$. We have forced T to use at least $n - 2 + \lceil \log n \rceil$ comparisons, proving the theorem. □

Hadian and Sobel generalized Schreier's bound to $V_t(n) \leq n - t + (t-1)\lceil \log(n + 2 - t) \rceil$, using a replacement-selection algorithm, which is a variation on the tree-selection algorithm of Schreier. The following proof is taken from Knuth.[Kn]

Theorem 3.2 $V_t(n) \le n - t + (t-1)\lceil \log(n+2-t) \rceil$.

Proof:(From [Kn].) Given X, such that $|X| = n$, let X_1 be some subset of X, such that $|X_1| = n + 2 - t$. Let $Y = \{y_1, y_2, \dots, y_{t-2}\} = X \setminus X_1$. Run a single-elimination tournament on X_1, to find its maximum x_1. Let $X_2 = X_1 \cup y_1 \setminus x_1$. Run a second tournament, replacing x_1 with y_1, to find x_2, the maximum of X_2. This may be done using only $\lceil \log(n+2-t) \rceil$ new comparisons, along the path taken by x_1 in the first tournament. Repeat: for $i = 3$ to $t-1$, let $X_i = X_{i-1} \cup y_{i-1} \setminus x_{i-1}$. Find x_i, the maximum of X_i, using at most $\lceil \log(n+2-t) \rceil$ comparisons by replacing x_{i-1} with y_{i-1}.

Finally, let $X_t = X_{t-1} \setminus x_{t-1}$. Find x_t, the maximum of X_t, using at most $\lceil \log(n+2-t) \rceil - 1$ comparisons, beyond those used to find x_{t-1}. This time, instead of replacing x_{t-1} with another element of X, we delete it, and run a tournament on the rest of the players. The element x_t is the tth greatest element of X, as $x_i > x_t$, for $i < t$, but $x \le x_t$ for all $x \in X_t$.

The first run of the tournament requires $n + 1 - t$ comparisons, each of the $t - 2$ substitutions requires an additional $\lceil \log(n+2-t) \rceil$ comparisons, and the final run requires $\lceil \log(n+2-t) \rceil - 1$ comparisons. Thus the total cost is $n - t + (t-1)\lceil \log(n+2-t) \rceil$ comparisons, which is the upper bound claimed for $V_t(n)$. □

Hyafil generalized the "oracle" argument of Theorem 2.1, yielding the lower bound

$$V_t(n) \ge n - t + (t-1)\lceil \log \frac{n}{t-1} \rceil$$

for all values of t in the range $1 \le t \le n$. For $t = n/2$, this lower bound is $n - 1$, which is uninteresting, while the upper bound of Theorem 2.2 is

$$V_{n/2}(n) \le n/2 + n/2\lceil \log(n/2 + 2) \rceil.$$

The upper bound is asymptotically equal to $(n/2)\log n$, which is already half the cost of sorting the entire set. The reader is again referred to [Kn].

In 1971, Blum produced a method of finding the median with $O(n \log\log n)$ comparisons. Finally, in 1973, Rivest and Tarjan produced a linear-time algorithm for the problem. Using their algorithm, they proved $V_t(n) \le 15n - 163$, for all values of $n > 32$, and all values of t, $1 \le t \le n$.

The SELECT algorithm which follows is a variation of the algorithm of Rivest and Tarjan. The difference is that the SELECT algorithm divides the original set into sets of size five, whereas the original algorithm divides the original set into sets of size seven. This change produces a slightly different cost. The algorithm SELECT is taken from [AHU].

The algorithm SELECT takes as its input a set S of size n, with an unknown underlying total order, and a value $k \le n$. It returns the k-th highest element of the set, x, as well as a partition of $X \setminus x$ into the set $\{y > x\}$ and the set $\{y < x\}$. The algorithm works recursively, making two calls to itself.

We get a good recursive bound for the cost of SELECT: there is a constant $\epsilon > 0$ such that the sum of the sizes of the two sets upon which the recursive calls are made is less than $(1-\epsilon)n$; moreover, the algorithm uses at most cn comparisons beyond those used in the recursive calls. Thus, if we let $T(n)$ denote the greatest cost of using SELECT on a set of size n, for all values of k in the range $1 \leq k \leq n$, there is a recursive inequality $T(n) \leq cn + T(n_1) + T(n_2)$, where $n_1 + n_2 < (1-\epsilon)n$. An easy argument shows that this recursive inequality implies the existence of a constant C such that $T(n) \leq Cn$ for all values of n.

The algorithm is as follows:

SELECT(k,S): (returns the element of rank k of the set S)

1. If $|S| < 50$ then sort S, and return the element of rank k.
2. ELSE begin
   (a) Divide S into $\lfloor \frac{n}{5} \rfloor$ sets of size 5, with one (possibly empty) remainder set of fewer than 5 elements.
   (b) Sort each of the 5-element sets, and form the set M comprised of the medians of these sets.
   (c) Call SELECT$(\lfloor \frac{|M|}{2} \rfloor, M)$ to find the median, m, of the set M.
   (d) Form the sets $S_1 = \{x \in X | x > m\}$ and $S_2 = \{x \in X | x < m\}$.
      - If $|S_1| \geq k$, return the output of SELECT(k, S_1); ELSE
      - If $|S_1| = k - 1$, return m; ELSE
      - Return the output of SELECT$(k - |S_1| - 1, S_2)$.
3. end

The first call of the algorithm (in (c)) finds an element m which is reasonably close to the middle of X, so the sizes of both the sets S_1 and S_2 are bounded by δn, for a constant value of $\delta < 1$. The second call (in (d)) finds the element of rank k, by recursively applying the algorithm to one of the (suitably small) sets S_1 or S_2.

Cost analysis:

Let $c_1 = \lceil \log 50 \rceil$. Then for all $n < 50$, a set of size n can be sorted with fewer than $c_1 n$ comparisons, so this algorithm will find the element of rank k with at most $c_1 n$ comparisons.

Let $T(n)$ denote the maximum cost, for all values of $1 \leq k \leq n$, of running SELECT on a set of size n. Fix $n \geq 50$, and $k \leq n$. As we run SELECT on a set of size k, there are three different costs. Let C_1 denote the cost of sorting the five-element sets, C_2 the cost of the recursive call upon M, and C_3 the cost of the recursive call upon S_1 or S_2. The cost of the current call of SELECT is $C_1 + C_2 + C_3$.

Sorting a five-element set can be done with 10 comparisons, so $C_1 \leq 10\lfloor \frac{n}{5} \rfloor \leq 2n$. Once m is found by the recursive call, forming the sets S_1, S_2, and S_3 can be done with at most n more comparisons. The size of the median sequence M is $\lfloor n/5 \rfloor$, so $C_2 \leq T(\lfloor n/5 \rfloor)$.

The final cost, C_3, depends on the relative sizes of the sets S_i. At least $\lfloor n/10 \rfloor$ of the elements of M are less than or equal to m. Since each of these elements is the median element of a five element set, for each of them, there are two more elements less than or equal to m, giving a grand total of at least $3\lfloor n/10 \rfloor$ elements of S less than or equal to m. Thus $|S_1| \leq n - 3\lfloor n/10 \rfloor \leq 7n/10 + 3$ for values of $n \geq 50$. An identical argument shows that $|S_2| \leq 7n/10 + 3$. Thus $C_3 \leq T(\lfloor 7n/10 + 3 \rfloor)$.

The cost function T thus satisfies:

$$T(n) \leq c_1 n$$

for $n < 50$, and

$$T(n) \leq 3n + T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor)$$

for $n \geq 50$.

Since $c_1 > 3$, an easy inductive argument shows that $T(n) \leq 20c_1 n$ for all values of n.□

4 Minimal Spanning Tree Verification

Suppose G is a connected graph ($G = (V, E)$), such that $|V| = n$, and $|E| = m$, and suppose $\phi : E \to \Re^+$ is a weight function associated to G. A minimal spanning tree T is a spanning tree of G which minimizes the sum $wt(T) = \sum_{e \in T} \phi(e)$.

Consider the family $\mathcal{T}$ of spanning trees of G. Let $G(\mathcal{T})$ be the graph on $\mathcal{T}$, with the edge set $E(G(\mathcal{T})) = \{T_1T_2 \| E(T_1) \cap E(T_2)| = n - 2\}$. Two trees, T and T' are adjacent in this graph if there is only one edge in $T \setminus T'$.

Claim 4.1 *For any two trees S and S' in $\mathcal{T}$, there exists a path $p(S, S') = (S_0, S_1, \ldots, S_k) \in G(\mathcal{T})$ with $S = S_0, S' = S_k$, and $k = |S \setminus S'|$.*

Proof: The proof is by induction on k. For $k = 1$, the statement is obvious. Suppose the claim is true for $k = n - 1$. If $k = n$, let e be an edge in $S' \setminus S$. Let $f \in C_e \setminus S'$. Let $S_1 = T \setminus \{e\} \cup \{f\}$. Then $|S_1 \setminus S'| = n - 1$, so by induction, there exists a path $(S_1, S_2, \ldots, S_k = S')$ in $G(\mathcal{T})$. We may append $S = S_0$ to the beginning of this path, yielding the desired path. □

For $T, T' \in G(\mathcal{T})$, we shall denote by $p(T, T')$ the path in $G(\mathcal{T})$ described in the above claim.

Suppose a spanning tree T is presented as a candidate for minimality. For each edge e of $G \setminus T$, there is a unique cycle $C_e \subset (T \cup e)$.

Claim 4.2 *Suppose T' is a spanning tree satisfying $wt(T') < wt(T)$. Then there is at least one edge $e \in T' \setminus T$ and an edge $f \in C_e \setminus \{e\}$, such that $\phi(e) < \phi(f)$.*

Proof:

Let $p(T, T') = (T_0T_1, T_1T_2, \ldots, T_{k-1}T_k)$, be the path given by Claim 4.1 such that $k = |T \setminus T'|$, $T_0 = T$ and $T_k = T'$. For $1 \leq i \leq k$, let e_i be the unique edge in $T_i \setminus T_{i-1}$, and let f_i be the unique edge in $T_{i-1} \setminus T_i$. If for all i, $\phi(e_i) \geq \phi(f_i)$, then since $wt(T') - wt(T) = \sum_{i=1}^{k}(\phi(e_i) - \phi(f_i))$, we conclude that $wt(T') \geq wt(T)$, a contradiction. Thus there must exist an i for which $\phi(e_i) < \phi(f_i)$, and this pair will satisfy the claim. □

So, to verify the minimality of T, it suffices to compare $\phi(e)$ with $\phi(f)$, for each pair of edges (e, f) satisfying $e \notin T, f \in C_e \setminus e$. As it would be preferable to make only one comparison for each $e \notin T$, we proceed to find $\max\{\phi(f)\}_{f \in C_e \setminus e}$ for each $e \notin T$.

Thus the minimal spanning tree problem translates to a set-maxima problem, where the maxima must be found over the family of sets $\{C_e \setminus e | e \in G \setminus T\}$. Actually, since ϕ is not required to be a 1-1 function, this problem is not precisely a set-maxima problem, if we take the existence of an underlying total order as part of the definition of "set-maxima." Relaxing the requirement that ϕ be 1-1 does not change the formulation of the set-maxima problem very much. Instead of finding the maximum of each subset, we may take any of possibly many maximal elements of each subset.

Komlós presents an $O(m+n)$ algorithm for solving this instance of the set-maxima problem. The total cost of the algorithm is less than $5n + n \log \frac{m+n}{n}$. For $m = \omega(n), n \log \frac{m+n}{n} = o(m)$, so the second term of the cost of the algorithm will be dominated by the cost of $m - n$ comparisons between the values of ϕ on the edges of $G \setminus T$ and the corresponding maxima of the paths $C_e \setminus e$. The complexity of this algorithm is thus of optimal order of magnitude.

Suppose we make T into a rooted tree by designating some vertex, v_1, as the root. There is a natural partial order on T, denoted $\leq_T$, gotten by taking $x >_T y$ if x is on the unique v_1y path of T. The root is the maximal element under this partial order.

Every path in T is either monotically directed away from v_1, or splits into two subpaths, each of which is oriented away from v_1, where the split occurs at the maximal element of the path with regard to $\leq_T$. We will find the maxima for all such oriented subpaths obtained from paths in the set $\{C_e \setminus e | e \in G - T\}$. Recall that $n = |V|$, and $m = |E|$. There are at most $2(m - n + 1)$ sets after splitting, and as $2(m - n + 1)$ is usually not equal to m, we are making a slight departure from our standard notation. If we solve the problem of finding these maxima in $O(m + n)$ time, and make the extra $2(m - (n - 1))$ comparisons of comparing, for each $e \notin T$, the value $\phi(e)$ with the two maxima of the corresponding half-paths of $C_e \setminus e$, we get an $O(m + n)$ solution to the minimal spanning tree verification problem.

For purposes of implementation, we alter the problem in the following fashion. Rather than assigning weights to edges, for each edge $e = xy$, we assign the weight $\phi(e)$ to $\min\{x, y\}$, where the minimum is taken with respect to $\leq_T$. Also, for $x >_T y$ we denote the path from x to y in T as $p(x, y)$. If we assign the value $-\infty$ to the root, this problem has the following formulation:

Given a tree T, rooted at v_1, the induced partial order $\leq_T$, an associated weight function, ϕ, defined on the vertex set, and a set A of directed paths (queries) of T, where $A \subset Q = \{p(x, y) | x, y \in V(T), x \geq_T y\}$, find $\max\{\phi(u) : x \geq_T u \geq_T y\}$ for all $p(x, y)$ in A.

First, we consider two specialized instances: (1) When T is a string, (2) when T is a full-branching tree. A string is a rooted tree which has only vertices of out-degree 1 or 0 (i.e. it's a path from the root). A full-branching tree is a tree where all non-leaves have at least two children, and all of the leaves are on the same level (i.e. at the same distance from the the root). A general tree will fall somewhere between these two extremes.

4.1 The SYMHEAP algorithm

In this section, we consider the case when T is a string. Let us index the vertices such that $V(T) = \{v_i | 1 \le i \le n\}$ with edges $\{(v_i, v_{i+1}) | 1 \le i \le n-1\}$. In this case, there are $\binom{n}{2}$ different intervals $[v_i, v_j]$, but Komlós provides the following algorithm, called SYMHEAP, which requires at most $2n$ comparisons to create a symmetric heap which contains all the information needed to find the maxima for all of the intervals.

Definition: Given a vertex set $V = \{v_i | 1 \le i \le n\}$, a *binary symmetric heap* H on V is a rooted tree such that:

1. The tree is *binary:* no vertex has more than two children. For our purposes, each non-leaf will be assumed to have two children, one or both of which might be the empty set, $\emptyset$.

2. The tree is a *heap:* for each vertex, there is a left-to-right order for its children.

3. The heap is *symmetric:* if $v_i < v_j$, then if $i < j$, v_i is in the left subtree of v_j, while if $i > j$, v_i is in the right subtree of v_j.

The SYMHEAP algorithm forms a symmetric heap H, on the vertex set $V \cup \{v_0\}$, where v_0 is an extra vertex added to be the root. The order on the heap will be such that $v_i \ge_H v_j$ implies $\phi(v_i) \ge \phi(v_j)$. Thus, once the heap is constructed, finding $\max\{\phi(v_j) | i \le j \le k\}$ is equivalent to finding the lowest common ancestor of v_i and v_k in the heap. Processing common ancestor queries does not contribute to the cost of the algorithm under our model, as it requires no comparisons between values of ϕ. Actually, in this case we observe a linear bound even if we strengthen our model of computation to account for the processing of common ancestor queries. Harel[H] has provided us with an algorithm which, after an $O(n)$ preprocessing cost, will process any number of lowest common ancestor queries for a constant cost each.

The SYMHEAP algorithm forms a symmetric heap using fewer than $2n$ comparisons, where $n = |V|$. The algorithm keeps track of the right shoulder, which is composed of the root and the subtree formed by repeatedly taking right subchildren. For simplicity's sake, we assign the value ∞ to v_0.

SYMHEAP

Arrays, Variables:

- $[\phi(i); 0 \leq i \leq n]$, the array which keeps track of the weight of each vertex.
- $[LC(i), RC(i); 0 \leq i \leq n]$, the array which keeps track of the developing heap structure. RC and LC stand for for right-child and left-child respectively.
- $[S(i); 0 \leq i \leq n]$, the shoulder array. We will use $S(i) = j$ to denote the fact that v_j is on the shoulder, i levels beneath the root, v_0.
- $SLENGTH$, an integral variable to keep track of the length of the shoulder array.
- $SSEARCH$, an integral variable which will keep track of the search along the shoulder array.
- i, an integral variable which will correspond to the vertex v_i as it is placed in the heap.

1. Initialize arrays
   (a) For $i = 1$ to n, read $\phi(i)$. [Load weights into weight array.]
   (b) $\phi(0) := +\infty$.
   (c) $S(0) := 0$. [v_0 is fixed as the root of the heap.]
   (d) $RC(0) := \emptyset; LC(0) := \emptyset$. [The heap begins with no subtrees.]
2. $SLENGTH = 0$. [The degenerate shoulder is of zero length.]
3. For $i = 1$ to n do
   (a) $SSEARCH := SLENGTH$.
   [Shoulder search begins at the bottom of the shoulder.]
   (b) while $\phi(i) > \phi(S(SSEARCH))$ do $SSEARCH := SSEARCH - 1$ REPEAT;
   [$SSEARCH$ is decremented until the proper place on the shoulder for the new vertex v_i is found]
   (c) $LC(i) := RC(S(SSEARCH))$.
   [The old right subtree of $v_{S(SSEARCH)}$ is moved to become the left subtree of v_i.]
   (d) $RC(i) := \emptyset$. [The right subtree of v_i is initialized to be empty.]
   (e) $RC(S(SSEARCH)) := i$.
   [v_i is inserted as the right subchild of $v_{S(SSEARCH)}$.]
   (f) $S(SSEARCH + 1) := i$.
   [The position of v_i on the shoulder is noted by the shoulder array.]

(g) $SLENGTH := SSEARCH + 1$.
[The new shoulder length is $SSEARCH{+}1$, as v_i is at the bottom of the shoulder.]

For each vertex v_i, the algorithm uses the variable $SSEARCH$ to find the vertex on the shoulder, $v_{S(SSEARCH)}$, minimal with respect to the heap order, which satisfies $\phi(v_{S(SSEARCH)}) \geq \phi(v_i)$. The vertex v_i is inserted as the new right child of $v_{S(SSEARCH)}$, which places it at the bottom of the shoulder, as the old right subtree of $v_{S(SSEARCH)}$ is inserted as the new left subtree of v_i.

Cost analysis: The only comparisons are done at the while loop. We can bound the number of comparisons by $2n$ by observing the following. Each vertex is added to the shoulder at most once, and therefore removed at most once. For each i, the number of comparisons used in processing v_i is equal to one more than the number of vertices this processing removes from the shoulder. As we sum over i, this yields the bound of $2n$.

Having constructed the symmetric heap with the SYMHEAP algorithm above, the algorithm then makes a call to the Harel algorithm to process all of the path queries on T in linear time. The Harel algorithm does not make any comparisons between the values of ϕ, so it does not contribute to the information-theory cost of the algorithm. Its cost is only considered with regard to implementation. Komlós only achieves linear-time implementation in the setting of strings. A variation on the Komlós algorithm, presented in [DRT], achieves linear-time implementation in all settings, but that result is not within the scope of this paper.

4.2 The Full-Branching Tree

The second extreme example is the case of a full-branching tree. Recall that a full-branching tree is a rooted tree in which all the leaves are on the same level, and all non-leaves have at least 2 children.

Recall also the formulation of the problem: we are given a set A of queries, $A \subset Q = \{p(x,z) | x, z \in V(T), x \geq_T z\}$. A query is a path oriented away from the root, v_1. We seek the maximum vertex of each query, with regard to the weight function ϕ. We use the phrase "processing a query" as shorthand for finding the maximum vertex of the query.

Here and in the full algorithm, we will make frequent use of Jensen's inequality.

Jensen's Inequality Suppose ψ is a convex real-valued function defined on the open interval (a,b), and suppose $x_i \in (a,b)$ for $i = 1, \ldots, k$. If $p_i \geq 0$ for $i = 1, \ldots k$, and $\sum_i p_i > 0$, then

$$\psi(\frac{\sum_j p_j x_j}{\sum p_j}) \leq \frac{p_j \psi(x_j)}{\sum p_j}.$$

As the logarithm function is convex over the entire positive real line, Jensen's inequality implies that for positive real values of x_i,

$$\sum_{i=1}^{k} \log x_i \leq k \log \frac{\sum_{i=1}^{k} x_k}{k} \tag{1}$$

.

Definition: A *list* is a set indexed by an ordinal. An *ordered list* is a list $S = \{s_i | i \in I\}$, with an associated rank function $r : X \to Y$, where Y is a totally ordered set, such that the rank function is monotonic with respect to the indexing, i.e. $i \leq j$ implies $r(s_i) \leq r(s_j)$.

For each $y \in T$, let $A(y)$ be the set of queries which include y, $A(y) = \{p(x, z) \in A | x \geq y \geq z\}$. Here and in what follows, we use $\geq$ as shorthand for $\geq_T$ when attention is focused upon a particular T. Let $A^*(y)$ be the set of restrictions of these paths to the interval $[root(T), y]$, $A^*(y) = \{p(x, y) | \exists z \in V(T), p(x, z) \in A(y)\}$.

We find the maxima for all sets $A^*(y)$ by starting at the root and proceeding downward. Assuming that this task has been performed to the i-th level, let us process all queries in $A^*(y)$ for y in the $(i + 1)$-st level. If $\bar{y}$ is the parent of y, we know the maxima on the restrictions to $[root(T), \bar{y}]$ of all paths in $A(y)$. The set $\{p(x, \bar{y}) | p(x, y) \in A^*(y)\}$ is an ordered list, where the rank function Φ is $\Phi(p(x, \bar{y})) = \max\{\phi(z) | z \in p(x, \bar{y})\}$, as $x_1 \geq x_2$ implies $\Phi(p(x_1, \bar{y})) \geq \Phi(p(x_2, \bar{y}))$. Thus to find the maxima over $A^*(y)$ we need only compare the value $\phi(y)$ with the values on this list. This can be done with a binary search, at a cost of at most $\lceil \log(|A(y)| + 1) \rceil$ comparisons.

Cost analysis: Let L_i denote the set of nodes on level i, and $l_i = |L_i|$. Since L_i is an antichain, the sets $A(y)$, $y \in L_i$, are pairwise disjoint. Thus, by Jensen's inequality:

$$\sum_{y \in L_i} \lceil \log(|A(y)| + 1) \rceil < \sum_{y \in L_i} (1 + \log(|A(y)| + 1)) \leq l_i + l_i \log \frac{|A| + l_i}{l_i}.$$

Let C denote the total cost of processing all queries, then

$$C < \sum_i (l_i + l_i \log \frac{|A| + l_i}{l_i}) \leq n + n \log \frac{|A| + n}{n} + \sum_i l_i \log \frac{n}{l_i}$$
$$\leq 3n + n \log \frac{|A| + n}{n}.$$

4.3 The Full Algorithm

For the general tree, we use a combination of the above approaches. We first look at its "scalps", which consist of disjoint sets of strings upon which we will use the SYMHEAP algorithm. We then look at the nodes connecting the scalps, referred to as the roots. The subtree structure generated by the partial order on the roots is similar to the full-branching tree, and the algorithm takes a similar approach to the one used by the full-branching tree algorithm.

Definition: The *scalp* of a tree T is defined as $S(T) = \{v \in V(T) | \deg(u) \leq 1 \text{ for all } u \leq v\}$, where in this context deg means outdegree.

We "scalp the tree" by setting $T_0 = T$, and inductively defining $T_{i+1} = T_i \setminus S(T_i)$, for $i = 0, 1, \ldots, k$, where k is the index representing the last non-empty T_i. This "scalping" induces a partition: $V(T) = \bigcup_{i=0}^{k} S(T_i)$.

Each vertex v not in T_i has a minimal ancestor in T_i, which we denote $R_i(v)$, the root of v in T_i. If $v \in S(T_{i-1})$, then $R_i(v)$ is the minimal root of v, which we denote by $R^*(v)$. Let R denote the set of all roots.

As with the case of the full branching tree, we wish to restrict our attention to the set A of actual queries, rather than processing all possible queries. For each i, let A_i denote the set of queries in A restricted to T_i. Thus $A_i = \{p(x,z) | \exists v \in V(T), p(x,v) \in A \text{ and } z = R_i(v)\}$. For $y \in V(T_i)$, define $A^+(y)$ to be the set of queries in A whose restriction to T_i ends at y, and define $A_i(y)$ to be $A_i(y) = \{p(x,y) | p(x,y) \in A_i\}$. Thus $A_i(y)$ is the set of queries in $A^+(y)$ restricted to T_i. Finally, for $y \in S(T_i)$, define $A^*(y) = A_i(y)$.

In the algorithm, we use the decomposition $V(T) = \bigcup_{i=0}^{k} S(T_i)$. This partitions $V(T)$ into disjoint sets of strings. We first perform SYMHEAP on each of the strings, at a total cost of less than $2n$ comparisons. We then process all of the queries in the set $\{p(x,y) | \exists p(w,z) \in A, \exists i, S(T_i) \cap p(w,z) = p(x,y)\}$, using the Harel algorithm. This set includes all of the queries which lie strictly in one component of the partition, as well as the restriction of each query to each scalp it intersects.

For $i = k$ down to 0 we find the answers for all queries in A_i. As T_k is a string, for $i = k$, the call to SYMHEAP processes all of the queries in A_k. Suppose we have processed all the queries in A_{i+1}. We proceed to process all the queries in $A_i \setminus A_{i+1}$. For a given vertex $y \in S(T_i)$ there are $|A_i(y)|$ queries in A_i that end at y. Queries contained in the interval $[y, R^*(y))$ have already been answered by SYMHEAP. For a particular query $p(x,y) \in A_i$ with $x \geq R^*(y)$ we already know a vertex $m_{xy} \in p(x, R^*(y))$ such that

$$\phi(m_{xy}) = \max_{x \geq u \geq R^*(y)} \phi(u).$$

Also, from SYMHEAP, we know a vertex $m_y \in p(R^*(y), y) \setminus \{R^*(y)\}$ such that

$$\phi(m_y) = \max_{R^*(y) > u \geq y} \phi(u).$$

The only additional comparison needed is that between $\phi(m_y)$ and $\phi(m_{xy})$. Rather than making a separate comparison for each $x \geq R^*(y)$ satisfying $p(x,y) \in A_i$, we use a binary search on the set $\Phi(y) = \{m_{xy} | x \geq R^*(y), p(x,y) \in A_i(y)\}$ to place $\phi(m_y)$ with respect to this list. $\Phi(y)$ is an ordered list, with associated rank function ϕ, since $x_1 > x_2$ implies $\phi(m_{x_1 y}) > \phi(m_{x_2 y})$. This binary search costs at most $\lceil \log(|\Phi(y)|+1) \rceil = \lceil \log(|A_i(y)|+1) \rceil = \lceil \log(|A^*(y)|+1) \rceil$ comparisons. We perform such a search for each $y \in S(T_i)$.

Cost analysis: Let C denote the total cost of the algorithm. Then

$$C \leq 2n + \sum_y \lceil \log(|A^*(y)| + 1) \rceil < 5n + n \log \frac{|A| + n}{n}.$$

The proof relies upon the following lemmas.

Lemma 4.1 *If Y is a set of vertices such that no element of Y is a root of another, then the sets $A^+(y), y \in Y$, are pairwise disjoint.*

Proof: If $p(x,z) \in A^+(y)$ and $y \in S(T_i)$, then $R_i(z) = y$. In such a case, $R_i(v) = y$ for all v in the range $y \geq v \geq z$. Thus if $y_1 \in S(T_i)$ and $y_2 \in S(T_j)$, where $i > j$, then the existence of a query $p(x,z) \in A^+(y_1) \cap A^+(y_2)$, would force $R_i(y_2) = y_1$, contradicting our assumption. □

We may apply Lemma 4.1 as follows. Suppose Y is a set of vertices satisfying the conditions of the lemma. For each $y \in Y$, we allow for $\lceil \log(|A^*(y)|+1) \rceil$ comparisons to perform the binary search of $A^*(y)$. Since the sets $A^+(y), y \in Y$, are pairwise disjoint, by Jensen's inequality

$$\sum_{y \in Y} \lceil \log(|A^*(y)|+1) \rceil < \sum_{y \in Y} [1 + \log(|A^*(y)|+1)] \leq |Y| + |Y| \log \frac{|A|+|Y|}{|Y|}. \qquad (2)$$

Letting Y be the set of non-roots, (2) implies

$$\sum_{y \notin R} \lceil \log(|A^*(y)|+1) \rceil < m + m \log \frac{|A|+m}{m}, \qquad (3)$$

where $m = n - r$ is the number of non-roots.

To evaluate the terms of the sum involving roots, we will use the following coloring of R by $\{1,2,3,\ldots\}$. For $u \in S(T_i)$ we define the *court* of u as $C(u) = \{v | R_i(v) = u\}$. Equivalently, $C(u) = \{v | u \text{ is a root of } v\}$. Color the elements of $S(T_1) \cap R$ with the color 1. Having colored the elements of $\bigcup_{1 \leq j < i} [S(T_j) \cap R]$, color the elements of $S(T_i) \cap R$ as follows: u gets the smallest color that has not been used to color any of the vertices of its court $C(u)$. Let R^i denote the set of roots of color i, R^0 denote the set of leaves of $T = T_0$, and $r^i = |R^i|$, while $r = |R|$. There is a natural partial order on R, namely $u \gg v$ iff there exists i such that $u = R_i(v)$. Each set R^i is an antichain under $\gg$, so Lemma 4.1 may be applied to each. We will show that the sizes of the sets R^i exhibit an exponential decay, so that as we sum over all i, (2) will give a good bound.

Lemma 4.2 *For any fixed $i \geq 1$, the sets $A^+(y), y \in R^i$ are disjoint.*

Proof: Each set R^i is an antichain under the partial order $\gg$, so this follows immediately from Lemma 4.1. □

Lemma 4.3 *For all values of i in our range, $R^i \subset T_i$.*

Proof: We use induction on i. The statement is immediate for $i = 0, 1$. Suppose the lemma is true at the $(i-1)$ level. Let $r \in R^i$. Then there exists r' such that $r' \in C(r)$, and r' is colored by $i-1$. By induction, $r' \in T_{i-1}$. Since r is a root of an element of T_{i-1}, it must be the case that $r \in T_i$. □

Lemma 4.4 *Leaves of $T_i (i \geq 1)$ have color i.*

Proof: We induct on i. For $i = 1$, the statement is trivial. Assuming validity for $i-1$, consider leaves of T_i. If u is a leaf of T_i, it has a child in T_{i-1}, so there exists a leaf v of T_{i-1} such that $R_i(v) = u$. By induction, the color of v is $i-1$, which means that we have used all of the colors $\{1, \ldots, i-2\}$ in coloring the elements of $C(v)$. Since u is a root of v, $C(v) \subset C(u)$, so the color of u is at least i. On the other hand, since $u \in S(T_i)$, its color cannot exceed i by Lemma 4.3. Thus u is of color i. □

Lemma 4.5 *Every node $u \in \bigcup_{j>i} R^j$ must have at least two children in T_i.*

Proof: If $u \in \bigcup_{j>i} R^j$, then there exists $v \in R^i$ such that $v \in C(u)$. Since $R^i \subset T_i$ by Lemma 4.2, v is a vertex of T_i having u as a root, which implies that the degree of u in T_i must be at least 2. □

Lemma 4.6 *For $i \geq 0$, we have the exponential decay*

$$\sum_{j>i} r^j < r^i, \tag{4}$$

whence $r < n/2$, and for $i = 1, 2, \ldots$

$$\sum_{j>i} r^j < r2^{-i}. \tag{5}$$

Consequently, we have

$$\sum_{i\geq 1} (r^i/r)\log(r/r^i) < \sum_{i\geq 1} i2^{-i} = 2.$$

Proof: In any rooted tree, the number of vertices of degree at least two is less than the number of leaves. Each root counted in the sum $\sum_{j>i} r^j$ has at least two children in T_i, by Lemma 4.5. Thus this sum is less than the number of leaves in T_i, which is less than or equal to r^i by Lemma 4.4. Thus $\sum_{j>i} r^j < r^i$. Similarly, each root has at least two children in T_0, so $r < n/2$.

The inequality $\sum_{j>i} r^j < r2^{-i}$ is shown using induction on i. For $i = 0$, it follows immediately from (3). For $i \geq 1$, we have $\sum_{j>i-1} r^j < r2^{-i+1}$ by induction, and $\sum_{j>i} r^j < r^i$ by (3), so since r^i is at least equal to half of $\sum_{j>i-1} r^j$, the bound on $\sum_{j>i} r^j$ follows.

To prove the final inequality,

$$\sum_{i\geq 1} (r^i/r)\log(r/r^i) < \sum_{i\geq 1} i2^{-i} \tag{6}$$

we consider the entropy function $H : D \to \Re$, defined as

$$H(p_1, p_2, \ldots, p_k) = \sum_{i=1}^{k} p_i \log(1/p_i),$$

on the domain

$$D = \{(v_1, v_2, \ldots, v_k) \in [0,1]^k | \sum v_i = 1, v_i \geq \sum_{j>i} v_j, 1 \leq i < k\}.$$

The domain D is a closed, bounded subset of R^k, so the continuous function H achieves a maximum on D. Let $\mathbf{q_0} = (q_1, q_2, \ldots, q_k) \in D$ be the point where equality is achieved in the constraints $\sum_{j>i} q_j \geq q_i$, i.e. $q_i = 2^{-i}$ for $i < k$, and $q_k = q_{k-1}$.

Claim 4.3 *H achieves its maximum at $\mathbf{q_0}$.*

Proof of Claim: Let $\mathbf{p_0} = (p_1, \ldots, p_k)$ satisfy $\sum_{j=1}^k p_j = 1$, $\sum_{j>i} p_j \leq p_i$ for $i = 1, \ldots k-1$. By a classic result of Hardy, Littlewood, and Pólya[HLP], since $h(x) = x \log(1/x)$ is a concave function on $(0,1)$, $\sum_{j\leq i} q_j \leq \sum_{j\leq i} p_i$ for $i = 1, \ldots, k-1$, and $\sum q_i = \sum p_i$, we observe the inequality $\sum_i h(p_i) \leq \sum_i h(q_i)$. □

We evaluate H at $\mathbf{p_0}$ to get the desired bound.

$$\max_{\mathbf{p}\in D} H(\mathbf{p}) = H(\mathbf{p_0}) = \sum_{i<k} i2^{-i} + k2^{-(k-1)}$$

$$= \sum_{i<k} i2^{-i} + \sum_{i\geq k} k2^{-i} < \sum_{i<k} i2^{-i} + \sum_{i\geq k} i2^{-i} = \sum_{i\geq 1} i2^{-i} = 2.$$

□

Proof of Cost Analysis: The cost of our algorithm includes at most $2n$ comparisons for the call to SYMHEAP, and $\lceil \log(|A^*(y)| + 1) \rceil$ comparisons for each $y \in T$.

Recall the claim was

$$C \leq 2n + \sum_y \lceil \log(|A^*(y)| + 1) \rceil < 5n + n \log \frac{|A| + n}{n}.$$

For the terms involving non-roots, recall the inequality (3)

$$\sum_{y \notin R} \lceil \log(|A^*(y)| + 1) \rceil < m + m \log \frac{|A| + m}{m},$$

where $m = n - r$. The remainder of the sum may be expressed as

$$C^1 = \sum_{i=1}^{k} \sum_{y \in R^i} \lceil \log(|A^*(y)| + 1) \rceil.$$

By Lemma 4.2, for fixed i the sets $\{A^+(y) | y \in R^i\}$ are pairwise disjoint, so we can use the following estimation for the inner sums

$$\sum_{y\in R^i} \lceil \log(|A^*(y)| + 1) \rceil \leq \sum_{y \in R^i} (1 + \log(|A^*(y)| + 1)) \leq r^i + r^i \log \frac{|A| + r^i}{r^i},$$

where the latter inequality follows from Jensen's inequality. With these bounds on the inner sums, we get the bound

$$C^1 < \sum_{i=1}^{k}(r^i + r^i \log \frac{|A| + r^i}{r^i}).$$

Using the exponential decay from Lemma 4.6, we have

$$C^1 < r + r\log \frac{|A|+r}{r} + \sum_{i\geq 1} r^i \log \frac{r}{r^i} < 3r + r\log\frac{|A|+r}{r}.$$

Using the fact that $m + r = n$, and the log-sum inequality

$$x\log\frac{a}{x} + y\log by \leq (x+y)\log\frac{a+b}{x+y},$$

we can bound the total cost by

$$\begin{aligned} C &< 2n + m + 3r + m\log\frac{|A|+m}{m} + r\log\frac{|A|+r}{r} \\ &< 5n + n\log\frac{|A|+n}{n}. \end{aligned}$$

□

5 The Rank-Sequence Algorithm

Bar-Noy, Motwani, and Naor use a deterministic algorithm to solve the set-maxima problem in two settings, both having $m = n$ (the most interesting case of the set-maxima problem). In the first setting, the subsets whose maxima are to be found are the hyperplanes of the projective geometry $PG(d, q)$. Of particular interest is the case when $d = 2$, when the intersection of any two subsets is exactly one point. The lack of large intersections is a contributing factor in the solution when $d = 2$. For all values of d, the strong expansion properties of the projective geometries motivate our result. Namely, we use the fact that a set Y of points in $PG(d, q)$ misses at most $O(\frac{nq}{|Y|})$ hyperplanes (see Lemma 5.1).

The second case is that of random sets. In this case, we are given a parameter $p = p(n)$ such that $\mathrm{Prob}(x_i \in S_j) = p$ for all i, j, with the events $\{x_i \in S_j\}$ mutually independent. With random sets, there is a large probability that computing a simple rank sequence will produce a linear-time solution. We show that, as $n \to \infty$, the expected number of comparisons used is $O(n)$, and that the probability of using more comparisons than expected is asymptotically negligible.

We will use the following terminology: W_x is the set of elements of X greater than or equal to x. Thus, as earlier, the *rank* of an element x is $r(x) = |W_x|$. A *rank sequence* is a monotonically decreasing sequence of ranks $r_1 \geq r_2 \geq \ldots \geq r_k$.

A single approach is used for both cases under discussion. We first choose an appropriate rank sequence $\vec{r} = (r_i)$ The SELECT algorithm is used to find the elements $Z = \{z_i | 1 \leq i \leq k\}$ satisfying $r(z_i) = r_i$. Note this also automatically yields a partition of X into intervals:

1. $I_0 = \{x | x \leq z_1\}$,
2. For $1 \leq i \leq k-1$, $I_i = \{x | z_i < x \leq z_{i+1}\}$,
3. $I_k = \{x | z_k < x\}$.

We shall refer to Z as the "flag sequence associated to R", and refer to this stage of the algorithm as "computing the flag sequence." (This is not standard terminology.)

For $i = 1, \ldots, n$ one forms the reduced set $S'_i = S_i \cap I_{l(i)}$, where $l(i)$ is the greatest value of l satisfying $S_i \cap I_l \neq \emptyset$. Then $\max(S'_i) = \max(S_i)$ and the algorithm concludes by using a brute force method to find the maximum element of each S'_i, at a cost of $\sum_{i=1}^{n}(|S'_i| - 1)$.

There are two costs associated with the rank-sequence algorithm. First, there is the cost of using SELECT to find Z and the partition of X into the intervals $\{I_i\}$, which we will denote $f_{\vec{r}}(n)$. Second is the cost $\sum_{i=1}^{n}(|S'_i| - 1)$ of finding the maxima of the sets $\{S'_i\}$. In both cases considered in [BMN], we can simultaneously get $O(n)$ bounds for each of these costs, because of the "expansion" properties of the set systems $\{S_i\}$.

The first cost is easily bounded for any finite-entropy rank-sequence. The *entropy* of a rank-sequnce $\vec{r} = (r_i)_{i=1}^{k}$ is

$$ent(\vec{r}) = -\sum_{i=1}^{k+1} \frac{r_{i-1} - r_i}{n} \log \frac{r_{i-1} - r_i}{n},$$

where $r_0 = n$ and $r_{k+1} = 0$. The following theorem gives a bound for the cost of computing the rank sequence, based on its entropy.

Theorem 5.1 *For any given sequence $\vec{r}$, the comparison complexity of computing the flag sequence Z associated to $\vec{r}$ is $f_{\vec{r}}(n) = \Theta(n(1 + ent(\vec{r})))$.*

Proof: The lower bound is the information-theory bound, obtained by a leaf-counting argument. As for the upper bound, it is obtained by recursion. First, choose j such that $r_j \leq \frac{n}{2} \leq r_{j-1}$, and use SELECT to find z_j and z_{j-1} in linear time. Let $X^- = \{x | x < x_{j-1}\}$ and $X^+ = \{x | x > x_j\}$ and use recursion on X^- and X^+. □

5.1 Case 1: Hyperplanes of the Projective Geometry

Projective geometries are well-known set-systems. We shall define the projective geometry $PG(d, q)$ for $d \geq 2$, and also define projective planes, which may be thought of as generalizations of $PG(2, q)$. For more information about projective geometries, the interested reader is referred to a good text on geometries, for example [Ba].

For any prime power q, and $d \geq 2$, let $V_{d+1}(q)$ be the $(d+1)$-dimensional vector space over the finite field $GF(q)$ with q elements. We define $PG(d, q)$ to be the pair (X, S) where X is the set of 1-dimensional subspaces of $V_{d+1}(q)$, and S is the set of d-dimensional subspaces of $V_{d+1}(q)$. For our purposes, we consider $x \in h$, for $x \in X, h \in S$, if x is a subspace of h.

The geometry $PG(2, q)$ is an example of a *projective plane.* A projective plane is defined as a pair (X, S), where $S \subset 2^X$, which satisfies the axioms:

1. For any $x, y \in X$, there exists a unique $l \in S$ such that $\{x, y\} \subset l$.

2. For any $l, m \in S$, there exists a unique $x \in X$ such that $x \in l \cap m$.

3. There exist $x_1, x_2, x_3, x_4 \in X$ such that for all $l \in S$, $\{x_1, x_2, x_3, x_4\} \not\subseteq l$.

In this case, elements of S are referred to as *lines*. (Similar axioms exist for $d > 2$, but a classic theorem says that if $d > 2$, only the geometries $PG(d, q)$ satisfy the axioms.)

We will solve the set-maxima problem for all of the projective geometries, (X, S). We shall consider the projective planes separately before tackling the geometries $PG(d, q)$, for $d > 2$. Both cases will use the following result:

Lemma 5.1 *Let Y be a non-empty set of points in $PG(d, q)$. Then Y misses at most $O(\frac{nq}{|Y|})$ hyperplanes.*

The interested reader is referred to [Al] for the proof of this lemma.

We first consider projective planes. Here $n = |X| = q^2 + q + 1$. Let $S = \{S_1, S_2, \ldots, S_n\}$ be an ordering of the lines. We fix ϵ satisfying $0 < \epsilon < 1$, and use the rank sequence $\vec{r} = (r_i)$, with $r_i = \lfloor \frac{n}{b_i} \rfloor$ for $2 \leq i \leq k$ where $b_i = i^{1-\epsilon}$, and $k = \lceil n^{\frac{1}{1-\epsilon}} \rceil$. Note that we are deviating slightly from our earlier convention, beginning with r_2, so our first interval is I_1.

The cost of computing the flag sequence associated to $\vec{r}$ is $O(n)$, by Theorem 5.1. We also can demonstrate the bound on this cost directly.

For all n, let $f_R(n)$ denote the cost of computing the above sequence on a set of size n. To compute the flag sequence on X, we first use SELECT on X to find the element z_2 whose rank is $r_2 = \lfloor \frac{n}{2^{1-\epsilon}} \rfloor$. We then compute the sequence associated to $\vec{r}'$, where $r'_i = \lfloor \frac{r_2}{b_i} \rfloor$ for $i = 2, \ldots, k'$ on the set W_{z_2}, and $k' = \lceil r_2^{1-\epsilon} \rceil$, yielding the elements z'_i such that $r(z'_i) = r'_i$. Since

$$\lfloor \frac{n}{(2i)^{1-\epsilon}} \rfloor - 1 \leq \lfloor \frac{\lfloor n/2^{1-\epsilon} \rfloor}{i^{1-\epsilon}} \rfloor \leq \lfloor \frac{n}{(2i)^{1-\epsilon}} \rfloor,$$

for each i, either $z'_i = z_{2i}$, or z'_i is the element whose rank is $r_{2i} + 1$, in which case, z_{2i} is the minimum among elements in $(z'_i \ldots z'_{i+1}]$. Thus we can find all of the elements of the flag sequence with ranks of the form r_{2i} with at most n comparisons beyond the cost of computing the sequence $vecr'$.

For the z_i where i is odd, let $i = 2l + 1$. Then we need only apply SELECT to the interval $[z_{2l}, z_{2(l+1)}]$, the cost of which is $O(r_{2l} - r_{2(l+1)})$, so the remaining elements of the flag sequence may be computed with a cost of $O(n)$ comparisons. Thus our cost function, $f_R(n)$ satisfies the recursive equation

$$f_R(n) = f_R(\frac{n}{2^{1-\epsilon}}) + O(n),$$

and therefore $f_R(n) = O(n)$.

Having divided X into intervals I_i, one forms the reduced sets $S'_i = S_i \cap I_{z(i)}$, where $z(i)$ is the greatest value of z for which the set $S_i \cap I_z$ is non-empty. The second cost of the rank-sequence algorithm is $\sum_{i=1}^{n}(|S_i'|-1)$. To show this cost is $O(n)$, we rely upon the following lemma.

Let $V = \{v_1, \ldots, v_s\}$ be a subset of the points of a projective plane, and let $W = \{w_1, \ldots, w_t\}$ be a subset of the lines. Define the $t \times s$ incidence matrix M by $M_{ji} = 1$ if $v_i \in w_j$, otherwise $M_{ji} = 0$. Then define $c_i = \sum_{j=1}^{t} M_{ji}$, the i-th column sum. The L_1-*norm* of M is $||M||_1 = \sum_{i=1}^{s} c_i$, the sum of the entries of M.

Lemma 5.2 *With the above definitions,*

$$\sum_{i=1}^{s} c_i \leq t\sqrt{s} + s.$$

Proof: First, assume that for all i, $c_i \geq 2$. Then

$$\binom{c_1}{2} + \binom{c_2}{2} +, \ldots, + \binom{c_s}{2} \leq \binom{t}{2} \tag{7}$$

The right-hand side of (6) is the number of unordered pairs of lines from the set M. Each $\binom{c_i}{2}$ counts the pairs of lines which intersect at v_i. Since each pair of lines intersects in at most one of the v_i's, the left-hand side sum counts no pair of lines twice, which implies the inequality.

Let $C = \sum_{i=1}^{s} c_i$. The left-hand side of (6) is no less than the value when $c_i = C/s$, since the function $\binom{x}{2} = \frac{x(x-1)}{2}$ is convex over all of $\Re$. Thus,

$$\frac{s}{2}(\frac{C}{s} - 1)^2 \leq \sum_{i=1}^{s} \binom{C/s}{2} \leq \sum_{i=1}^{s} \binom{c_i}{2} \leq \binom{t}{2} \leq \frac{t^2}{2},$$

which yields $C \leq t\sqrt{s} + s$ when rearranged.

We made the assumption that for all i, $c_i \geq 2$. Extending this to the more general case is a trivial matter. Let S' denote the set of indices i for which $c_i < 2$, and let $s' = |S'|$. Then

$$\sum_i c_i = \sum_{i \in S'} c_i + \sum_{i \notin S'} c_i \leq s' + (t\sqrt{s-s'} + (s - s')) \leq s + t\sqrt{s}.$$

This proves the lemma. □

With Lemma 5.2, we bound the L_1 norm of the t by s incidence matrix by $t\sqrt{s} + s$, a significant improvement over the trivial st bound. Let us look at what effect this has upon the number of comparisons the algorithm uses.

Let $s_i = |I_i|, L_i = \{l \in S | l \cap I_i \neq \emptyset, l \cap (\bigcup_{i<j\leq k} I_j) = \emptyset\}, t_i = |L_i|$ and let M_i be the $t_i \times s_i$ incidence matrix of $L_i \times I_i$.

The second cost of the rank sequence algorithm may be expressed as either $\sum_{j=1}^{n}(|S_j'|-1)$ or as $(\sum_{i=1}^{k} ||M_i||_1) - n$. By Lemma 5.2, $||M_i||_1 \leq t_i\sqrt{s_i} + s_i$. Since $\sum_{i=1}^{k} s_i = n$, to show a linear bound for this cost, we need only show that $\sum_{i=1}^{k} t_i\sqrt{s_i} = O(n)$.

Recall that

$$s_i = \lfloor \frac{n}{i^{1-\epsilon}} \rfloor - \lfloor \frac{n}{(i+1)^{1-\epsilon}} \rfloor = O(n \frac{(i+1)^{1-\epsilon} - i^{1-\epsilon}}{((i+1)i)^{1-\epsilon}}).$$

The geometric expansion properties of projective geometries are crucial in bounding the cost in question. By Lemma 5.1, if $|Y| = \lfloor \frac{n}{i^{1-\epsilon}} \rfloor$, then Y misses at most $O(qi^{1-\epsilon})$ lines. Since the sequence of s_i's is decreasing, the sum $\sum_{i=0}^{k} t_i\sqrt{s_i}$ is maximized when each set W_{z_i} misses the maximum possible number of lines. Thus, we may assume $t_i = O(q((i+1)^{1-\epsilon} - i^{1-\epsilon}))$, for $i > 1$, while $t_1 = O(q)$.

By the mean value theorem,

$$(i+1)^{1-\epsilon} - i^{1-\epsilon} = (1-\epsilon)(i+\eta)^{-\epsilon} = O(i^{-\epsilon}),$$

for some $0 \le \eta \le 1$, for $i > 1$.

Thus $s_i = O(ni^{-2+\epsilon})$, $t_i = O(qi^{-\epsilon})$, and

$$t_i\sqrt{s_i} = O(ni^{-1-(\epsilon/2)}).$$

As we sum over the range of i, we get

$$\sum_{i=1}^{k} t_i\sqrt{s_i} = O(n\sum_{i=1}^{k} \frac{1}{i^{1+\epsilon/2}}) = O(n).$$

The cost of the algorithm is thus seen to be $O(n)$ comparisons. □

For the case $d > 2$, the algorithm is even simpler. Here the underlying set is the set of points in a projective geometry, again denoted X, and the collection of subsets whose maxima are to be found is the set of hyperplanes, denoted H. The rank sequence used is $r_i = \lfloor \frac{n}{i} \rfloor$. By Theorem 5.1 we may compute this sequence in linear time.

The bound on the second cost of the algorithm is given by the following lemma.

Let $\{v_1, \ldots, v_s\} \subset X$, and $\{m_1, \ldots, m_t\} \subset H$. Define the $t \times s$ incidence matrix M as usual, M_{ji} is 1 if $v_i \in m_j$, and 0 otherwise.

Lemma 5.3 *For $1 \le i \le s$, let $c_i = \sum_{j=1}^{t} M_{ji}$. Then*

$$\sum_{i=1}^{s} c_i = O(dts^{\frac{d-1}{d}} + ds).$$

The proof of this lemma is similar to the proof of the analagous lemma we used for the case $d = 2$. Here, we would use the inequality

$$\binom{c_1}{d} + \binom{c_2}{d} +, \ldots, + \binom{c_s}{d} \le \binom{t}{d}.$$

The proof is omitted in the interests of brevity. □

Returning to the cost analysis, $s_i \le \frac{n}{i^2}$ and by Lemma 5.1, $it_i \le n^{\frac{1}{d}}$. Therefore

$$dt_i s_i^{\frac{d-1}{d}} \le dni^{-\frac{2(d-1)}{d}}.$$

As $d > 2$ is fixed, the series $\sum_{i=1}^{\infty} di^{-\frac{2(d-1)}{d}}$ is convergent, which implies

$$\sum_{i=0}^{k}(dt_i s_i^{\frac{d-1}{d}} + ds_i) = O(n),$$

which is equivalent to

$$\sum_{j=1}^{n} |S_j'| = O(n).$$

In conclusion, both costs associated with the rank-sequence algorithm have been shown to be linear for this case. □

5.2 Case 2: Random Sets

Bar-Noy, Motwani, and Naor also achieved results with a rank-sequence approach in the setting of random sets. Here we again take the number of sets, m, to equal n, the cardinality of the underlying set, X. By "random sets," we mean the following.

We are given a parameter $p = p(n)$, and the ordered set $X = \{x_1, \ldots, x_n\}$, where $i < j$ implies $x_i > x_j$. We create the family of sets $S = \{S_1, S_2, \ldots, S_n\}$ by including x_i in S_j with probability p, where the n^2 choices are made independently.

As before, the algorithm is based on using an appropriate rank sequence. The rank sequence we use is simple: we take $r_i = \lceil \frac{n}{2^i} \rceil$, and $k = \log n$. Then z_1 is the median element of X, and for $i > 1$, z_i is the median element of $W_{z_{i-1}}$. By Theorem 5.1, computing this sequence may be done at a cost of $O(n)$. (In this case it is easy to see such a bound: if we first calculate z_1, then z_2, and so forth, then, since the cost of calculating z_i is $O(r_i)$, the cost of computing the whole sequence is $O(r_1) + O(r_2) +, \ldots, +O(r_k) = O(n)$.

Thus the first of the two costs associated with this algorithm is bounded by a linear number of comparisons. It remains to show that the second cost, $\sum_{i=1}^{n}(|S_i'| - 1)$, is linear in n with high asymptotic probability. To do so, we compare the random variables $C_i := |S_i'| - 1, i = 1, \ldots, n$ with independent random variables $\hat{C}_i$, where for all i, $\hat{C}_i$ will stochastically dominate C_i.

Definition: Given two real-valued random variables, X_1 and X_2, we say that X_1 *stochastically dominates* X_2 if for all real values α, $\text{Prob}(X_1 > \alpha) \ge \text{Prob}(X_2 > \alpha)$.

Let us fix i, and consider how C_i varies as a function of the n independent choices determining S_i. To simplify the use of the rank sequence, let us suppose that n is a power of 2. Let $m(i)$ denote the number of elements of X greater than the maximal element of S_i. Let $\hat{C}_i$ be the sum of $m(i)$ i.i.d. random variables, each taking the value 1 with probability p, and 0 with probability $1 - p$. Note that $\hat{C}_i$ depends upon $m(i)$, and thus on the composition of S_i, so C_i and $\hat{C}_i$ are dependent variables.

Claim 5.1 *$\hat{C}_i$ stochastically dominates C_i.*

Let $l(i)$ denote the least l satisfying $r_l \leq m(i)$. Then $|S_i'| = |I_{l(i)-1} \cap S_i|$. Thus $|S_i'| - 1$ is equal to the number of elements of $I_{l(i)}$ which are in S_i, but are not the maximum of S_i. Recall that the length of $I_{l(i)-1}$ is equal to $r_{l(i)}$, by choice of the rank sequence.

We may regard choosing the random set S_i as making n coin flips in order, and including x_j in S_i iff the j^{th} coin comes up heads, assuming the coins land heads up with a probability of p. From this standpoint, the variable $m(i)$ represents the number of tails which appear before the first head. The value C_i represents the number of heads which appear after the first head, but before reaching the top of the interval $I_{l(i)-2}$. There are thus $r_{l(i)-1}-(m(i)+1)$ relevant coin flips after the first head appears. Since $r_{l(i)-1} - (m(i)+1) \leq m(i)$, the random variable $\hat{C}_i$ stochastically dominates the random variable C_i. □

Let $C = \sum_{i=1}^n (|S_i'| - 1) = \sum_{i=1}^n C_i$. Then C is stochastically dominated by the random variable $\hat{C} = \sum_{i=1}^n \hat{C}_i$, since each term C_i is stochastically dominated by the corresponding term $\hat{C}_i$. Since each $\hat{C}_i$ is independent, with binomial distribution $B(m(i), p)$, the sum $\hat{C}$ is also binomially distributed as $B(M, p)$, where $M = \sum_{i=1}^n m(i)$. (So M is itself a random variable.)

Each variable $m(i)$ is geometric with parameter p; that is, $Pr[m(i) = d] = q^d p$, where $q = 1 - p$. Thus $E[m(i)] = q/p$, so $E[M] = nq/p$, and $E[\hat{C}] = n(q/p)p = nq$, as each of the M i.i.d. variables has mean p.

Each random variable $m(i)$ has the probability generating function $\frac{p}{1-qz}$, so the p.g.f. for M, the sum of the variables $m(i)$, is the product of their respective generating functions, i.e. the generating function for M is $g(z) = (\frac{p}{1-qz})^n$.

The variable $\hat{C}$ is defined as the sum of M i.i.d. variables, each of which has the p.g.f. $b(z) = q + pz$. Thus, the p.g.f. of $\hat{C}$ is the composition of $g(z)$ and $b(z)$, let us call it $h(z)$. So $h(z) = g(b(z)) = (\frac{p}{1-q(q+pz)})^n = (1 + q(1-z))^{-n}$.

For tail probabilities, the following is an easy consequence of Markov's inequality.

Theorem 5.2 *Let X be a positive integer-valued random variable with the probability generating function $F(z)$. Then, for any $z \geq 1$, $Prob[X \geq \lambda] \leq \frac{F(z)}{z^\lambda}$*

We use this inequality as follows. For $\alpha > 1$ a positive constant, let $\lambda = \alpha n q$ and $z = 1 + \frac{1}{2q}$. Then

$$Pr[\hat{C} \geq \alpha n q] \leq h(1 + \frac{1}{2q})(1 + \frac{1}{2q})^{-\alpha n q} = 2^n (1 + \frac{1}{2q})^{-\alpha n q}$$

$$\leq 2^n e^{(-\alpha n)/2} = (\frac{2}{e^{\alpha/2}})^n.$$

If we choose $\alpha > \ln 4$, this quantity will converge to zero at an exponential rate.

As $\hat{C}$ stochastically dominates C, this implies that $Pr[C \geq \alpha n q]$ will also decrease exponentially as n increases linearly. In particular, we have $E[C] = O(n)$.

5.3 Worst-case behavior

The strength of the rank-sequence algorithm in tackling set-maxima for random sets might lead us to hope that this algorithm could see reasonable results in any setting. But a quick examination of worst-case behavior squashes such a hope.

For a rank sequence $\vec{r}$ over a set with n elements, define $m(\vec{r}) = (1/n) * \max_i(r^i - r^{i+1})$.

Lemma 5.4 *For any rank sequence $\vec{r}$, $ent(\vec{r}) \geq -\log m(\vec{r})$.*

This is immediate from the definition of the entropy function. Let $c(\vec{r}, n)$ be the maximum cost of the second step of the rank sequence algorithm on a set with n elements, using rank sequence $\vec{r}$.

Lemma 5.5 *As defined above, $c(\vec{r}, n) = n * (nm(\vec{r}) - 1)$.*

It is clear that, for a given rank sequence $\vec{r} = (r^i)$, the worst-case behavior under the rank-sequence algorithm is that all of the sets equal the largest open interval, whose length is $\max_i(r^i - r^{i+1} - 1)$.

Theorem 5.3 *Let $(\vec{r}_n)$ be a sequence of rank sequences, with $\vec{r}_n = (r_n^1, r_n^2, \ldots, r_n^{l(n)})$. Let $m_n = m(\vec{r}_n)$, as defined above. Let w_n denote the maximum total cost of using the rank sequence $\vec{r}_n$ on a set with n elements. Then*

$$w_n = \Omega(n * (-\log m_n)m + n * (nm_n - 1)).$$

Proof: There are two costs to the rank-sequence algorithm, the cost of computing the flag sequence, and the cost of finding the maxima of each set. By Theorem 5.1 the first cost is $n * ent(\vec{r}_n)$ which by Lemma 5.4 is at least $n * (-\log m_n)$. By Lemma 5.5 the second cost, $c(\vec{r}_n, n)$, can be as much as $n * (nm_n - 1)$. This sum of the two costs is our lower bound. □

We find the lower bound for w_n by finding the value of m_n which will minimize the sum of two costs. This happens with $m_n = \frac{\log n}{n}$, which gives us $w_n = \Omega(n \log n)$.

Theorem 5.4 *In the worst case, any rank-sequence algorithm will require $\Omega(n \log n)$ comparisons.*

The worst-case behavior is when the rank-sequence step has provided absolutely no sorting information for our family of sets, each of which is as large as possible. Thus, in the worst-case analysis, the rank-sequence algorithm provides little advantage over sorting the underlying set. Clearly, if one were to seek to establish a good deterministic rank-sequence based algorithm to work in all settings, one would need to identify problem intervals by some method to remove the worst-case scenario.

6 Randomized Algorithms

Goddard, King, and Schulman presented randomized algorithms to solve the local-sorting and set-maxima problems, with optimal expected running times. Another problem they solve is the "set-sort" problem, which is closely related to the local-sorting problem.

The Set-Sort Problem: Given a set, X, and a family $S = \{S_1, \ldots, S_m\}$ of subsets of X, fully sort each subset S_i.

The set-sort algorithm creates a graph, G, with X as its vertex set, and the edge set $E = \{xy | \exists i, \{x, y\} \subset S_i\}$. This graph contains a clique corresponding to each of the sets in S, so locally sorting G fully sorts each of the sets of S. Since the set-maxima algorithm makes a call to the set-sort algorithm, which in turn makes a call to the local-sorting algorithm, we present the local-sorting algorithm first.

If one assumes a good degree of uniformity among the degrees of a graph, the limited-degree algorithm which follows will give good results. If the graph in question has a more varied degree sequence, we use a divide-and-conquer method, dividing the graph into subgraphs, each of which is of relatively uniform degree, and applying the limited-degree algorithm to each of them.

6.1 The Limited-Degree Algorithm

The limited-degree algorithm works by taking a series of nested samples from the vertex set, $V(G) = R_0 \supset R_1 \supset \ldots \supset R_k$, where $|R_k| = O(1)$. We fully sort R_k, and then locally sort each sample R_i to radius 2^{2i}. To "locally sort a subset $H \subset V(G)$ to radius r" is to know the relative order of v and w, for all $v, w \in H$ of distance at most r in G. The algorithm is called "limited-degree" because its expected running time is $O(n \log \Delta)$, where Δ is the maximal degree of the graph. If Δ is of the order of magnitude of the average degree of the graph, this algorithm will be of optimal speed. When Δ is much larger than the average degree, the $O(n \log \Delta)$ bound will be higher than desired.

Leaving the probabilities p_i to be determined later, we select the samples as follows: start with $R_0 = X$. Inductively choose the sample R_i from R_{i-1} by taking elements independently, with probability p_i/p_{i-1}. Then $E[|R_i|] = np_i$. The p_i's will be chosen so that $E[|R_k|] = O(1)$. (Note: we could start by selecting R_k first and moving in the opposite direction. The direction of our sample selection is not important as long as $E|R_i| = np_i$ for all i.)

The algorithm works by using the information gained from the local sort of R_i to radius 2^{2i} to locally sort R_{i-1} to radius $2^{2(i-1)}$, for $i = k, \ldots, 1$. Let $d(\cdot, \cdot)$ be the standard distance function on a graph, and for all $v \in V(G)$ and all $r \in \Re^+$, let $B_v^r = \{x \in V_G | d(x, v) \leq r\}$. To begin, we sort R_k at a constant expected cost. Assuming that R_i has been sorted to radius 2^{2i}:

1. For each $x \in R_{i-1}$, we find the rank of x with respect to the set $R_i \cap B_x^{2^{2i-1}}$ using a binary search (this set is already sorted by induction). This search requires at most $\log \lceil |R_i \cap B_x^{2^{2i-1}}| \rceil$ comparisons.

2. For each $x \in R_{i-1}$, define the set

$$C_x = \{y \in R_{i-1} \cap B_x^{2^{2i-2}} \mid \nexists z \in R_i \cap B_x^{2^{2i-2}} (x < z < y) \vee (y < z < x)\}.$$

   The elements of C_x are of the same rank as x with regard to the set $R_i \cap B_x^{2^{2i-2}}$. (Note that this step requires no new comparisons.)

3. If $x \in C_y$ and $y \in C_x$, compare x, y.

The set $R_i \cap B_x^{2^{2i-2}}$ can be thought of as a set of local signposts, which we use to avoid making unnecessary comparisons: if $z \in R_{i-1} \cap B_x^{2^{2i-2}}$, but there exists $y \in R_i \cap B_x^{2^{2i-2}}$ satisfying either $x \leq y \leq z$ or $x \geq y \geq z$, then there is no need to compare x and z.

Cost analysis: Let Δ denote the maximum degree of G. Then for all vertices x, and for all values of r, $|B_x^r| \leq (\Delta+1)^r$. Thus the number of comparsions used in step (1) for a fixed vertex x is at most $\lceil \log |B_x^{2^{2i-1}}| \rceil \leq 2^{2i-1} \log(\Delta+1)$. As we consider all x in $R_{i-1} - R_i$, the expected total cost of Step 1 is bounded above by $np_{i-1}2^{2i-1}\log(\Delta+1)$.

Step 2 involves making no new comparisons, and does not contribute to the cost of the algorithm.

The cost of Step 3 is at most $\frac{1}{2}\sum_{x \in R_{i-1}} |C_x|$. We use the following bound on $E[|C_x|]$.

Lemma 6.1 *For any i, and any $x \in R_{i-1}$, $E[C_x] \leq 2p_{i-1}/p_i$.*

Proof: Let $B^+ = \{z \in R_{i-1} \cap B_x^{2^{2i-2}} | z \geq x\}$. Let z^+ be the smallest element of $B^+ \cap R_i$. Then $B^+ \cap C_x$ consists of those elements of B^+ less than z^+. If we start at x and continue upwards through B^+, testing membership in R_i, $|B^+ \cap C_x|$ represents the number of members of B^+ not in R_i seen before the first member of R_i. Thus $|B^+ \cap C_x|$ is a geometric random variable. Each element of B^+ is in R_i with probability $p = p_i/p_{i-1}$. If B^+ were infinite, $|B^+ \cap C_x| + 1$ would have a geometric distibution with parameter p and thus expected value $1/p$. The finiteness of B^+ only reduces this value, so $E[|B^+ \cap C_x|] < (1-p)/p$.

Now define B^- and z^- analogously. An identical argument shows that $E[|B^- \cap C_x|] < (1-p)/p$. Since $C_x = x \cup (B^+ \cap C_x) \cup (B^- \cap C_x)$, it follows that $E[|C_x|] < 1 + 2(1-p)/p < 2/p$. □

We return to the cost of Step 3, $1/2\sum_{x \in R_{i-1}} |C_x|$. Since the expected value of each term is bounded by $2p_{i-1}/p_i$, and the expected number of terms in the sum is np_{i-1}, the expected value of this sum is bounded by np_{i-1}^2/p_i.

At this juncture we assign values to the probabilities p_i to equalize the cost bounds from steps (1) and (3). This is accomplished by taking $p_i = \frac{1}{(\log(\Delta+1))^i 2^{i^2}}$. For these values of p_i, the expected total cost of the transition from R_i to R_{i-1} is bounded by

$$np_{i-1}2^{2i-1}\log(\Delta+1) + np_{i-1}^2/p_i = \frac{n}{\log(\Delta+1)^{i-2}2^{i^2-4i+1}}.$$

Thus the expected number of comparisons used in the transition from R_k to R_1 is at most

$$\sum_{i=2}^{k} \frac{n}{\log(\Delta+1)^{i-2} 2^{i^2-4i+1}} \leq n \sum_{i \geq 2} 2^{-i^2+4i-1} \leq 13n,$$

so the total expected cost is at most $4n\log(\Delta+1)+13n$ comparisons.

6.2 The General Local-Sorting Algorithm

The limited-degree algorithm solves the local-sorting problem with a cost of $O(n\log(\Delta+1))$ comparisons. Recall that the information-theory lower bound for this problem is $\Omega(\sum_{v \in V(G)} \log(d_v+1))$. If G is a fairly regular graph, then we already have an algorithm which is optimal, up to a constant coefficient. But for many graphs, $n\log(\Delta+1)$ will be much greater than the information-theory lower bound, which suggests that the algorithm might be improved if it were to deal more flexibly with wide variance in the degree sequence.

The general local-sorting algorithm works by covering G by a sequence of graphs, H_i, such that each edge of G is represented in some H_i. In creating these graphs, we ensure that $\Delta(H_i) \leq 2^{2i}-1$, and that

$$\sum_i |V(H_i)| \log \Delta(H_i) = O(\log D(G)).$$

Recall from Section 2.2, that $D(G) = \prod_{v \in G}(d_v+1)$, and that $\log D(G)$ is the information theory lower bound for the local-sorting problem.

By applying the limited-degree algorithm to each H_i, we will locally sort G at an optimal speed. The general algorithm does not incorporate new algorithmic ideas beyond the limited degree algorithm. Rather, it merely "dissects" G into manageable parts, such that the limited-degree algorithm is optimal on each part.

The first step is to discard isolated vertices (they do not come into play, obviously). After an appropriate relabeling, we assume that no vertices of G are isolated. Let U_0 be the set of all vertices of degree 1 in G, and let $G_0 = G - U_0$. We continue by defining U_i as the set of all vertices of degree at most $a_i := 2^{2^i}-1$ in G_{i-1}, and $G_i = G_{i-1} - U_i$, for $i = 1, 2, \ldots$. As G is finite, we eventually reach the empty graph.

For all $i > 0$, let $N_{i-1}(U_i)$ denote the set of all vertices in G_{i-1} adjacent to vertices in U_i. When we delete U_i from G_{i-1}, we must keep track of these adjacencies, in some manner which will retain the efficiency of the limited-degree algorithm. To do so, we "split vertices" by creating the sets R_i and B_i, which will be comprised of "red vertices" and "blue vertices," respectively. These new vertices will correspond to vertices of $N_{i-1}(U_i)$, in a manner which will ensure that all of the edges of G are represented in one of the new graphs, H_i, yet each new graph H_i will be of maximum degree a_i.

To define these vertices, we consider the subgraph generated by including all edges of G_i which have at least one vertex in U_i. For each vertex v in $N_{i-1}(U_i)$ of degree at most a_i, we include one copy, $r_i(v)$, of v in R_i. We also include the edge $xr_i(v)$ for each $x \in U_i$ satisfying $xv \in E(G)$. If v is a vertex of $N_{i-1}(U_i)$ of higher degree, we split it into a number of copies, by creating vertices $b_i^j(v)$ which are included in B_i, the set of blue vertices. Edges are also copied, as above, with each blue vertex assigned between $a_i/2$ and a_i of the edges associated to the original vertex. The number of blue vertices is chosen to make such a division possible. These edges are also included in H_i, which also includes the induced subgraph on U_i.

Each blue or red vertex is assigned the element of the total order corresponding to the vertex of which it is a copy. Thus for $v \in N_{i-1}(U_i)$, for $x \in U_i$, $\phi(r_i(v)) > \phi(x)$ iff $\phi(v) > \phi(x)$, and similarly, $\phi(b_i^j(v)) > \phi(x)$ iff $\phi(v) > \phi(x)$. Thus, if we use the limited-degree sorting algorithm on each H_i, this is equivalent to locally sorting G.

Cost Analysis: Let us define H as the disjoint union of the H_i's, and $h_i = |H_i|$. The maximum degree of H_i is a_i, so we will require at most an expected $O(\sum_i h_i \log(a_i+1))$ comparisons to locally sort H.

To analyze this cost, let $\eta = \sum_i h_i \log(a_i+1)$. Each vertex $v \in V(G)$ shows up in exactly one U_i. Also, for $j < i$, there may be one red copy $r_j(v)$ of v, or several blue copies of the form $b_j^k(v)$. If $v \in U_i$, its original degree d_v must exceed a_{i-1}. Thus $\log(d_v+1) \geq \log(a_{i-1}+1) = \frac{1}{2}\log(a_i+1)$. Thus the contribution of vertices of $\bigcup U_i$ to η is at most $2\log D(G)$.

A vertex $v \in V(G)$ has at most one red vertex derived from it in each $N_{j-1}(U_j)$, for $j \leq \lceil \log\log(d_v+1) \rceil - 1$. Thus its total red contribution to η is at most

$$\sum_{j=1}^{\lceil \log\log(d_v+1)\rceil - 1} \log a_j \leq \sum_{j=1}^{\lceil \log\log(d_v+1)\rceil - 1} 2^j \leq 2\log(d_v+1).$$

Thus the contribution of the red vertices to η is at most $2\log D(G)$.

Blue vertices of H_i are only adjacent to vertices of U_i. As each blue vertex is of degree at least $a_i/2$, and each vertex of U_i is of degree at most a_i, there can be at most $2|U_i|$ blue vertices in H_i. Thus the cost of the blue vertices is at most twice the cost of vertices in $\bigcup U_i$, so the cost of the blue vertices is at most $4\log D(G)$.

Combining all of these costs, it follows that $\eta \leq 8\log D(G)$. Thus the general local-sorting algorithm runs with an optimal expected number of comparisons.

The Set-Sort Problem The above algorithm locally sorts a graph on n vertices with m edges using an expected $O(n\log((2m+n)/n))$ comparisons, where this bound is observed by again noting the concavity of the function $\log x$. Also note that locally sorting a clique is equivalent to fully sorting the set underlying the clique. This leads to the following application of the local-sorting algorithm.

Suppose one wished to fully sort m different subsets, $S_1, S_2, \ldots, S_m$, of a totally ordered set X. This amounts to locally sorting the graph $G = (X, E)$, where $xy \in E$ iff there exists an index i such that $\{x, y\} \subseteq S_i$. The local-sorting algorithm applied to the graph G will totally sort each of the sets S_i, using an expected $O(n\log(\sum_j |S_j|^2/n))$ comparisons.

As mentioned above, the local-sorting algorithm used in this context is referred to as the Set-Sort algorithm. This is the version of the local sorting algorithm which is called by the Set-Maxima algorithm.

6.3 The Randomized Set-Maxima Algorithm

The motivating idea behind the randomized algorithm is similar to that of the rank-sequence algorithm. We reduce the sizes of the sets by taking a small sample, R, and then ignoring all elements in each set S_i less than $r_i := \max S_i \cap R$. We construct R by taking each element of X with a fixed probability, p, which may depend on m and n. Here R serves the role served by the set $Z = \{z_0, z_1, , \ldots, , z_k\}$ of the rank-sequence algorithm.

In this section, we shall once again rely heavily on the notion of *rank*, which was defined earlier as $r(x) = |\{y \in X | y \geq x\}|$. In our earlier usage, there was never more than one underlying set in question. In this section, we will use the notation $r_X(x)$ to refer to the rank of x in X, as we shall measure rank with regard to many different underlying sets. Also, we define the *bottom-rank of x in X* as $b_X(x) = |\{y \in X | y \leq x\}|$ (Note: sometimes what we call "bottom-rank" is referred to as "rank", and the term "top-rank" is used where we use "rank".)

There are two versions of this algorithm: a naive version which uses $O(n \log\log n)$ comparisons, and a sophisticated version which gives the optimal $O(n)$ bound. In the naive version, we fully sort the sample R, while in the sophisticated version, we only sort part of R. Leaving a full discussion of the sophisticated version to later, let us examine the naive version.

The naive version begins with the selection of the random sample R as described above. We choose $p = p(n)$, and then choose R by taking each element of X with probability p, where the choices are independent. We sort R, and define $r_i = \max(R \cap S_i)$. The next step is the construction of the "reduced sets" $S'_i = \{x \in X | x \geq r_i\}, 1 \leq i \leq m$. Consider the family $T_x = \{r_j | x \in S_j\}$. Note that $\{j | x \geq r_j, r_j \in T_x\} = \{j | x \in S'_j\}$, so placing x in T_x for all $x \in X$ will completely determine the sets S'_j.

For each $x \in X$, we perform a doubling search upon the T_x to place x. A *doubling search* first finds the least value of l for which the element of bottom-rank 2^l of T_x is greater than x, and then performs a binary search on the interval of elements of T_x of bottom-ranks in the interval $[2^{l-1}, 2^l)$ to place x. Since T_x is a set of maxima, its elements will tend to be of high bottom-rank in X. Thus we expect a doubling search to be more efficient than a binary search, as a typical x ought to be of low bottom-rank in $T_x \cup x$.

Once we know, for each $x \in X$, the order of x relative to T_x, we form the reduced sets $S'_i = \{x \in X | x \geq r_i\}$. We then use the set-sort algorithm to find the maxima of the reduced sets.

In this cost analysis, we use Jensen's inequality to evaluate a log-sum. Given a random variables Z, which assumes t different positive values, Jensen's inequality implies that $E[\log Z] \leq t \log(E[Z]/t)$.

There are three costs associated with the naive version. First, there is the cost of sorting R. Since $|R| = B(n, p)$, this will come with an expected cost of $np \log(np)$ comparisons. Next, there is the cost of the doubling searches. For a fixed x, the cost of a doubling search on T_x is at most $2 \log |\{r_j \in T_x | x \geq r_j\}$. Since $x \geq r_j$ iff $x \in S'_j$, we have

$$\sum_{x \in X} |\{r_j \in T_x | x \geq r_j\}| = \sum_{j=1}^{n} |S'_j|.$$

Applying Jensen's inequality and the preceding equation, we get the inequality

$$E[\sum_{x \in X} 2 \log |\{r_j \in T_x : x \geq r_j\}|] \leq 2n \log E[(\sum_{x \in X} |\{r_j \in T_x : x \geq r_j\}|/n)]$$

$$= 2n \log E[(\sum_j |S'_j|/n)].$$

The final cost is the $n \log(\sum_j |S'_j|^2/n)$ comparisons used by the set-sort algorithm. We again observe a bound of $n \log(\sum_j E[|S'_j|]/n)$ for this cost by applying Jensen's inequality, as above.

Since each element of S_j is in R with probability p, $|S'_j|$ is a geometric random variable with parameter p, whence $E[|S'_j|] = 1/p$. Our three costs are

1. The sort of R, at an expected $np \log(np)$ comparisons.

2. The doubling searches, at an expected $2n \log(m/np)$ comparisons.

3. The set-sort algorithm, at an expected $n \log(m/np^2)$ comparisons.

For $m = O(n)$, there is no choice of p which will guarantee that the algorithm will use $O(n)$ comparisons, a consideration which will lead us to the sophisticated algorithm.

By way of illustration, consider the case when $m = n \log^c n$, $c > 0$. In this case, we choose the elements of R with the fixed probability $p = 1/\log n$, and the bounds computed above evaluate to an optimal $O(n \log \log n)$. On the other hand, if $m = o(n \log n)$, then the $n \log \log n$ bound obtained with the above choice of p is still $\omega(n \log(m/n))$, so we do not attain the information-theory lower bound with this naive approach. We now consider the sophisticated version of the algorithm. Recalling the insight that most elements x are likely to have a low bottom-rank with respect to T_x, we improve the algorithm by only sorting the low end of R. This will typically be sufficient to place x in T_x. The few exceptional x's are dealt with separately, at a small cost.

At this point, it becomes useful to generalize the question of set-maxima to the t-maxima problem. In this problem, we are asked for the t largest elements of each set, in order. By convention, if $t > |S_i|$, t-maxima asks that S_i be returned fully sorted.

The following is the [GKKS] algorithm LARGE, which finds the largest t elements of each of the subsets $S_1, \ldots, S_m$ of the universe X. LARGE is recursive, and includes a call to the analogous algorithm SMALL (which is just the mirror image of LARGE). Without loss of generality, we assume that $m \geq 2n$.

LARGE $(t, X, \{S_j\}_{j\leq m})$:

1. If $|X| \leq 16$ sort X, and return the largest $\min\{t, |S_i|\}$ elements of each S_i, in order. Otherwise continue.

2. Let R be the sample generated by taking each $x \in X$ independently with probability $1/t$. Call LARGE $(t, R, \{S_j \cap R\})$. For $1 \leq j \leq m$, let r_j be the element of rank t in $R \cap S_j$, or $-\infty$ if $|R \cap S_j| < t$.

3. For each $x \in X$, let $T_x = \{r_j : x \in S_j\}$. Do SMALL $(mt^3/n, R, \{T_x\})$, and let τ_x be the element of T_x of bottom-rank mt^3/n.

4. For each $x \in X$, compare x with τ_x. If $x > \tau_x$, include x in the exceptional set Y. If $x \leq \tau_x$, place x in T_x by performing a doubling search on the portion of T_x sorted in Step 3.

5. Call LARGE $(t, Y, \{S_j \cap Y\})$

6. Let $S'_j = \{x \in S_j \cap (X \setminus Y) : x \geq r_j\} \cup \{y \in S_j \cap Y | r_{S_j \cap Y}(y) \leq t\}$. Apply set-sort to the family $\{S'_j\}_{j\leq m}$.

The expected cost of the algorithm has been trimmed. To take advantage of the fact that most of the doubling searches of the naive algorithm would typically be limited to the lower ends of the sets T_x, the sophisticated algorithm only fully sorts the bottom of each T_x. The set Y consists of those exceptional x for which we cannot use this shortcut, and which are dealt with by a separate recursive call. For this algorithm to be faster than the naive version, the set Y must have a small expected size, and the cost of SMALL$(mt^3/n, R, \{T_x\}_{x\in X})$ must be less than the cost of sorting all of R.

Cost analysis: Comparisons are performed only in the recursive calls, and in Steps 4 and 6. Let $\rho_j = r_{S_j}(r_j)$. For $x \in Y$, Step 4 will involve only one comparison. For $x \in X \setminus Y$, the number of comparisons used in Step 4 is the cost of searching for the interval containing x, and the cost of placing x within that interval. If x lies in the interval of T_x of elements of bottom-rank $[2^{l-1}, 2^l]$, then l comparisons are needed to find this interval, and l comparisons are needed to place x within this interval. Thus the cost of Step 4 is at most

$$\sum_{x\in X} 2\log(|\{j | r_j \leq x, x \in S_j\}|)$$

comparisons. But

$$\sum_{x\in X} |\{j | r_j \leq x, x \in S_j\}| = \sum_j |\{x \in S_j | x \geq r_j\}|.$$

Thus

$$\sum_{x\in X} 2\log(|\{j | r_j \leq x, x \in S_j\}|) \leq 2n\log(\sum_{x\in X} |\{j | r_j \leq x, x \in S_j\}|/n)$$

$$= 2n \log(\sum_j \rho_j / n).$$

In Step 6, the m sets S'_j have size at most $t + \rho_j$. The set-sort algorithm will sort all of the sets with an expected cost of at most $c_0 n \log(\sum_j (t+\rho_j)^2)/n)$ comparisons, for some constant c_0.

We use the following estimates for ρ_j and $(t+\rho_j)^2$.

Lemma 6.2 *As defined above,* $E[\rho_j] \leq t^2$ *and* $E[(t+\rho_j)^2] \leq 4t^4$.

Proof: If $S_j \cap X$ were infinite, ρ_j would be the rank in S_j of the t^{th} largest element of $S_j \cap R$. Since each element of X is in R with probability $p = 1/t$, ρ_j would be the time required for t successes in a sequence of independent Bernoulli trials, each with parameter p. For $i = 1, \ldots, t$, let X_i be the independent geometric random variable representing the number of trials needed to achieve the i^{th} success after the $(i-1)^{st}$ success has been achieved. Since $E[X_i] = t$, for all i, $E[\rho_j] = t^2$. The finiteness of $S_j \cap X$ can only decrease this expected value, so $E[\rho_j] \leq t^2$.

For the second bound, we have $E[(t+\rho_j)^2] = t^2 + 2tE[\rho_j] + E[\rho_j^2]$, so we just need to bound $E[\rho_j^2]$. As before, this is no more than it would be if $S_j \cap X$ were infinite, in which case, with X_i's as above,

$$E[\rho_j^2] = \sum_{i=1}^{t} \sum_{k=1}^{t} E[X_i X_k].$$

For $i \neq k$, $E[X_i X_k] = E[X_i]E[X_k] \leq t^2$, as X_i and X_k are independent. A geometric random variable Z with parameter p satisfies $E[Z^2] = (2-p)/p^2$, so $E[X_i^2] \leq 2t^2 - t$. As we sum over i and k, we get

$$E[(t+\rho_j)^2] \leq t^2 + 2t^3 + (t^2 - t)t^2 + t(2t^2 - t) = 3t^3 + t^4 \leq 4t^4.$$

□

Now using the concavity of the logarithm function, we have

$$E[2n \log(\sum_j \rho_j / n)] \leq 2n \log(\sum_j E[\rho_j]/n) \leq 2n \log(mt^2/n).$$

Also,

$$E[c_0 n \log(\sum_j (t+\rho_j)^2/n)] \leq c_0 n \log(\sum_j E[(t+\rho_j)^2]/n)$$
$$\leq c_0 n \log((4mt^4)/n).$$

Thus the expected cost of (4) and (6) is at most

$$2n \log((mt^2)/n) + c_0 n \log((4mt^4)/n) \leq cn \log(mt/n),$$

where $c = 4 + 4c_0$.

We must also consider the costs of the recursive calls. Let $f(t,n,m)$ denote the average cost of LARGE(t, X, S_j), including recursive calls on worst possible input, such that $|X| = n$ and the number of subsets is m. Let $g(t,n,m) = 2cn \log(mt/n)$.

Theorem 6.1 *Using the above definitions,*

$$f(t,n,m) \le g(t,n,m) \tag{8}$$

for $m \ge 2n$, *and* $t \ge 16$.

Proof: This is true for $n \le 16$, as $t \ge n$, and one can sort X with at most $dn\log(n)$ comparisons, for d a small constant.

Assume $n > 16$. One level of the algorithm makes two recursive calls to LARGE, and one call to SMALL. Using the bound obtained above for the comparisons not used in the recursive calls, we see that $f(t,n,m)$ satisfies the recursive inequality

$$\begin{aligned} f(t,n,m) &\le cn\log(mt/n) + E[f(t,|R|,m)] \\ &+E[f(mt^3/n,|R|,n)] + E[f(t,|Y|,m)], \end{aligned}$$

where the expectations are taken as the sample R varies.

By induction,

$$\begin{aligned} f(t,n,m) &\le cn\log(mt/n) + E[g(t,|R|,m)] \\ &+E[g(mt^3/n,|R|,n)] + E[g(t,|Y|,m)]. \end{aligned}$$

Since $n\log 1/n$ is concave, we may infer the inequality

$$\begin{aligned} f(t,n,m) &\le cn\log(mt/n) + g(t,E[|R|],m) \\ &+g(mt^3/n,E[|R|],n) + g(t,E[|Y|,m). \end{aligned} \tag{9}$$

We'll need the following estimates on $E[|R|]$ and $E[|Y|]$

Lemma 6.3 $E[|R|] = n/t$ *and* $E[|Y|] \le n/t$.

Proof of Lemma: The first statement is obvious. As for the second statement, note that $x \in Y$ iff there are at least mt^3/n values of $r_j \in T_x$ which are less than x. Thus

$$(mt^3/n)|Y| \le |\{(x,j) : x \ge r_j, x \in S_j\}| = \sum_j \rho_j.$$

Hence $E[|Y|] \le (n/mt^3)E[\sum_j \rho_j] = (n/mt^3)(mt^2) = n/t$. □

We may thus reformulate (8) as

$$f(t,n,m) \le cn\log(mt/n) + 2g(t,n/t,m) + g(mt^3/n,n/t,n), \tag{10}$$

which leads to

$$\begin{aligned} f(t,n,m) &\le cn\log(mt/n) + 2(2c(n/t)\log(mt^2/n)) + 2c(n/t)\log(mt^4/n) \\ &\le cn\log(mt/n)(1+8/t+8/t) \le 2cn\log(mt/n), \end{aligned}$$

since $t \ge 16$, and $m \ge 2n$. □

The algorithm solves t-maxima with an expected $O(n\log(mt/n))$ comparisons. In particluar, if we take $t = 1$, we solve set maxima with $\Theta(n\log(m/n))$ expected comparisons, which is optimal as a function of m and n.

7 Conclusion

The randomized algorithm described in Section 6.3 solves the set-maxima problem with an optimal expected number of comparisons. It is still not known whether there exists a deterministic algorithm which would can achieve an $o(n \log n)$ bound, say, when $m = n$. Furthermore, we essentially know no non-trivial ($\omega(n)$) lower bounds for $m = n$.

Deterministic algorithms exist for the cases explored in [K] and [BMN], but the algorithms for these cases depend upon special features of the set-systems, and do not appear to generalize readily to arbitrary set-systems. The Komlós algorithm would appear to be impossible to generalize, as it relies heavily upon properties of trees. On the other hand, while the rank-sequence algorithm is a general approach, its successes in Section 5 depend upon expansion properties of the set-systems. The simple rank-sequence algorithm fails to solve the set-maxima problem optimally when applied in the most general setting. As demonstrated in Section 5.3, the worst-case behavior of the rank-sequence algorithm is as bad as that of a brute-force method.

Thus it remains open whether good algorithms exist to solve the general set-maxima problem in $O(m+n)$ time. It would be interesting to find other particular classes of set-systems for which $o(n \log n)$ algorithms exist. Given the results with local-sorting, it would be interesting to explore a set-system whose family of sets is the set of radius-one balls of a graph. Of particular interest would be the set-system of radius-one balls of the graph (Q_n, E), where $Q_n = \{0,1\}^{[n]}$, and $vw \in E$ if $v = (v_1, \ldots, v_n)$, $w = (w_1, \ldots, w_n)$, with $v_i = w_i$ for exactly $(n-1)$ values of i. In this instance, we have n sets of size $\log n$, so a brute force method would use $n \log n$ comparisons. This set-system is somewhat expansive, but less so than the cases studied in Section 5

The local-sorting problem is optimally solved by the randomized algorithm, but as with the set-maxima problem, no efficient deterministic algorithms are known to solve the problem for arbitrary graphs. Also, the local-sorting algorithm uses comparisons between distant elements. Might there be a (randomized) algorithm which solves the problem at an optimal speed, while only making comparisons between elements whose distance is less than a specified comparison horizon $h = h(n)$? For dense graphs, local sorting would appear to be as difficult as sorting the underlying set. Do good deterministic solutions exist for sparser graphs?

Instead of the search problems of set-maxima and local-sorting, we may consider the possibly easier *verification problem.* In the set-maxima verification problem, we are given a solution vector $(x_1, x_2, \ldots, x_n)$, and asked to verify that $x_i = \max S_i$ for all i. Similarly, local-sorting verification consists of verifying that a given acyclic orientation corresponds to the order upon the vertex set of a graph. Rather surprisingly, these formally much easier problems seem in practice to be of about the same difficulty as the corresponding search problems.

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[Al] N. Alon, "Eigenvalues, Geometric Expanders, Sorting in Rounds, and Ramsey Theory," *Combinatorica* 6 (1986) pp. 207-219.

[Ba] L.M. Batten, *Combinatorics of Finite Geometries,* Cambridge University Press, 1986.

[BMN] A. Bar-Noy, B. Motwani, and J. Naor, "A Linear Time Approach to the Set Maxima Problem."

[DRT] B. Dixon, M. Rauch, and R. E. Tarjan, "Verification and sensitivity analysis of minimum spanning trees in linear time," *SIAM J. of Computing* 21, 1992, pp. 1184-1192.

[GKKS] W. Goddard, C. Kenyon, V. King, and L. Schulman, Optimal Randomized Algorithms for Local Sorting and Set-Maxima."

[GYY] R. Graham, F. Yao, and A. Yao, "Information Bounds are Weak in the Shortest Distance Problem," *JACM* 27 (1980) pp. 428-444.

[HLP] G.H. Hardy, J.E. Littlewood, and G. Pólya, "Some simple inequalities satisfied by convex functions," *Messenger of Mathematics* 58, 145-152.

[H] D. Harel, "A Linear Time Algorithm for the Lowest Common Ancestor Problem."

[Kn] D. Knuth, *The Art of Computer Programming, Vol. 3 Sorting and Searching,* Addison-Wesley, 1973.

[Ko] J. Komlós, "Linear Verification for Spanning Trees," *Combinatorica* 5 (1985) pp. 57-65.

[Ko2] J. Komlós, private correspondence.

[MT] U. Manber and M. Tompa, "The Effect of Number of Hamiltonian Paths on the Complexity of a Vertex-Coloring Problem," *SIAM J. Comp.* 13 (1984), pp. 109-115.