Space-time trade-offs for some ranking and searching queries

Adrian Dumitrescu
Applied Mathematics and Statistics
SUNY Stony Brook
e-mail: dumitres@ams.sunysb.edu

William Steiger
Computer Science
Rutgers University
e-mail: steiger@cs.rutgers.edu

Abstract

We study space/time tradeoffs for querying some combinatorial structures. In the first, given an arrangement of n lines in general position in the plane, a query for a real number t asks about $Rank(t)$, the number of vertices of the arrangement with x-coordinates $\leq t$. We show that for $K = O(n/\log n)$, after a preprocessing step that uses space $S = O(n^2/(K \log K))$ the query can be answered in time $O(n \log K)$. The second query involves the Cartesian sum of vectors $\underline{a} = (a_1, \ldots, a_n)$ and $\underline{b} = (b_1, \ldots, b_n)$. For a given real t, it asks about $Rank(t)$, the number of sums $a_i + b_j$ which are $\leq t$. We show that for some positive constant c and $K \leq c(\log n)/(\log \log n)$, after a preprocessing step that uses space $S = O(n^2/K^2)$, the query may be answered in time $O((n/K) \log K)$. Both results fit neatly between two obvious extremes.

Keywords: Algorithms; Ranking; Searching; Computational geometry.

1 Introduction

We examine two problems. In the first, given lines $\ell_1, \ldots, \ell_n$, $\ell_i = \{(x, y) : y = m_i x + b_i\}$, we say they are in general position if the set $V = \{\ell_i \cap \ell_j, i < j\}$ of vertices has $N = \binom{n}{2}$ distinct elements. We write $y_i(t) = m_i t + b_i$ for the intercepts of the lines on the vertical line $x = t$. We consider the following two queries. Given a real number t, the ranking query asks for $Rank(t)$, the number of vertices in the left half-plane $x \leq t$. The search query asks whether there is a vertex in V with x-coordinate equal to t.

To answer such queries we are allowed to preprocess the lines and store any computed results using space S. The goal now is to be able to answer ranking queries as efficiently as possible, say in time $T(S)$. At one end of the spectrum we use minimal space $S = O(n)$ with which we store the data (m_i, b_i) in the order of decreasing slopes. Then we answer the query for t in time $T = O(n \log n)$ by counting $I(t)$, the number of inversions in the permutation that sorts $y_i(t)$, the intercepts at $x = t$. It is familiar [3] that $I(t) = Rank(t)$. At the other extreme we use space $S = O(n^2)$ to store the x-coordinates from V in increasing order, say $x_1 \leq \cdots \leq x_N$. The query about $Rank(t)$ can now be answered by binary search in $T = O(\log n)$ time. One question [6] is whether, using space $S = o(n^2)$, the ranking query can be answered in time $T = o(n \log n)$; i.e., is there anything between the extremes? In the next section we show how to answer queries in time $T = O(n \log K)$, only using space $S = O(n^2/(K \log K)$, for $K = O(n/\log n)$, and thus give an affirmative answer as long as $K = \Theta(\log n)$. For this purpose we store partial order information of the lines at an equally spaced sample of all the x-coordinates of the intersection points. On the other hand the trade-off is quite incomplete, and it is even open as to whether space $S = O(n^{2-\epsilon})$

(with $0 < \epsilon < 1$) gives any advantage over space $S = O(n)$ in settling ranking queries. These results extend in a straightforward way to the search version.

In the second problem we are given vectors $\underline{a} = (a_1, \ldots, a_n)$ and $\underline{b} = (b_1, \ldots, b_n)$ and we consider the Cartesian sum matrix $\Sigma = (\sigma_{ij})$, $\sigma_{ij} = a_i + b_j$. The ranking query for t asks for $Rank(t)$, the number of sums in Σ that are $\leq t$, and the searching query asks whether there is a pair (i, j) with $a_i + b_j = t$. We preprocess $\underline{a}$ and $\underline{b}$ and store any desired results using space S. Now, given a query t, we want to compute $Rank(t)$ as efficiently as possible, say in time T. In one extreme, using space $O(n)$ in which we store the elements of $\underline{a}$ and $\underline{b}$, each in increasing order, we can easily count $Rank(t)$ in time $T = O(n)$. At the other extreme, by computing all elements of Σ and storing them in sorted order using space $S = O(n^2)$, the queries can be settled in time $T = O(\log n)$. A natural question [2] is whether there is anything between these extremes; i.e., whether using space $S = o(n^2)$, we may answer a query in time $T = o(n)$. In Section 3 we show that after preprocessing the Cartesian sum and using space $S = O(n^2/K^2)$, the query may be answered in time $T = O((n/K)\log K)$ as long as $K \leq c(\log n)/(\log\log n)$, for some positive constant c. Thus we give an affirmative answer by selecting $K = c(\log n)/(\log\log n)$. The question of whether $S = O(n^{2-\epsilon})$ (with $0 < \epsilon < 1$) allows $T = o(n)$ is open.

2 The rank/search query for line arrangements

Given lines $\{\ell_1, \ldots, \ell_n\}$, $\ell_i = \{(x, y) : y = m_i x + b_i\}$, let $V = \{\ell_i \cap \ell_j\}$ denote the vertex set and $X = \{x_1, \ldots, x_N\}$, the x-coordinates of the $N = \binom{n}{2}$ vertices, in sorted order. We write $y_i(t) = m_i t + b_i$ for the intercepts of the lines on the vertical line $x = t$.

We begin by observing that if the space is to be $o(n^2)$, we could only compute and save a sparse subset of $x_1 \leq \cdots \leq x_N$. So take an integer K and suppose that preprocessing computes

$$U = \{u_i = x_{iK}, i = 1, \ldots, N/K\},$$

the x-coordinates of the stored vertices. Now, given a query t, we could find the largest $j : u_j \leq t$ in $T = O(\log(N/K))$ and know that

$$Rank(t) = jK + |\{x_i \in [u_j, u_{j+1}) : x_i \leq t\}|.$$

Moreover we could count $|\{x_i \in [u_j, u_{j+1}) : x_i \leq t\}|$ in time $T = O(K \log n)$ using the Bentley-Ottman line sweep [1], so if we want $T = o(n \log n)$, we would need $K = o(n)$.

On the other hand, the line-sweep algorithm requires the sorted order of intercepts at u_j. To sort $\{y_i(u_j), i = 1, \ldots, n\}$ once the query is presented would already take too much time, so we would need to be able to obtain this sorted order from the preprocessing data in $o(n \log n)$. Finally we observe that we could not store the N/K permutations in $S = o(n^2)$ because of the above requirement on K. Our solution is to compute and store the count at a sparse set of vertices, and a partial order information on the set of lines at an even sparser (sub-linear) set of vertices.

Let $K = O(n/\log n)$ be an integer parameter to be specified later, and for simplicity assume that $K|n$. For a positive integer d, a *d-sample* of X is the (sorted) sublist x_{id}, $i \geq 1$. Given integers $\delta < \Delta$, assume that $\delta|\Delta$ and $\Delta|n$. If we have a δ-sample and a Δ-sample of X, the δ-sample forms a finer subdivision of the set X, and the Δ-sample is a subset of the δ-sample.

Following Cole et. al. [3], a K-*grouping* of the lines $\{\ell_1, \ldots, \ell_n\}$ at $x = u$ is a permutation π_u of $[n]$ such that for $1 \leq i < j \leq n/K$, $1 \leq r, s \leq K$,

$$y_{\pi_u((i-1)K+r)}(u) \leq y_{\pi_u((j-1)K+s)}(u);$$

that is π_u partitions the lines into groups of size K, and for $1 \leq i < j \leq n/K$, all y-coordinates of the lines in group i at $x = u$ are smaller or equal than the y-coordinates of the lines in group j at $x = u$. Write $u_i, i \geq 1$ for the points of the Δ-sample of X, and $v_j, j \geq 1$ for the points of the δ-sample of X. We take $\Delta = \Delta(u_i) = nK \log K$ and $\delta = \delta(v_j) = K \log K$.

At each point of the Δ-sample store a K-grouping (of size n) of the lines at that point (the sorted order is a possible choice). At each point of the δ-sample store the number of intersection points to the left of that point. The space used by the Δ-sample and the δ-sample are

$$\frac{N}{\Delta} n = O\left(\frac{n^2}{K \log K}\right) \quad \text{and} \quad \frac{N}{\delta} 1 = O\left(\frac{n^2}{K \log K}\right),$$

respectively. So the total space is $O(n^2/(K \log K))$. The following Lemma will be useful for our algorithm. It shows that the information about the vertical ordering of the lines at a sample point $x = v$ can be obtained quickly from the partial order information at a nearby sample point $x = u$.

Lemma 2.1 *Consider an interval $[u, v]$ on the x-axis such that the vertical strip $u \leq x \leq v$ contains at most $nK \log K$ vertices in V. Given a K-grouping of the lines at $x = u$, one can compute the (exact) order of the lines at $x = v$ in $O(n \log K)$ time.*

Proof. This is a modification of a basic element of [3], with a much simpler proof and may be of independent interest. The algorithm has two phases. In the first phase, we maintain a grouping of the lines at $x = v$ as a linked list of pointers to groups. Let L_2 denote this list. Each group is stored in an array of size $2K$ and has a size between K and $2K$ during this phase. Elements in each group are the y-coordinates of the lines in that group at $x = v$. The y-coordinate of the lowest line in the group (at $x = v$) is stored in the first position of its array (called the minimum of that group). In the second phase the exact order of the lines at $x = v$ is computed from the previous representation.

First phase scans the lines in the order given by the K-grouping at $x = u$. Let L_1 denote this list. The first group of L_1 generates the first group of L_2 (the y-coordinates at $x = v$ of the K lines in the first group of L_1). At the current step, a line of L_1 is inserted into the appropriate group of L_2, and assume a list of groups L_2 is available at $x = v$. When inserting in L_2 the current line l of L_1, its y-coordinate at $x = v$ is compared to the y-coordinate of the lowest line in the last group of L_2 (having the largest minimum). Then the search moves sequentially towards the head of the list L_2, until a proper group is found to accommodate the new line. The line l (its y-coordinate at $x = v$) is inserted in the first free position (end of group) in the first group which was found having a smaller minimum value. If the search reaches the first group of L_2, l is inserted there and the minimum is recomputed (in constant time). When a group becomes too large (i.e. has size $2K$), it is split into two groups of size K (a new group being created), which are relinked at the same position in L_2. This is done by selecting the median in the large group and moving the larger half (K keys) to the first half of a new array of size $2K$.

The second phase computes the exact order of the lines at $x = v$, from the list of groups in L_2 (of sizes between K and $2K$), by sorting each group and appending the results.

To analyze the complexity of the first phase, we account for the time to split large groups when creating new groups and for the time to search for the correct group when inserting new lines at $x = v$. There are at most $O(n/K)$ groups created and each takes $O(K)$ (the linear time selection algorithm is employed). This amounts to $O(n)$. To account for the search time, assume that c_l comparisons have been made to locate a group when inserting the current line l at the new position

$x = v$. Then at least $c_l - 1$ groups of size at least K lines have been skipped. Write i for the index of the line in the K-grouping at $x = u$, and write j for the index of the line in the list obtained by concatenating all previous groups (from L_2) and the located group (from L_2) at that time. Then $i - j \geq (c_l - 1)K$, hence l has at least $(c_l - 1)K$ intersections with other lines of smaller index $k < i$ at $x = u$ (see Figure 1). We assumed that there are at most $nK \log K$ vertices in the strip $u \leq x \leq v$, which implies

$$\sum_l (c_l - 1)K \leq nK \log K.$$

This bounds the complexity of the search, because

$$\text{COST} = \sum_l c_l \leq n \log K + n.$$

The second phase takes $(n/K)O(K \log K) = O(n \log K)$ time. Therefore the total complexity of the algorithm is $O(n \log K)$ which proves the Lemma. □

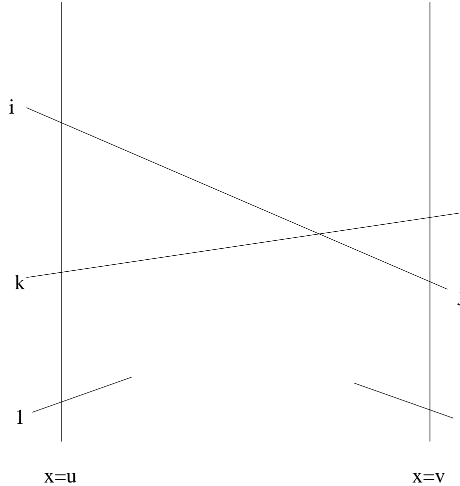

Figure 1: Counting intersections in a vertical strip

The query algorithm is described next.

1. Binary search with the query value t for the Δ-interval $[u_i, u_{i+1})$ and the δ-interval $[v_j, v_{j+1})$ containing t. Note that the δ-interval is a subinterval of the Δ-interval. This step takes $O(\log n)$ time.

2. Compute the (exact) ordering of the lines at $x = v_j$ from the ordering at $x = u_i$ in time $O(n \log K)$ (as described in Lemma 2.1).

3. Perform a sweep with a vertical line from $x = v_j$ to $x = t$ using the line sweep algorithm of Bentley and Ottmann [1] (see also [5]).

The number of intersection points k in the vertical strip $v_j \leq x \leq t$ satisfies $k \leq K \log K$. Since the time to initialize the sweep is $O(n)$ and processing each event during the sweep takes

$O(\log n)$, the vertical sweep line reaches $x = t$ in $O(n + k \log n) = O(n + K \log K \log n)$ time. For $K = O(n/\log n)$, this becomes $O(n + (n/\log n)(\log n)(\log K)) = O(n \log K)$. Therefore the total time to process a query is $O(n \log K)$, and we proved

Theorem 2.2 *Using preprocessing space $O(n^2/(K \log K))$, the ranking query for line arrangements can be answered in time $O(n \log K)$, K an integer satisfying $K = O(n/\log n)$.*

Write $S(n)$ for the space and $T(n)$ for the query time. Then $S(n)T(n) = O(n^3/K)$. In particular, by selecting $K = \log n$, we obtain

$$S(n) = O\left(\frac{n^2}{\log n \log\log n}\right) = o(n^2), \quad T(n) = O(n \log\log n) = o(n \log n).$$

Remark. It would be interesting to decide if using space $S(n) = O(n^{2-\epsilon})$, a query can be answered in time $T(n) = o(n \log n)$, for some fixed $\epsilon > 0$. More interesting from a practical point would be to obtain $T(n) = o(n)$ with space $S(n) = o(n^2)$.

3 The rank/search query for Cartesian sums

For two vectors $\underline{a} = (a_1, \ldots, a_n)$ and $\underline{b} = (b_1, \ldots, b_n)$, the Cartesian sum is the set of the n^2 sums

$$\Sigma = (\sigma_{ij}), \;\; \sigma_{ij} = a_i + b_j.$$

Assume that both vectors are in sorted order. For the ranking query we want to count $Rank(t) = |\{\sigma_{ij} \in \Sigma : \sigma_{ij} \le t\}|$ using storage S, and in time $T(S)$. It will be useful to describe the algorithm for the case when $S = O(n)$. Call it **ALG A:** We start with RANK$=0$, $i = n$ and $j = 1$. In each step, compare σ_{ij} with t. If $\sigma_{ij} \le t$, we add i to COUNT and increase j by one; otherwise we decrease i by one. The process continues until $j < 1$ or $i > n$. Clearly it uses at most $2n-1 = O(n)$ comparisons.

To use $S = o(n^2)$ storage we will choose an integer $K > 0$, $K|n$, and partition Σ into $(n/K)^2$ square submatrices Σ_{rs} of size K^2 each, where for $1 \le r, s \le n/K$,

$$\Sigma_{rs} = \{\sigma_{ij} : (r-1)K < i \le rK, \; (s-1)K < j \le sK\}.$$

If the sorted order of the elements in Σ_{rs} were known, the integer

$$\mathrm{COUNT}(r, s) = |\{\sigma_{ij} \in \Sigma_{rs} : \sigma_{ij} \le t\}|$$

could be computed in time $O(\log K)$ using binary search on an array of size K^2. The following algorithm to answer the ranking query is based on **ALG A**, above.

Start with RANK$=0$, $r = n/K$, and $s = 1$. In each step we compare $\sigma_{rK,sK}$ to t. If $\sigma_{rK,sK} \le t$ we add rK^2 to RANK and increase s by one; otherwise we add COUNT(r, s) to RANK and decrease r by one. The process continues until $r < 1$ or $s > n/K$. The complexity is $O(n/K)$ comparisons and $O(n/K)$ calls of COUNT(r, s) which, assuming Σ_{rs} is sorted, gives

$$T = O((n/K) \log K).$$

We use a coding device to enable us to run COUNT(r, s) in time $O(\log K)$ at query time. In preprocessing, for each of the $(n/K)^2$ submatrices Σ_{rs} of size K^2, we store a pointer to an array

with K^2 pairs of indices, each with values in $\{1, \ldots, K\}$. These pairs describe the sorted order of the values in the corresponding sub-matrix Σ_{rs}. For example, $(1,1),(1,2),(2,1),(2,2)$ says that in the 2 by 2 matrix Σ, $\sigma_{11} \leq \sigma_{12} \leq \sigma_{21} \leq \sigma_{22}$. The set of arrays of size K^2 can be stored in a table M with R rows, where $R = O(K^{8K})$ (there are at most this number of distinct orderings of the K^2 elements in Σ_{rs}, see [4]). The space taken to store M is $\leq 2K^2R = O(K^{8K+2})$. We store a pointer into M for each Σ_{rs}, so the total space for the list of pointers and the table is

$$(\frac{n}{K})^2 + O(K^{8K+2}).$$

For some appropriate constant $c > 0$, as long as $K \leq c(\log n)/(\log\log n)$, the space is bounded by $O(n^2/K^2)$, and we proved

Theorem 3.1 *Using preprocessing space $S = O(n^2/K^2)$, the ranking query for Cartesian sums can be answered in time $T = O((n/K)\log K)$, K an integer satisfying $K \leq c(\log n)/(\log\log n)$, for some suitable positive constant c.*

Write $S(n)$ for the space and $T(n)$ for the query time. Then $S(n)T(n) = O((n/K)^3 \log K)$. In particular, by selecting $K = c(\log n)/(\log\log n)$, we obtain

$$S(n) = O\left(\frac{n^2(\log\log n)^2}{(\log n)^2}\right) = o(n^2), \quad T(n) = O\left(\frac{n(\log\log n)^2}{\log n}\right) = o(n).$$

Remark. One can take $c = 1/5$ in the above calculation.

References

[1] L. Bentley, T. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Transactions on Computing* **28** (1979), 643-647.

[2] B. Chazelle, Personal communication.

[3] R. Cole, J. Salowe, W. Steiger, E. Szemerédi, An Optimal Time Algorithm for Slope Selection, *SIAM Journal on Computing* **18** (1989), 792-810 .

[4] M. Fredman, How good is the information theory bound in sorting?, *Theoretical Computer Science* **1** (1976), 355-361.

[5] F. Preparata, M. Shamos, *Computational Geometry, An Introduction*, Springer, New York, 1985.

[6] M. Sharir, Personal communication.