T U M

I N S T I T U T F Ü R I N F O R M A T I K

Binary Search on Two-Dimensional Data

Riko Jacob

TUM-I0821
Juni 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I0821-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Binary Search on Two-Dimensional Data

Riko Jacob

Institut für Informatik, Technische Universität München
D-80290 München

Abstract

We consider the problem of searching for the predecessor (largest element that is smaller than some key) among elements that are organized as a matrix such that every row and every column is weakly monotonic (sorted matrix). The results are matching upper and lower bounds for the number of accesses to the matrix and comparisons with the key.
Keywords: sorted matrix, binary search
ACM Computing Classification System (1998): F.2.2

1 Introduction

It is well known that the complexity of searching for the predecessor of some key in a set of size n heavily depends on how the set is organized: If it is sorted, it can be done in $\log n$ steps, if nothing is known about the set, it will take n steps. Here, we consider a setting where the set is organized in a matrix (two dimensional array) such that each column and each row is sorted. A generalization of such matrices are known in the literature as "Young Tableaux", and there is a rich literature on the topic, see for example the textbook by Fulton [2]. Usually, such tableaux are used to represent elements of an algebra, and the interest is in the structure of some operations acting on the tableaux. In contrast, here we are considering a single tableau in which we search a predecessor as efficiently as we can.

This questions originates in the work of Matthias Altenhoefer and Sven Oliver Krumke in the context of parametrized search for flow computations. There, an access to the oracle corresponds to a max-flow computation on G, and the matrix has one row and column per edge of G.

A variant of the problem, where oracle questions are cheap (as expensive as matrix accesses) has been considered in [1].

1.1 Setting

Here, we assume that the entries of the matrix stem from an ordered universe, and that the only operation allowed on such elements is the comparison. The indices necessary to describe positions of the matrix are natural numbers. We assume that such numbers (up to $n \cdot m$, the size of the matrix) can be added and compared at unit cost.

We count the indexed accesses to the matrix separately. Further, comparisons involving the element for which we search the predecessor are counted as oracle questions.

Monotone Matrix Search

Given A real (totally ordered universe) $m \times n$ matrix A, where the values are sorted within a column and sorted within a row. More precisely, we have that $i \leq i'$ and $j \leq j'$ implies $a_{ij} \leq a_{i'j'}$.
A monotone oracle B; internally, B contains a value b, and a question a to the oracle gives the yes/no answer to $a < b$

Output A largest entry of A for which the oracle answers yes, i.e., an $a_{ij} \leq b$ such that no $a_{i'j'}$ exists with $a_{ij} < a_{i'j'} < b$.

One algorithm to solve a Monotone Matrix Oracle is the following: Sort the $n \cdot m$ entries of the matrix A in time $O(nm \log nm)$, then use the oracle B to perform a binary search. This algorithm accesses $n \cdot m$ elements of the matrix, and uses the oracle $\lceil \log n \cdot m \rceil$ times. In the following, we want to reduce the number of accesses to the matrix without significantly increasing the number of questions to the oracle.

semi-Monotone Matrix Search is an Monotone Matrix Search, where only the columns are sorted. Here it is convenient to allow different sized columns (there is no connection between columns anyway). In that situation the vector $(m_1, \ldots, m_n)$ gives the size (length) of the different columns.

f, ℓ **Matrix Oracle** An $n \times n$ Matrix given by $a_{ij} = (f + i\ell)/j$. After mirroring around a vertical axis, this matrix is strictly monotone in the above sense.

2 Auxiliary Algorithms

2.1 Weighted Median

The following result appeared in [3].

weighted median

Given n pairs (a_i, w_i)

Output a such that for $N = \sum_i w_i$ we have

$$\sum_{i:a_i<a} w_i \leq N/2$$

and

$$\sum_{i:a_i>a} w_i \leq N/2$$

Lemma 1 *weighted median can be computed in $O(n)$ time (assuming addition to be constant time).*

Proof. Find the unweighted median in $O(n)$ time. Check if it is too big or to small in linear time. Recurse with updated weight demand. □

3 Lower Bounds

3.1 Oracle

Lemma 2 *For every n, m, there is a Monotone Matrix Search matrix A such that for every algorithm there exists an oracle that is queried at least $\log n + \log m$ many times.*

Proof. Consider the matrix A with the entries $a_{ij} = i \cdot n + j$. Now, all the $n \cdot m$ different outcomes possible, depending on the oracle. The questions to the oracle that an algorithm asks on A gives rise to a binary decision tree, where each leaf is labeled with an entry of A. Hence, there must be $n \cdot m$ leafs, which is impossible if the algorithm asks always less than $\log n + \log m$ questions. □

3.2 Matrix Access

Antichain

Lemma 3 *There exists a $n \times n$ Monotone Matrix Search A such that any algorithm must access at least n entries.*

Proof. Define $M = n^2 + 2$. Consider the matrix A with entries $a_{ij} = i \cdot n + j$ if $j < n - i$, $a_{ij} = 2M + i \cdot n + j$ if $j > n - i$, and $a_{i,(n-i)} = x_i = M + i$. Take an oracle with value M. This matrix is clearly Monotone Matrix Search, and the correct answer is $a_{n-1,1} < M - 1$. Now, if an algorithm does not access one of the positions on the secondary diagonal, one of the values x_i is not read. Hence, the matrix A' that is like A and only x_i is changed to $M - 1$, will lead the algorithm to the wrong answer. Any algorithm must access at least all the n elements on the secondary diagonal of A. □

semi-Monotone Matrix Search

Lemma 4 *A semi-Monotone Matrix Search with chain-length $m_1, \ldots, m_n$ requires $\sum_{i=1}^{n}(1+\log m_i)$ accesses to the matrix by a deterministic algorithm in the worst case.*

Proof. Consider the following game where the oracle is fixed to 0 (and may be accessed by the algorithm without cost). The algorithm and the adversary take turns. The algorithm points at a position, the adversary announces some value for this position. With $m = \max\{m_1, \ldots, m_n\}$, the adversary may use integers in $[-m, m]$. The goal of the algorithm is to find in each column the position of the smallest positive number. The number of alternations the adversary can enforce gives a lower bound on the worst case number of accesses to the matrix by the algorithm.

The strategy of the adversary is the following: It "fills in" consecutive negative numbers from the bottom and consecutive numbers from the top. Initially, it places $-(m + 1)$ below, and $m + 1$ above. Now, if the algorithm asks a position that is not yet filled in, the adversary has the choice (within its self-given framework), to either extend the positive numbers or the negative numbers. It chooses whatever leaves more spaces open. Hence, the algorithm can at best force the adversary to reduce the number of open positions from m' to $\lceil (m' - 1)/2 \rceil$.

Hence, the biggest $m(i)$ for which an algorithm can get away with i questions is given by the recursion $m(1) = 1, m(i + 1) = 2m(i) + 1$, which has the solution $m(i) = 2^i - 1$. From this the statement of the lemma follows. □

Using Yao's Minimax-principle [5, 4, p. 35], Lemma 4 shows that the worst case running time cannot be improved by using randomization. This principle (together with some standard considerations) show that Lemma 4 remains valid for the expected execution time.

non-square Monotone Matrix Search

Lemma 5 *An $n \times m$ Monotone Matrix Searchwith $m \geq n$ requires $n \log \lfloor m/n \rfloor$ accesses to the matrix.*

Proof. If $m < 2n$ there is nothing to be shown. Otherwise, there is a secondary diagonal consisting of n pieces, each $\lfloor m/n \rfloor$ positions long, pieces non-overlapping. Place $-\infty$ in the positions below this, $+\infty$ above. On the diagonal, embed an arbitrary $n \times \lfloor m/n \rfloor$ semi-Monotone Matrix Search. By Lemma 4 the statement of the lemma follows. □

4 Monotone Matrix Search Algorithms

4.1 Algorithm 0 / Optimizing Matrix Accesses only

The following algorithm can be understood as a variant of one of the algorithms in [1].

Consider the situation of a square matrix A and $n = 2^k - 1$. Hence, there are k natural submatrices A_i, given by both indices being divisible by 2^{k-i} for $i \in \{1, \ldots, k\}$. We generalize the task in the following way: Instead of asking only for the largest element for which the oracle answers yes, we want in every row and in every column this element (in case of ties, the one with largest index). Actually, this is equivalent to finding the correct position of b in each row and each column. Now, assume this knowledge is already available for the submatrix of level i, and we want to create it for level $i+1$. To this end, we only need to access elements that could still be this boundary. Observe that every new element either is in an old row or column and has two old horizontal and vertical neighbors, or has two old diagonal neighbors. Because old rows, columns and diagonals are ordered and there the boundary is known, there can be at most one new element per old row, column, and diagonal. Hence there are at most $2^i + 2^i + 2 \cdot 2^i = 4 \cdot 2^i$ new elements to be accessed and tested in this round. Hence, this algorithm accesses in total $\sum_{i=1}^{k} 4 \cdot 2^i = 4 \cdot (2^{k+1} - 1) \leq 8n$ out of the n^2 elements of the matrix.

The number of questions to the oracle can be kept at $k^2 = \log^2 n$ by not asking every element individually. Instead, we collect all new elements of the matrix of one round, sort them, and perform a binary search on this list. Then, all necessary answers of the oracle can be deduced. The $O(n \log n)$ computation time of this step can be improved to $O(n)$ by replacing the sorting with repeated median finding.

4.2 semi-Monotone Matrix Search

Considering the columns of a semi-Monotone Matrix Search as individual tasks, it can certainly be solved with $n \log m$ oracle questions and matrix accesses. A first improvement is to proceed in rounds, where each column contributes with the current median of the remaining possible answers. Again, instead of asking all n oracle questions, the n values are sorted and a binary search with the oracle is performed, yielding all n answers. The total number of questions to the oracle is now $\log n \cdot \log m$.

The following algorithm reduces this to $O(\log n + \log m)$ questions to the oracle. The main idea is to let go the synchronization between the searches in the different columns. Then, in every column, at all times, the search can be characterized by 3 entries, namely the smallest one known to be larger than b, the largest one known to be smaller than b, and the middle position between the two, the element x_i for which the comparison with b (the question to the oracle) is pending (unless the column is finished). This identifies the number m_i of elements in this column that are still potentially the right answer. Now, the median x of the x_i weighed with m_i is computed and is used to question the oracle. If the oracle answers "yes" ($x < b$) this makes progress for all searches with $x_i \leq x$, if the answer is "no" it makes progress for all searches with $x_i \geq x$. If a search makes progress, the number m_i is halved. Because x is the weighted median, this progress happens for one half of the potential outcomes, and $\sum_i m_i$ is reduced by a factor $3/4$. Hence, the number of rounds and oracle questions is $O(\log(n \cdot m)) = O(\log n + \log m)$, and the number of accessed matrix elements is $O(n \log m)$, which is optimal.

4.3 square Monotone Matrix Search

Now, the ideas of Section 4.1 and 4.2 can be combined to an algorithm that on square Monotone Matrix Searchs performs $O(n)$ accesses to the matrix and $O(\log n)$ oracle questions.

Observe that we can identify new elements to be accessed in the matrix (and to schedule to implicitly check with the oracle) as soon as its neighbors one level higher are identified to be on the boundary. Define for every element that is accessed in Algorithm 0 in level i its weight to be $(2^{k-i})^2$. Similarly to Section 4.2, we keep a list of active nodes (for which the comparison with the oracle is pending), compute their weighted median, and query the oracle with this median. This resolves either all active nodes whose values are at least as large as the median, or at most as large as the median.

If an element a of level i is resolved (implicitly compared to the oracle), it gives rise to at most three new elements on level $i + 1$, namely one for the row, one for the column, and one for the diagonal. Because the weight of one such element is one quarter of the weight of a, exchanging a with its three replacements (follow-ups), still reduces the weight of active nodes by one quarter of the weight of a. Hence, every question to the oracle reduces half the weight of the active nodes by a factor $3/4$. Because initially there is one active node of weight n^2, the total number of queries to the oracle is $\log_{4/3} n^2 = O(\log n)$.

4.4 general Monotone Matrix Search

Assume w.l.o.g. $n > m$. Select n columns of A at distance roughly n/m, leading to the square submatrix A'. Solve Monotone Matrix Search on A', remembering the dividing line. Now, solve the remaining pieces of the rows as columns of a (transposed) semi-Monotone Matrix Search. In total, this gives $O(\log nm)$ questions to the oracle and $O(n\lceil\log(n/m)\rceil$ accesses to the matrix (both terms dominated by the second part).

4.5 f, ℓ Matrix Oracle

For the particularly structured case of a f, ℓ Matrix Oracle, a simpler algorithm achieves $O(n)$ matrix accesses and $O(\log n)$ oracle queries, asymptotically matching the Monotone Matrix Search case. For

non-square situations it achieves $O(\min(n, m))$ accesses to the matrix, improving over the general case.

Extend the matrix to a real valued function in the plane $(x, y) \mapsto (f + x\ell)/y$. The level curves of this function are straight lines of the form $y = f + x\ell$. All these straight lines meet in the point $(-f/\ell, 0)$. Hence, the overall task is to find the point with integer coordinates in $[1, n] \times [1, m]$ with highest slope from $(-f/\ell, 0)$ and positive oracle answer.

The algorithm first performs a binary search on the points of the form (i, m) and (n, i) (the totally ordered backmost boundaries). This leads to two points a, b on the backmost boundary of the square. Next, it performs another binary (perhaps exponential) search at a perpendicularly to the boundary. Now, the remaining cone from $(-f/\ell, 0)$ has within $[1, n] \times [1, m]$ the height and width at most 1. Hence, at most one element per column and per row (whatever is less) still needs to be considered. These elements can be computed with constantly many multiplications, divisions and +1 or -1.

5 Adaptivity to Short Antichain

In the above considerations, we analyzed the worst-case performance of the algorithms. It is possible to improve on this if the antichain of elements smaller than b is short.

It is possible, that for a sorted matrix A and an oracle b a very short proof of the correct output exists. For example, if the smallest entry of a row is actually larger than the largest value of the previous row ($a_{ij} = i \cdot n + j$). Then with the oracle $n + .5$ it is sufficient to evaluate $a_{1n} = n$ (showing that entries in the first row are at most n) and $a_{21} = n + 1$ (showing that the remaining matrix is at least $n + 1$).

More generally, a position (i, j) of A is called an upper witness if $b \leq a_{ij}$ and $a_{i-1,j} < b$, $a_{i,j-1} < b$. Symmetrically, a position with $a_{ij} < b$ and $b \leq a_{i+1,j}$, $b \leq a_{i,j+1}$ is called a lower witness. Obviously, any correct algorithm needs to access all upper and lower witnesses. Observe that the witnesses form a staircase (the boundary between smaller than b and larger than b), where right-turns are around lower witnesses and left-turns are around upper witnesses. Hence, the number of lower and upper witnesses can differ by at most one. Let p denote the number of witnesses. In the following, we discuss an algorithm whose number of accesses to the matrix depends on p in an asymptotically optimal way.

5.1 Adaptive Algorithm

The following algorithm is a modification of Algorithm 0. To show this analogy, define for an integer x the number $z(x)$ to be the number of trailing zeros in binary representation. Like in Algorithm 0 we consider coordinates with larger $z(x)$ first, mimicking a binary search.

- Maintain a region of possible witnesses, partitioned into L-shaped regions (alternating directions), with the invariant that in each L is at least one witness.
  Similar to the other algorithms, as described in detail below, from each L one entry of the matrix participates in a weighted median search that results in an oracle question comparing some of the entries with b.

- One L is specified by 3 x-coordinates x_1, x_2, x_3 and 3 y-coordinates y_1, y_2, y_3 (all integral). Coinciding lines are allowed (L is a rectangle).

Keep the invariant that for $x_1 < x < x_2$ we have $z(x) < \min\{z(x_1), z(x_2)\}$, and the analog statement for x_2, x_3, y_1, y_2 and y_2, y_3.
Different Ls overlap only at the side where the boundary enters and leaves, there the dividing lines (given by x_1 and y_3 or y_1 and x_3) are shared with the neighboring L. More precisely, the positions (x_i, y_j) are already compared to the oracle, such that the other boundaries are already known to not contain further witnesses. Hence, the area for a lower L (turning right) is $(x_3 - x_1 - .5) \cdot (y_3 - y_1 - .5) - (x_3 - x_2 + .5) \cdot (y_3 - y_2 + .5)$, an upper L analogously has area $(x_3 - x_1 - .5) \cdot (y_3 - y_1 - .5) - (x_2 - x_1 + .5) \cdot (y_2 - y_1 + .5)$. It is allowed that (x_2, y_2) is the only witness of an L.
Initially, there is one degenerated L consisting of the whole matrix; Assuming the dimensions of the matrix to be powers of two by filling in $+\infty$ or $-\infty$, this satisfies the invariants about $z(.)$ with $x_1 = x_2 = y_1 = y_2 = 0$, $y_3 = n$, $x_3 = m$.

- In every refinement step of a region double precision, i.e., choose $x_{12} = (x_1 + x_2)/2$, i.e., the x-value with largest $z(x)$ between x_1 and x_2. Choose x_{23}, y_{12}, and y_{23} analogously. This leads to 5×5 new probe positions, of which 7 are not already implicitly compared to the oracle. Points on the boundary might already be clear by previous questions of other Ls, other points might be clear by previous questions to the oracle. (All of this is ignored in the analysis.)

- Access and sort the entries at the probe positions, build the binary tree of height 3 that models a binary search for b in the probes. Following a path in this tree, the current probe participates in the median search (resulting in a question to the oracle) with weight $(3/4)^{\frac{i}{3}}$ times the area of the original L, where $i = \{0, 1, 2\}$ denotes the distance from the root in the binary search tree.

- Once a leaf of the decision tree is reached, the grid inside the L-region is completely evaluated. One possibility is that the L just shrinks in width. By the choice of new lines, the invariants are maintained.
The other possibility is that three or more (always an odd number) of new L-shaped regions (possibly degenerated) come into existence. Observe that there are at most 4 rectangles in a row or column (inside the L). Hence, there are at most two consecutive non-corner rectangles, such that these rectangles can be attached to Ls. If there is one such rectangle, one (arbitrarily) of the Ls is degenerated. Hence, only neighboring x-values and y-values are used to define new Ls. This maintains the invariant about $z(.)$.

5.2 Analysis

Oracle questions Define $\gamma = (3/4)^{\frac{1}{3}}$. The total virtual weight is the sum of the weights of elements participating in the median selection. Note that this is one point from every L. At the very first step, there is only one degenerated L with weight nm. Note that an L with weight < 1 cannot contain a point and has hence weight 0. Now, for every point that is implicitly answered by the oracle, the follow up weight (virtual area, perhaps summed up) is at most the current multiplied by $\gamma < 1$: During the binary search, this factor changes from γ^j to γ^{j+1}, when the L is split from γ^2 to $3/4 = \gamma^3$ or less.

Hence, every oracle question reduces the remaining total virtual weight by $(1 - \gamma)/2 > 0$. The factor is $\tau = 1 - (1 - \gamma)/2$, hence the task is finished after x oracle questions if the area is $nm\tau^x < 1$, $\log nm + x \log \tau < 0$, $x > -\frac{1}{\log \tau} \log nm$. Hence, at most $15 \log nm$ oracle questions suffice.

Matrix accesses Assume $m \geq n$. The following discussion assumes that p is a power of two. Consider running Algorithm 0. Annotate positions accessed by this algorithm with the round they are accessed in. After for $\log \frac{n}{p}$ rounds, the resulting region where further witnesses must be consists of $O(p)$ rectangles of size $\frac{n}{p} \times \frac{m}{p}$, and the total number of accesses to the matrix is $O(p)$.

Define the width of an L to be the smaller side of the corner-rectangle, i.e., for a right turning L $\min\{x_2 - x_1, y_2 - y_1\}$, and the relative width $v = \min\{\frac{x_2-x_1}{m}, \frac{y_2-y_1}{n}\}$. If $v > 1/p$, then at least one of the probe positions of the adaptive algorithm has a level smaller than $\log p$. Charge the constantly many accesses of refining this L to the probe with this smallest annotation.

Observe that the Ls used in the algorithm form a hierarchy (tree by inclusion, no subsequent L ever extends outside an ancestor). Mark all Ls that have relative width $v > 1/p$, but not all of their children have this. Then, the total access cost in marked Ls and their ancestors is $O(p)$ by comparison with Algorithm 0, and the marked Ls form an alternating staircase from lower left to upper right.

Say that an L passes its name on to its leftmost child, the other children are said to be created in the step. Observe that one L (identified by its name over time) will use at most logarithmic in its area number of rounds and accesses to the matrix before it identifies a single witness. In this way, we can account the matrix accesses to certain witnesses, namely, the overall number of accesses within marked Ls is the sum over all witnesses, taking the logarithm of its Ls. More precisely, focus on the Ls with a lower witness (turning right), the other Ls yield the same bound. Note that no two such Ls contain the same x- or y-coordinate. Denote by $(X_i, Y_i, b_i)_{i=1\ldots k}$ the dimensions $(x_3 - x_1)$ and the number of points of the staircase formed by the marked lower witness Ls. Now we have $\sum X_i = m$, $\sum Y_i = n$, $\sum b_i \leq p/2 + 1$, and the total access inside this Ls is bounded by $O(\sum_{i=1}^{k} b_i \log X_i Y_i)$. Observe that the number of witnesses per L is between 1 and its width. Hence, the number of Ls is $k \leq p$ and $k \cdot m/p \geq p$, i.e., $k \geq p^2/m$. Now, we want to find an upper bound for $w = \sum_{i=1}^{k} b_i \log x_i$. To this end, observe that w cannot decrease if we shift some weight (b_i) from a small to a larger or equal x_j. Hence, as long as there are b_i and b_j with $b_i + b_j < m/p$, we can shift weight to the larger $x_.$. After these operations, we have $p^2/m \leq k \leq 2p^2/m$, and we continue by increasing all b_i to m/p. Then we get $w \leq \frac{2p^2}{m} \frac{m}{p} \sum \log x_i = p \sum \log x_i$. It is well known that the sum over the logarithms is maximized for equally distributed $x_i = m/k = m\frac{m}{p^2} = \left(\frac{m}{p}\right)^2$. Hence, we get $w \leq 2p \log \frac{m}{p}$, as desired.

Applying the same reasoning to the y_i, we arrive at the number of accesses to the matrix being $O\left(p \log \frac{m}{p}\right)$.

For $p = \Theta(n)$, this is optimal by the Lemma 5. Generalizing that idea, we can embed any $p \times m/p$ semi-Monotone Matrix Search into an $n \times m$ matrix. Hence, the lower bound of Lemma 4 implies a lower bound of $\Omega(p \log \frac{m}{p})$, matching the upper bound asymptotically.

Numbers The algorithm needs to be able to calculate (multiply, add, compare) with numbers up to $27(nm)^3$ (representation size $O(\log nm)$). If we assume it does so in constant time per algebraic operation, the computation time is asymptotically the same as the number of accesses to the matrix.

6 Acknowledgment

I would like to thank Sven Oliver Krumke and Matthias Altenhoefer for introducing me to the problem and for several fruitful discussions.

References

[1] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.

[2] W. Fulton. *Young tableaux*, volume 35 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 1997. With applications to representation theory and geometry.

[3] D. B. Johnson and T. Mizoguchi. Selecting the kth element in $x + y$ and $x_1 + x_2 + \cdots + x_m$. *SIAM Journal on Computing*, 7(2):147–153, 1978.

[4] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.

[5] A. C. C. Yao. Probabilistic computations: towards a unified measure of complexity. In *Proc. 18th FOCS*, pages 222–227. IEEE, 1977.