Distribution-sensitive algorithms *

Sandeep Sen and Neelima Gupta
Department of Computer Science and Engineering,
I.I.T., New Delhi 110016, India
{ssen , neelima}@cse.iitd.ernet.in

Abstract

We investigate a new paradigm of algorithm design for geometric problems that can be termed *distribution-sensitive*. Our notion of distribution is more combinatorial in nature than spatial. We illustrate this on problems like planar-hulls and 2D-maxima where some of the previously known output-sensitive algorithms are recast in this setting. In a number of cases, the distribution-sensitive analysis yields superior results for the above problems. Moreover these bounds are shown to be tight in the linear decision tree model.

Our approach owes its spirit to the results known for sorting multisets and we exploit this relationship further to derive fast and efficient parallel algorithms for sorting multisets along with the geometric problems.

1 Introduction

The practice of measuring efficiency of algorithms only with respect to input-size has a major drawback. By restricting ourselves to only one parameter, we are trying to optimize the number of steps assuming worst-case scenario for that parameter only. Perhaps we are losing out on other additional parameters that capture the complexity of the problem more accurately. Suppose the running time of algorithm A (for some problem) is $O(n^2)$ where n is the size of input. An alternate algorithm B has running time $O(n \log n + k)$ where $k \leq n^2$ is another parameter for the problem. Then in a large number of cases where $k = o(n^2)$ algorithm B will be superior although in the worst-case both have similar performances. In particular, if the lowerbound (with respect to input-size only) is $\Omega(n^2)$ then both the algorithms are optimal by definition. The anomaly clearly lies in our choice of parameters to measure the running time.

*A preliminary version of the results in this paper appeared in the Scandenavian Workshop on Algorithmic Theory, Stockholm, Sweden, 1998.

One of the early examples of this type is that of constructing the convex-hull of point sets. On the plane, the known lower bound [2, 27] was $\Omega(n \log n)$ and a number of algorithms matched this bound. However, it was also known that if the number of hull vertices are small - say $O(1)$, the hull can be constructed in $O(n)$ time. This lacuna was resolved by Kirkpatrick and Seidel [15] who showed that the complexity of this problem is $\Theta(n \log h)$ [1] where h is the number of hull vertices. This not only resolved the mystery but also shed more light on the complexity of the problem in terms of two parameters. What made their algorithm more interesting was that no prior knowledge of the output-size was required for the algorithm.

A *multiset* is a a set with repeated elements. It is a classical result from information-theoretic arguments that sorting n elements requires $\Omega(n \log n)$ comparisons in the worst case. However, if there are repeated elements, say the input consists of only 0's and 1's, then such a set can be sorted in linear time easily. It was shown in [21] that the complexity for sorting a multiset is $\Theta((n \log n - \sum_i n_i \log n_i) + n)$ where n_i is the number of elements that have the i-th largest value. Furthermore, the algorithm does not require any prior knowledge of the n_i's although the running time is sensitive to these values. Note that for $i = 2$, the running time is $O(n)$ irrespective of the number of 0's and 1's.

The starting point of our work is a simple observation that can be summarized as follows.

Observation 1 *Given a multiset of cardinality n and h distinct values, it can be sorted using $O(n \log h)$ comparisons.*

This follows from the fact that $n \log n - \sum_i n_i \log n_i$ is maximized at $n_i = \frac{n}{h}$. The bound in the previous observation is explicit in several earlier papers investigating the running time of quicksort on multisets ([25]). In other words, if we were to measure the complexity of sorting multisets in terms of number of distinct values then the complexity of the problem has a characteristic similar to that of planar hulls mentioned earlier. However, in some sense this is a *loss of information* or a weaker characterization of the complexity of sorting multisets. In a large number of cases, especially for very non-uniform distribution of the n_i's, $O(n \log n - \sum_i n_i \log n_i)$ can be much less than $O(n \log h)$. As a special case consider all n_i's to be 1 except n_h which is $n - h - 1$ - in this case the running time of $O((n - h - 1) \log \frac{n}{n-h-1} + (h-1) \log n)$ is linear when $h \leq \frac{n}{\log n}$. Contrast this with the bound of $O(n \log n)$ from Observation 1.

Turning around this argument, one suspects that there may be a more precise characterization of the complexity of some of the geometric problems in the context of output-size sensitivity. As we shall show in the subsequent section that this is indeed so and we are able to design algorithms that take advantage of *distribution* of the input points *without*

[1] We will use Θ to denote that the upperbound and lowerbound are asymptotically same

prior knowledge of the distribution. We will define the notion of *distribution* more precisely in the context of the specific problems. In most cases we will be able to reanalyze the existing algorithms in this light - so we don't claim much originality of the algorithms as such. However, it may be viewed as an alternative framework for analyzing algorithms that can pay rich dividends for other problems.

The remaining paper is organized as follows. We will illustrate our new paradigm on the problem of determining the maximal vectors in two dimensions followed by distribution-sensitive algorithms for planar-hulls. Subsequently, we will present matching lowerbounds in the linear decision tree and in the algebraic model (under a mild assumption for the latter model). In the remaining paper we will present efficient distribution-sensitive parallel algorithms for the geometric problems and that of sorting multisets. Our output-sensitive algorithm for sorting multisets runs in $O(\log h \log^* n)$ expected time using $O(n \log h)$ operations. Farach and Muthukrishnan [8], obtained a similar result for *Renaming* - a problem closely related to Multisorting. Other than the above work, we are not aware of any comparable results for these problems.

2 Maximal Vectors in plane

Let T be a set of vectors in R^d. The partial order $\leq_M$ is defined for two vectors $x = (x_1, x_2, \ldots x_d)$ and $y = (y_1, y_2, \ldots y_d)$ as follows : $x \leq_M y$ (x is *dominated* by y) if and only if $x_i \leq y_i$ for all $1 \leq i \leq d$.

Definition A vector $v \in T$ is said to be maximal element if it is not dominated by any other vector $w \in T$.

The problem of maximal vectors is to determine all maximal vectors in a given set of input vectors. Here we will consider the special case when the vectors are two-dimensional. The output looks like a staircase (Figure 3). We will denote the x and y projections of a vector p by $X(p)$ and $Y(p)$ respectively.

An early work [17] on maximal vectors established that the complexity of the maximal vectors (on plane) is $\Theta(n \log n)$ for an input of n vectors. A later development due to Kirkpatrick and Seidel [16] showed that a more precise characterization of the complexity can be obtained by considering the size of the output, namely the number of maximal vectors. Using a technique that they called *marriage before conquest*, they developed an algorithm that had running time $O(n \log h)$ where h is the number of maximal vectors. No prior knowledge of h was required and further they showed that the lower bound is also $\Omega(n \log h)$ in terms of n and h. In the following description we will use the term vector and point interchangeably.

Definition Given T, let $t_1, t_2 \ldots t_h$ denote the maximal vectors in a reverse sorted order along x direction (i.e. as we go up the staircase). Let us denote the set of input vectors

Algorithm Max1(T)

1. If $|T| \leq 1$ then return T
2. else partition T into two (nearly) equal size sets T_1 and T_2 using the median of T with respect to x coordinate.
(* T_1 has smaller x-coordinate than T_2 *)
3. Determine the leftmost maximal point (of T) in T_2, say L and the rightmost maximal point (of T) in T_1, say R. Let $T_1' = T_1 - \{q | q \in T_1$ and $X(q) > X(R)\}$. $T_2' = T_2 - \{p | p \in T_2$ and $X(p) \leq X(L)\}$.
(* Discard all points of T that are between R and L *)
4. Return (Max1 (T_2') $\cup$ Max1 (T_1'))

Figure 1: A divide-and-conquer Algorithm for computing maximal vectors on plane

that lie between $X(t_i)$ and $X(t_{i+1})$ including t_i by N_i - note that except t_i none of these vectors are maximal. If $|N_i| = n_i$, then we say that the distribution of T is characterized by $(n_1, n_2 \ldots n_h)$. Also for all the vectors $v \in N_i$, we will define $slab(v) = i$. Note that $\sum_i n_i = n$.

See Figure 3 for an illustration of the definition.

2.1 A distribution-sensitive algorithm

The algorithm in Figure 1 is a slight modification of the output-sensitive algorithm described by Kirkpatrick and Seidel [16]. The time complexity of the algorithm is dominated by Steps 2 and 3, which involves median finding and filtering with respect to the closest maximal vector to the right of the median line. Notice that L has the maximum y coordinate in T_2 and R is a point that has the maximum x coordinate among points in T_1 that are not dominated by L. Recall that selection (including median) can be done in linear time. If the current interval (in the recursive call) is $\mathcal{I}$ then the above steps take $O(|\mathcal{I}|)$ steps - for concreteness let us assume $c|\mathcal{I}|$ steps for some constant $c > 1$. We will view this as an average of c steps for every point in the interval, i.e. we will assign a cost of c to every point. We will analyze the above algorithm by keeping track of the total cost assigned to each point over the entire algorithm. As the algorithm progresses, the size of the intervals (or equivalently the sub-problem size in the recursive calls) become smaller. With each point p, we can associate an interval $I(p)$ containing the point p. When the algorithm is called recursively on $I(p)$, the subsequent interval containing p becomes about half the size of $I(p)$ and p is assigned an additional cost c. We will denote the sequence of intervals containing p (as the algorithm proceeds) by $I_0(p), I_1(p) \ldots I_k(p)$ where $I_0(p)$ is the given set T and $I_{j+1}(p)$ is obtained by splitting $I_j(p)$. The total cost accumulated by p is ck. Clearly

$k \leq \log n$ but we can get a more accurate estimate on k that will depend on the distribution of T.

Lemma 2.1 *The total cost accumulated by a point $p \in T$ is no more than $c \cdot \log \frac{n}{n_r}$ where $r = slab(p)$.*

Proof This follows from the observations that $|I_k(p)| \geq n_r$ and $|I_{j+1}(p)| \leq |I_j(p)/2|$. The second property is obvious. To see the first property, notice that a point $p \in N_i$ is charged till N_i is not split in Step 2 of Algorithm **Max1**. When N_i is split the maximal point dominating p to its right is computed and subsequently p is discarded in Step 3. Therefore, if the interval containing p is split k times then,

$$\frac{n}{2^k} \geq |I_k(p)| \geq n_r$$

from which the lemma follows.

Therefore we obtain the following.

Theorem 2.1 *The total running time of the algorithm can be bounded by $O(\sum_i n_i \lceil \log \frac{n}{n_i} \rceil + n)$ or alternately $O((n \log n - \sum_i n_i \log n_i) + n)$.*

As observed earlier, this is always bounded by $O(n \log h)$. This analysis not only gives a better bound but also sheds more light on the somewhat 'mysterious' bound of $O(n \log h)$ that came out of solving a recurrence.

Next, we analyze a slightly modified version of the above algorithm where we replace Step 3 with the following.

3'. Determine the point $L \in T_2$ that has the largest y coordinate. Let $T_1' = T_1 - \{q | q \in T_1$ and $q \leq_M L\}$. $T_2' = T_2 - \{p | p \in T_2$ and $p \leq_M L\}$.
(* Discard all points of T that are dominated by L *)

Here, instead of filtering with respect to R and L, we use only L. This may actually change the way the non-maximal points are eliminated quite drastically. Now L can eliminate points in other slabs of T_1. This makes the algorithm slightly harder to analyze and it is not clear if the previous bound still applies.

Let us focus on a particular slab, say r. Some of the points may be eliminated before $I_r = \log \frac{n}{n_r}$ iteration causing the slab to become *thinner*. This could delay elimination of the remaining points according to our previous analysis beyond I_r iterations. We will try to bound this extra cost. Extending our previous observation, at least half the points in the slab must be eliminated before we encounter the I_r iteration. So at most $n_r/2$ points will encounter an extra iteration. Continuing the same argument, at most $n_r/4$ points in the slab will survive the $I_r + 1$ iteration. This adds a total of cost of at most $O(n_r)$ to the points in slab r to the original algorithm. Thus the upperbound of Theorem 2.1 remains unchanged with this modification and is likely to be better in practice as some of the points are eliminated sooner.

2.2 A heap-based approach

Munro and Raman [20] described a very simple heap-based multiset sorting algorithm (that is in-place) and achieves optimality up to constant factor of the leading term. The same can be adapted for finding maximal vectors that gives an algorithm which has a somewhat different behavior than the previous algorithm.

The crux of Munro and Raman's method is the observation that while doing heapsort, all the elements that are equal to the maximum (root) element form a sub-tree rooted at the top of the heap. This makes it easy to locate these elements in time proportional to the number of such elements. For example, if there are n_i such elements equal to the maximum, then using a depth-first-search from the root, we can locate all these elements in $O(n_i)$ steps (recall that a heap is similar to a complete binary tree). Now instead of discarding the root as in normal heapsort, we start deleting the n_i elements in decreasing distance from the root. Recall that if an element is distance d from the bottom of the heap (leaf-level of the tree), then deletion and the subsequent restoration of the heap takes $O(d)$ comparisons. Munro and Raman show that the maximum number of comparisons are required when the n_i elements form a balanced tree which yields the bound $2(n_i \log n - n_i \log n_i)$ comparisons for the n_i elements and $2(n \log n - \sum_i n_i \log n_i) + O(n)$ comparisons overall.

For determining maximal vectors, we build a heap on the x-projections of the vectors. Clearly, the vector t_1 with the maximum x-coordinate is at the top of the heap. Analogous to the arguments of Munro and Raman, we claim that the vectors in N_1 (these are the vectors dominated by t_1 only) form a subtree rooted at the top of the heap. However, unlike sorting, it is not straightforward to detect these vectors. If we somehow knew t_2, then our job would have made simple as we would have eliminated all vectors with x coordinates between $X(t_1)$ and $X(t_2)$. The vector t_2 (and in general t_{i+1}) can be characterized by the following straightforward observation.

Observation 2 $t_{i+1} = \max\{X(p) : Y(p) > Y(t_i)\}$, *that is,* t_{i+1} *has the largest x-coordinate among all vectors that have larger y-coordinate than* t_i.

We locate the elements of N_i as follows. We do depth-first-search until we hit a vector that has larger y-coordinate than t_i. This gives us a set of vectors N_i' containing N_i. From Observation 2, t_{i+1} is at the root of the heap after restoration.

By using the same analysis as that of Munro and Raman, this algorithm has running time of $O(n \log n - \sum_i n_i' \log n_i')$, where $n_i' = |N_i'|$. This running time is not in conformity with our definition of distribution and cannot be compared directly with the result of Theorem 2.1. In fact, n_i''s could be different for different permutations of the same input ! This is because the n_i''s are dependent on the initial heap which can be different for different permutations.

Remark This also exposes an inherent difficulty of the distribution-based approach, namely that of a robust definition of the distribution itself. Certain definitions would be suitable for certain approaches as was the case for the divide-and-conquer based approach. However, note that even in the heap-based approach the running time can always be bounded by $O(n \log h)$ as $\sum_i n_i' = n$.

3 Planar Hulls

After Kirkpatrick and Seidel's discovery of the output-sensitive algorithm, there have been some recent simplification by Chan et al. [5] and Wenger[26] and Bhattacharya and Sen [3]. The latter two are randomized and variations of a folklore method called quickhull.

The definition of distribution that we will use for planar hulls is similar in spirit to the two-dimensional maximal vector problem. We will confine ourselves to upper-hulls only - for the lower hull it is symmetric. Let $v_1, v_2 \ldots v_h$ be the hull vertices in anti-clockwise order. We will denote the input points that lie below the edge v_i, v_{i+1} by N_i and let $n_i = |N_i|$. In other words, for all points $p \in N_i$, the vertical visibility edge is $v_i v_{i+1}$. Also for any point $p \in N_i$, we define $slab(p) = i$. The vector $(n_1, n_2 \ldots n_h)$ is the distribution vector for the convex-hull problem. As before $\sum_i n_i = n$.

3.1 Kirkpatrick and Seidel's approach

We will first analyze the original approach of [15] in the context of distribution-sensitive algorithm. Later, we shall show that the simplified algorithms that do not use linear programming also lead to comparable running times.

The underlying structure of the algorithm is very similar to that of Algorithm Max1 (see Figure 2). Step 3 can be done in $O(|T|)$ steps [19, 7] as are the other steps. The analysis can also be done along the lines of Algorithm Max1. By accumulating cost with every point that participates in the interval corresponding to the recursive calls, we obtain a bound identical to Lemma 2.1. Thus we can summarize as follows.

Theorem 3.1 *The total running time of the algorithm Hull1 can be bounded by* $O(\sum_i n_i \lceil \log \frac{n}{n_i} \rceil + n)$ *or alternately* $O((n \log n - \sum_i n_i \log n_i) + n)$.

Remark The above algorithm differs from the more standard version where the extreme points are part of the recursive calls in Steps 4 and 5. If L are R are respectively the left

Algorithm Hull1(T)

1. If $|T| \leq 1$ then return T
2. else partition T into two (nearly) equal size sets T_1 and T_2 using the median of T with respect to x coordinate.
(* T_1 has smaller x-coordinate than T_2 *)
3. Find the hull-edge (also called the bridge) between T_1 and T_2. This can be done using linear programming with two variables.
4. If t_1t_2 is the bridge (supporting points are t_1 and t_2 in T_1 and T_2 respectively), let $T_1' = T_1 - \{q|q \in T_1 \text{ and } q \text{ lies below } t_1t_2\}$. $T_2' = T_2 - \{p|p \in T_2 \text{ and } p \text{ lies below } t_1t_2\}$.
(* Discard all points of T that lie below the edge t_1t_2. *)
5. Return (Hull1 $(T_2') \cup$ edge t_1t_2 $\cup$ Hull1 (T_1')).

Figure 2: KS algorithm for planar hull

and the right extreme points, then all the points in the trapezoid Lt_1t_2R are discarded and subsequently Lt_1 (t_2R) become the extreme points for left (right) subproblem. We can take advantage of early elimination of some points and carry out an analysis identical to the modified version of out maximal-points algorithm. However, we cannot rotate the axes in our recursive calls. This would pose problems in our framework as the distribution is not *rotation invariant*, to the contrary it can change drastically.

The simplified algorithms in [5, 3, 26] are also divide-and-conquer algorithms but they avoid the complicated linear-time linear programming of Step 3. The algorithm of [5] as well as the randomized version [3] detects an output vertex instead of an edge and simultaneously split the problem into roughly equal size subproblems (each sub-problem no more than 3/4 the original size). Subsequently all the points in the triangle defined by the extreme points and the output-vertex are eliminated. To preserve our definition of the distribution vector, we can modify the algorithms slightly as follows. After detecting the output vertex and splitting the problem into roughly equal size subproblems, we can find an incident edge (by a simple angular scan) and then eliminate all points below this edge. Then we can use the previous analysis directly to obtain a bound identical to Theorem 3.1. On the other hand, it is not clear if the algorithms of Chan [4] and Clarkson and Shor [6] (that do not use divide-and-conquer) can be modified to make them attain distribution-sensitive bounds.

4 Lower bounds

In this section, we will address the complexity of the maximal vector problem in terms of distribution in an algebraic model. We will show that the upperbounds obtained in the previous section are optimal to within a constant factor in the linear decision tree model. In a more general algebraic model, the lower bound holds for the following class of algorithms. For our lower-bound proof, we assume that for every non-maximal point p, the algorithm provides a *witness* q, that is p is dominated by q. If there are many witnesses, the algorithm will provide at least one. This property holds for all known algorithms although it is not clear if this holds for all algebraic decision tree for the maximal problem. The lower bound makes use of the following general result of Ben-Or [2].

Theorem 4.1 *Let W be a subset of $\mathbb{R}^d$ and let $\#(W)$ represent the number of (disjoint) connected components of W. Then any algebraic decision tree that correctly determines the membership query $x \in W$ has $\Omega(\log \#(W) - d)$ height.*

A similar result holds for fixed-degree algebraic decision trees (Steele and Yao [29]).

Instead of applying Ben-Or's lowerbound to the original problem $\mathcal{A}$, we will define a related problem $\mathcal{B}$ where it is easier to count the number of connected components in the underlying solution space. The problem $\mathcal{B}$ is chosen such that it is easily reducible (typically linear-time) to the original problem $\mathcal{A}$.

In our case, we will use a construction due to [14] to define problem $\mathcal{B}$. Let $S_1 = \{p_1, p_2 \ldots, p_m\}$ where $p_i = (i, m - i + 1)$ for $1 \leq i \leq m$. Let L_1 and L_2 be the lines whose equations are $x + y = m + 1$ and $x + y = m$ respectively. All points in S_1 are maximal and they lie on L_1 (see Figure 4).

Definition Dominance Testing with restriction (DTR). Given a set $S_2 = \{q_1, q_2 \ldots q_{n-m}\}$ with $n - m$ points, determine whether they are all dominated by the points in S_1 and lie strictly between lines L_1 and L_2.

This problem is clearly $O(n)$ time transformable to the problem of determining the maximal vectors[2] of $S_1 \cup S_2$. If the number of maximal elements in $|S_1 \cup S_2| \neq m$, then the answer to the DTR problem is NO. Otherwise, the answer to DTR is YES iff all the points in S_2 lie between L_1 and L_2.

The above transformation can be extended to include the distribution vector. We define the problem of Dominance Testing with Restriction and Distribution (DTRD) to be an instance of DTR along with a distribution vector $(n_1, n_2, \ldots n_m)$ where $\sum_i n_i = n - m$.

Definition DTRD: Given a set $S_2 = \{q_1, q_2 \ldots q_{n-m}\}$ with $n - m$ points and a vector $(n_1, n_2, \ldots n_m)$ where $\sum_i n_i = n - m$, determine whether the points in S_2 are all dominated by the points in S_1 and lie strictly between lines L_1 and L_2. Moreover, it must also verify if the distribution matches the given vector.

This problem is $O(n)$ transformable to the maximal vectors problem for algorithms that report the distribution by using a slight modification of the previous argument. However,

[2]Strictly speaking, we should consider the decision version of this problem, i.e., if the output contains m vectors. Since it does not cause any technical difficulty we have used the standard version.

this (reporting distribution) is not a natural restriction as evidenced by the heap-based approach.

We shall first prove a lower bound for the DTRD problem. The effective problem in DTRD is to determine whether each of the points of S_2 lies in one of the regions T_i and moreover, T_i contains exactly n_i points. Now, each instance of a DTRD problem can be considered to be a point in $\mathbb{R}^{2(n-m)}$. Consider the number of ways of assigning $n-m$ points to the T_i's such that T_i contains n_i points - denote this by K. Each YES instance of DTRD corresponds to such an assignment. Moreover, if $S_2{}'$ and $S_2{}''$ are two distinct assignments, then there are at least two indices l such that $q_l' \in S_2'$ and $q_l'' \in S_2''$ are in different T_i's. Any continuous path from S_2' to S_2'' in $\mathbb{R}^{2(n-m)}$ must necessarily pass through a point that is not a YES instance to the DTRD. Either all the points in S_2 do not between L_1 and L_2 or some point in S_2 is not dominated by a point in S_1. Hence these are in separate connected components of the solution space.

From the elementary combinatorics of assigning (non-identical) balls to bins, we know that $K = \frac{(n-m)!}{n_1! \cdot n_2! \ldots n_m!}$. By simplifying using Stirling's approximation, we obtain the following result as a consequence of Theorem 4.1

Lemma 4.1 *Any algebraic computation tree that solves the maximal vector problem with distribution vector $(n_1, n_2 \ldots n_m)$ has height at least $\Omega(\sum_i n_i \log\left(\frac{(n-m)}{n_i+1}\right))$.*

We reduce DTRD to DTR as follows. If the answer to DTR is NO, then the answer to DTRD is NO. If the answer to DTR is YES, then the answer to the DTRD problem is YES iff the distribution is the required one. If we force the DTR algorithm to report the distribution, then our reduction is clearly valid. However, we can do away with this restriction by a slight modification of the DTR algorithm. Whenever a point q_i is found to be dominated by some other point [3] q_j, we introduce a directed edge from q_i to q_j if there isn't any edge going out of q_i. After the termination of the DTR algorithm, we have a directed forest with at most T edges where T is the running time of the DTR algorithm. If we compress the forest (say, by running a connected-component algorithm), we can determine the distribution vector in $O(T+n)$ time. All points that are dominated by p_i (i.e., they lie in the region T_i) are in the directed tree rooted at p_i. Note that all maximal points are roots of some directed tree and that any other point that is not a maximal point will have a parent. With the above modification, the running time of the DTR algorithm does not increase by more than a constant factor.

For linear-decision trees we can do away with the above assumption because of the following property.

Lemma 4.2 *Any linear decision tree that solves the DTR problem has the property that a non-maximal point corresponding to a leaf-node has a unique maximal point as witness.*

[3] this is where we require a witness for every non-maximal point.

Proof Clearly every non-maximal point p has a witness, but it may be of the form p_i or p_j. Consider the following example involving three input points $(x_i, y_i), i = 1, 2, 3$ [28]. We can have a branch corresponding to the following decisions.

$$x_1 > x_2, \quad y_2 > y_1, \quad x_1 > x_3, \quad y_2 > y_3$$
$$(x_3 - x_2)(y_3 - y_1) < 0$$

The result is that (x_1, y_1) and (x_2, y_2) are maximal and (x_3, y_3) is dominated by one of them. We will show that this is not possible if only linear decisions are allowed.

Recall that every node corresponds to a convex region in the linear-decision trees and there is at most one connected component in any leaf-node. Moreover, any non-maximal point must correspond to a convex-region in $\mathbb{R}^{2n}$ as it must be the intersection of half-spaces. In DTR, there is no intersection between the regions dominated by p_i and p_j, i.e., the regions T_i and T_j are not connected. Consequently, in any leaf node, for every non-maximal point, there is a unique witness from the set $\{p_1, p_2 \ldots p_m\}$. (A convex region must lie entirely in regions corresponding to T_i or T_j). We can (hypothetically) label every non-maximal point q by the unique maximal point witness.

We now claim that any linear decision tree for DTR can be transformed to a linear-decision tree for DTRD without increasing the running time. For any fixed distribution vector, we will retain only those leaves that have unique maximal witnesses for the non-maximal points according to the distribution vector. Although this is non-uniform, it is sufficient for our lower bound argument as the bound of Theorem 4.1 holds in the non-uniform case also.

An $\Omega(m \log m)$ lowerbound easily follows from the order-types of the output points (for example see [14]). In conjunction with Lemma 4.1, we obtain the following theorem (for example by separating the cases $m \leq n/2$ and $m > n/2$).

Theorem 4.2 *Any algorithm in the linear decision tree that solves the maximal vector problem takes time at least $\Omega(\sum_i (n_i + 1) \log \left(\frac{n}{n_i+1}\right))$ for distribution vector $(n_1, n_2 \ldots n_m)$.*

Since we make use of the witnesses (for non-maximal points), we obtain a slightly weaker corollary for the more general algebraic computation trees.

Corollary 4.2.1 *Any algorithm in the algebraic computation tree that solves the maximal vector problem producing witnesses for non-maximal points takes time at least $\Omega(\sum_i (n_i + 1) \log \left(\frac{n}{n_i+1}\right))$ for distribution vector $(n_1, n_2 \ldots n_m)$.*

Remark Our definition of distribution vector is actually $(n_1 + 1, n_2 + 1 \ldots)$ that includes the maximal vectors. So the expressions for lower and upper bounds are identical.

Using the construction of Figure 5, we can prove a result similar to Corollary 4.2.1 for planar hulls. See Kapoor and Ramanan [14] for justification of the lower-bound construction. A witness for a non-extreme point can be defined as follows. Consider a triangle (defined by input points) that encloses the non-maximal point - the highest vertex of the triangle will be a witness.

5 Optimal distribution-sensitive parallel algorithms

In this section we present a parallel algorithm to sort the multisets. The underlying structure of the algorithm is similar to that of **Max1** and **Hull1**, i.e. the algorithm is recursive and we use the technique of 'marriage before conquest' of Kirkpatrick and Seidel. Subsequently we show that the algorithms **Max1** for planar vector maxima and **Hull1** for planar convex hulls can be directly implemented in parallel in the same bounds as the algorithm **ParMset1** for sorting multisets. Unless otherwise mentioned, the model used in this section is the CRCW PRAM.

Let T be the input set with repeated elements.

Algorithm ParMset1(T)

1. If the total size of the problem at the i^{th} stage is $\leq n/\log n$ then solve the problem directly using an optimal work, $O(\log n)$ time algorithm.

2. else

   (a) Find the median x_m of T. (* An approximate median will suffice *)

   (b) Let $T_{left} = \{x \in T | x < x_m\}$ and $T_{right} = \{x \in T | x_m < x\}$. Let $T_1 = T_{left} - \{x | x = x_m\}$ $T_2 = T_{right} - \{x | x = x_m\}$.

   (c) If T_1 is not empty then ParMset1(T_1).

   (d) If T_2 is not empty then ParMset1(T_2).

5.1 Analysis

The following results for *extremal-selection*, *approximate-median* and *approximate compaction* will be used for analyzing the previous algorithm in the CRCW PRAM.

Lemma 5.1 *There is a CRCW algorithm that finds an element with rank k such that $\frac{n}{3} \leq k \leq \frac{2n}{3}$ deterministically in $O(\log\log n)$ time using $(n/\log\log n)$ processors [13]. Extremal selection can also be done in the same bounds.*

Compacting m non-zero elements in an array of size at most $\lambda \cdot m$ for some constant $\lambda > 1$ is called *approximate compaction* that can be used for load-balancing.

Lemma 5.2 ([13]) *Approximate compaction can be done deterministically in $O(\log\log n)$ steps using $n/\log\log n$ CRCW processors.*

Let N_i denote the total size of the sub-problems after the ith stage and let n_t, $1 \le t \le h$ be the distribution vector. **Claim** In Algorithm **ParMset1**,
(i) The total size of the sub-problems can be bound by $n/\log n$ after $O(\log h + \log\log n)$ stages.
(ii) $\sum N_i \le O(\sum_{t=1}^{h} n_t \log(n/n_t))$.

Proof: In the i^{th} stage, the maximum size of a sub-problem is $< n/c^i$, for some constant $c > 1$. Thus total size of sub-problems $< \frac{n}{c^i} h < n/\log n$ after $O(\log h + \log\log n)$ stages. Part (ii) Follows from the sequential bound in Section 3. $\square$

The total time spent before the problem is directly solved is $O(\sum_i N_i/p + \log\log n(\log h + \log\log n))$ using Brent's slow-down technique with work redistribution. From Lemma 5.1 an approximate median can be computed in linear work and $O(N_i/p + \log\log n)$ time. Hence the running time is bounded by $O(\log\log n \cdot (\log h + \log\log n) + \sum_i N_i/p + \log n)$ steps.

Theorem 5.1 *Multiset sorting problem can be solved deterministically in time $O((\sum_{t=1}^{h} n_t \log(n/n_t)/p)$ with $p \le n/\log n \log\log n$ processors, where n is the size of the input set and h is the number of distinct elements in it.*

The algorithms **Max1** for planar vector maxima can be directly implemented in parallel in the same bounds as the algorithm **ParMset1** for sorting multisets. Using Lemma 5.1 to find the median and the maximum we've the following result for the vector maxima problem.

Theorem 5.2 *The planar vector maxima problem can be solved deterministically in time $O(\sum_{t=1}^{h} n_t \log(n/n_t)/p)$ using $p \le n/\log n \log\log n$ processors, where n and h are the input and the output sizes respectively.*

Using the simplified algorithms in [5, 3, 26] that do not require the linear programming step, we obtain an identical result for planar hulls.

5.2 Expected bounds using randomization

By substituting the deterministic routines by their faster randomized counterparts we obtain faster algorithms (expected running times). In particular, we make use of constant-time algorithms for extremal selection and approximate-median ([24]) and $O(\log^* n)$ algorithms for approximate-compaction and *semisorting*.

Definition For all $n, m \in N$ and $\lambda \geq 1$, the *m-color semisorting* problem of size n and with slack λ is the following: Given n colors (integers) $x_1, \ldots, x_n$ in the range $0, \ldots m$, compute n nonnegative integers $y_1, \ldots, y_n$ (the placement of x_i) such that

(1) All the x_i of the same color are placed in contiguous locations (not necessarily consecutive).

(2) $\max\{y_j : 1 \leq j \leq n\} = O(\lambda n)$.

Lemma 5.3 ([1]) *There is a constant $\epsilon > 0$ such that for all given $n, k \in N$, the problems of* **semisorting** *and* **approximate compaction** *can be solved on a CRCW PRAM using $O(k)$ time, $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^\epsilon}$. Moreover, for $t \geq \log^* n$, it can be done in expected $O(t)$ steps using n/t processors.*

Then the total time spent before the problem is solved directly is $O(\sum N_i/p,\ \log^* n(\log h + \log\log n))$. For $p \leq n/\log^* n \log n$, the term $\log^* n(\log h + \log\log n)$ is subsumed in n/p. Hence we have the following

Theorem 5.3 *Multiset sorting problem can be solved in $O(\sum_{t=1}^{h} n_t \log(n/n_t)/p)$ steps with high probability with $p \leq n/\log^* n \log n$ processors, where n is the size of the input set and h is the number of distinct elements in it.*

Here *high-probability* refers to probability exceeding $1 - 2^{-n^\epsilon}$.

Theorem 5.4 *The planar convex hull and the planar maxima problems can be solved in $O(\sum_{t=1}^{h} n_t \log(n/n_t)/p)$ steps with high probability using $p \leq n/\log^* n \log n$ processors, where n and h are the input and the output sizes respectively.*

6 Optimal multiset sorting

In this section, we present a simpler and faster output-sensitive algorithm for sorting multisets in the CRCW PRAM. The optimality is achieved with respect to the number of distinct elements, h. This is the best possible if all the distinct elements have the same cardinality, i.e., $\Theta(n/h)$. The h distinct values will be referred to as *slabs* and the number of elements in slab t will be denoted by n_t. The algorithm is randomized and is described below.

Algorithm Mset2(T)

1. Choose a random sample R of size $\sqrt{n}$ and sort it.

2. Delete the duplicate elements. Denote the resulting set by R^*. This partitions the input set into $O(h)$ partitions.

3. Delete an element of the input set T in slab t if an element from slab t has been picked up in the sample . Denote the resulting set by T^*.

4. Sort T^*. Delete the duplicate elements and denote the resulting set by T^{**}.

5. Merge R^* and T^{**}. Denote the resulting set by S.

6. Label each element of T by its rank in S and apply $h-$color semisorting ([1]).

6.1 Analysis

The following result due to Hagerup[Hag92] is useful in our analysis.

Lemma 6.1 *For every fixed $\epsilon > 0$, the 1's in an array A can be compacted, in constant time, into an array of size $k^{1+\epsilon}$ using $O(n)$ CRCW PRAM processors, where n is the size of the array A and k is the number of 1's in it.*

Assume that we have a linear number of processors. Assume $h = O(n^\delta)$ for an appropriate $\delta > 0$, for otherwise the problem can be solved in $O(\log n) = O(\log h)$ time.

A sample of size $\sqrt{n}$ can be sorted in constant time with n CRCW processors ([12]). The sample is compressed, that is duplicate elements are deleted, by an application of compaction. Using Lemma 6.1, the sorted sample can be compacted into an array of size $O(h^{1+\epsilon})$ for every fixed $\epsilon > 0$ in constant time. Thus the size of R^* is $O(h^{1+\epsilon})$. Therefore checking if an element from slab t has been picked up in the sample or not can be done in $O(\log h)$ time. Call an element of T in slab t *active* if no element from slab t was picked up in the sample.

Claim 6.1 *The number of active elements in T is $O(h\sqrt{n}\log^2 n)$ with high probability.*

Proof: Probability that none of the elements in slab t is picked up in the sample is $(1-1/\sqrt{n})^{n_t}$. Probability that none of the elements in slab t is picked up and $n_t \geq \sqrt{n}\log^2 n$ is $\leq 1/n^2$, giving us the required result. □

Thus for every $\epsilon > 0$, the active elements of T can be compacted into an array of size $O((h\sqrt{n}\log^2 n)^{1+\epsilon})$ in constant time. For $h = O(n^\delta)$, $1/2 > \delta > 0$, size of T^* can be bounded by $O(n^{\epsilon'})$ for some $1 > \epsilon' > 0$ with high probability. Therefore, T^* can be sorted and compacted into T^{**} in constant time with high probability. Merging can also be done in $O(\log h)$ time. Semisorting takes $O(\log^* n)$ time. Hence, we have the following

Theorem 6.1 *The multiset sorting problem can be solved in $O(\log h + \log^* n)$ time with high probability with n processors, where n is the size of the input and h is the number of distinct elements in it.*

Remark Given the $\Omega(\log^* n)$ lower bound for approximate compaction [18] in the CRCW PRAM, $O(\log h + \log^* n)$ is an optimal time bound for Multisorting.

A closely related problem called *Renaming* involves determining the rank of each element, where the rank of an element is determined by the number of *distinct* elements less than the element. Farach and Muthukrishnan [8] described an $O(\log h)$ time $O(n \log h)$ work randomized algorithm for this problem which implies an algorithm for multisorting with bounds matching our result.

With $p = n/\log^* n$ processors and redistributing processors evenly our algorithm runs in $O(\log^* n \log h)$ time with high probability. Hence we have the following

Theorem 6.2 *The multiset sorting problem can be solved in $O(\log^* n \log h)$ time with high probability with $n/\log^* n$ CRCW PRAM processors, where n is the size of the input and h is the number of distinct elements in it.*

7 Concluding Remarks

There is a natural extension of the notion of distribution to three dimensions but the present techniques fall short of giving any improved results. Moreover, our definition of distribution appears more suited to certain algorithms. Is there a more natural way to define distribution for configurations of point sets ?

In the context of lower bounds, it will be interesting extend the lower bound for maximal points to the algebraic computation tree without the assumption on witness. The counter-example does not rule out the possibility that one may be able to recover the information of witnesses in additional linear time.

Acknowledgement The first author would like to thank A.C. Yao for discussions and examples related to witness for non-maximal points in decision trees.

References

[1] H Bast and T Hagerup. Fast parallel space allocation,estimation and integer sorting. *Technical Report, MPI-I-93-123*, June 1993.

[2] M. Ben-Or. Lower bounds for algebraic computation trees. *Proc. of the 15th ACM STOC*, pp. 80–86, 1983.

[3] B. Bhattacharya and S. Sen. On a simple, practical, optimal, output-sensitive randomized planar convex-hull algorithm. *to appear in* Journal of Algorithms.

[4] T.M. Chan. Output-Sensitive Results on convex hulls, extreme points and related problems. ACM Symp. on Comput. Geom., 1995.

[5] Timothy.M.Y. Chan, Jack Snoeyink, Chee-Keng Yap. Output-Sensitive Construction of Polytopes in Four Dimensions and Clipped Voronoi Diagrams in Three. *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms* 1995, pp 282-291.

[6] Kenneth L Clarkson and Peter W Shor. Applications of random sampling in computational geometry ii. *Discrete Comp. Geom.*, 4:387–421, 1989.

[7] M. Dyer. Linear time algorithms for two and three variable linear programs. *SIAM Journal on Computing*, 13(1), pp. 31–45, 1984.

[8] Farach and Muthukrishnan. Optimal parallel randomized renaming. *Information Processing Letters*, pp. 12–15, Jan 1997.

[9] R L Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Proc. Lett.*, 1:132–133, 1972.

[10] N. Gupta and S. Sen. Optimal, output-sensitive algorithms for constructing planar hulls in parallel. *Computational Geometry: Theory and Applications*, 8, 151–166, 1997.

[11] T. Hagerup. Fast deterministic processor allocation. *Proc. of the 4th ACM Symposium on Discrete Algorithms* 1993, pp. 1–10.

[12] T. Hagerup and R. Raman. Waste makes haste: Tight bound son loose parallel sorting. *Proc. of Foundations of Computer Science*, 1992, pp. 628 – 637.

[13] T. Goldberg and U. Zwick. Optimal Deterministic Approximate Parallel Prefix Sums and Their Applications. In *Proc. Israel Symp. on Theory and Computing Systems (ISTCS'95)*, (1995), pp. 220-228.

[14] S. Kapoor and P. Ramanan. Lower bounds for maximal and convex layer problems. *Algorithmica*, pages 447–459, 1989.

[15] D G Kirkpatrick and R Seidel. The ultimate planar convex hull algorithm. *SIAM Jl. of Comput.*, 15(1):287–299, Feb. 1986.

[16] D G Kirkpatrick and R Seidel. Output-Size sensitive algorithms for finding maximal vectors. *Proc. of ACM Symp. on Computational Geometry* 1985, pp. 89–96.

[17] H.T. Kung, F. Luccio and F.P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22, pp. 469–476, 1975.

[18] P. D. MacKenzie. Load balancing requires $\Omega(\log *n)$ expected time. *Symp. on Discrete Algorithms*, pp. 94–99, 1992.

[19] N. Meggido. Linear time algorithms for linear progamiing in R^3 and related problems. *SIAM Journal on Computing* 12(4), pp. 759–776, 1983.

[20] I.J. Munro and V. Raman. Sorting multisets and vectors In-place. *Proc. of the 2nd WADS*, LNCS 519, pp. 473–479, 1991.

[21] I. Munro and P.M. Spira. Sorting and Searching in Multisets. *SIAM journal on Computing*, 5(1), pp. 1–8, 1976.

[22] F P Preparata and S J Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, 20:87–93, 1977.

[23] F P Preparata and M I Shamos. *Computational Geometry : An Introduction.* Springer-verlag, New York, 1985.

[24] S. Rajasekaran and S. Sen. *Random sampling Techniques and parallel algorithm design.* J.H. Reif editor. Morgan, Kaufman Publishers, 1993.

[25] L.M. Wenger. Sorting a linked list with equal keys. *Information Processing Letters*, 15(5), pp. 205–208, 1982.

[26] R. Wenger. Randomized Quick Hull. *to appear in* Algorithmica.

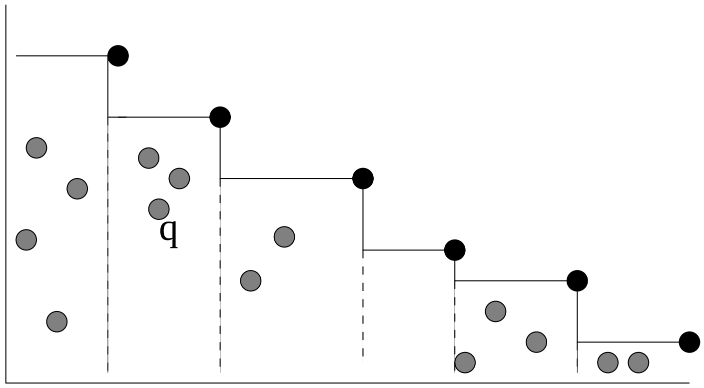

Figure 3: Distribution vectors for 2-D Maximal vectors is $\{3, 4, 1, 3, 4, 5\}$. h = 6 and slab(q) = 5.

[27] A.C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28, pp. 780–787, 1981.

[28] A.C. Yao. personal communication.

[29] J. Steele and A.C. Yao. Lower bounds for algebraic decision trees. *Journal of Algorithms* 3: pp. 1–8, 1982.

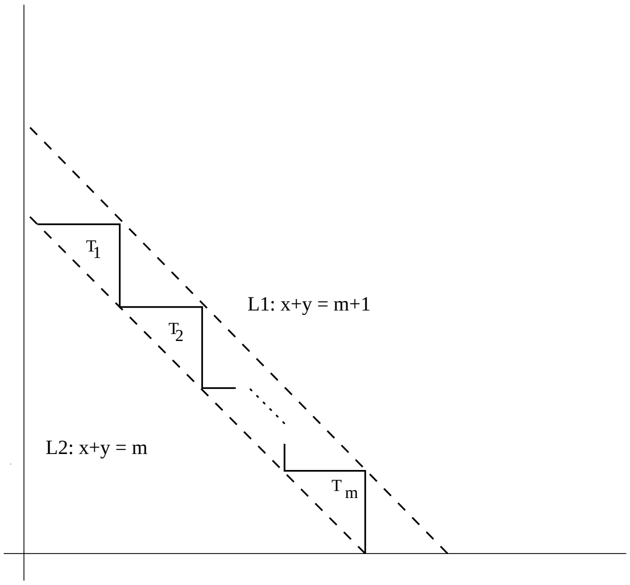

Figure 4:

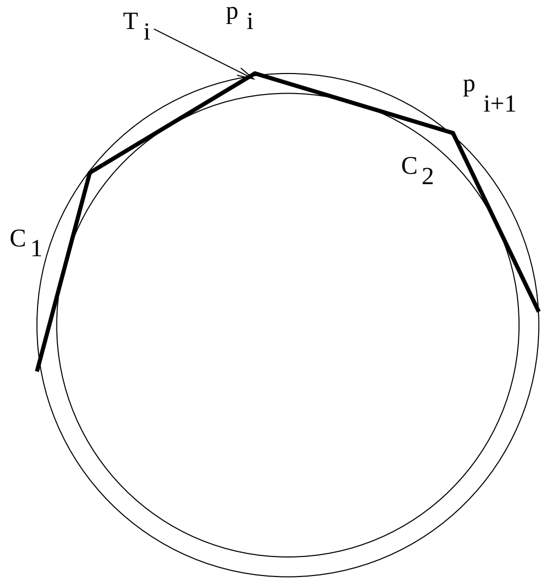

Figure 5: A regular h-gon with circumscribing circle C_1 and inscribing circle C_2. The remaining points q_j lie in one of the wedges (T_i is a wedge for h_i) according to the distribution vector.