Finding the Minimum Cost Acceptable Element in a Sorted Matrix

Sebastián Urrutia
Molde University College, Norway

Vinicius dos Santos
Federal University of Minas Gerais, Brazil

Abstract

In this work we introduce the problem of finding a minimum cost acceptable element in an $n \times n$ matrix M whose columns and rows are sorted in non-decreasing order. More precisely, given a sorted matrix M and access to a given oracle function $f : \mathbb{N} \times \mathbb{N} \to \{\text{TRUE}, \text{FALSE}\}$, one has to find a pair (i, j) of indices such that $f(i, j)$ returns TRUE and the value $M[i, j]$ is as small as possible. Assuming the computation of $f(i, j)$ takes time bounded by a constant, a naive algorithm scanning all the positions of the matrix takes time $O(n^2)$. Another natural approach, based on a priority queue, takes time $O(z \log z)$ in which z stands for the position of the first pair of indices for which the oracle returns TRUE in a sorted list of all elements of M. In the worst case, when $z = n^2$, the naive algorithm is better than the priority queue one. In this work we introduce different algorithms with complexities depending on n and z, such as $O(n\sqrt{z})$ and $O(\min(n^2, z^2))$, and compare them, both theoretically and experimentally, in terms of running time and number of calls to the oracle. Among other things, we find that in most cases our algorithms do not make a significantly larger number of calls to the oracle than the priority queue-based algorithm, which achieves the minimum of such call when all elements of the matrix are distinct, while being much faster in large instances.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Search, Sorted matrix, Oracle function, Algorithm complexity

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.28

Funding *Vinicius dos Santos*: Partially funded by FAPEMIG grant APQ-01707-21 and CNPq grants 312069/2021-9, 406036/2021-7, 404479/2023-5.

1 Introduction

Consider two tasks to be performed simultaneously. Both of them need some resources (time, space, devices, raw material, manpower, etc.) in order to be executed. Tasks may be done in a given number n of modes and each of their n modes has an associated cost. Since resources are shared by the two tasks, some pairs of modes of execution (one mode for each task) may be incompatible. Then, it is natural to think on a problem in which one needs to assign an execution mode to each task such that the assigned modes are compatible and the cost of choosing both modes simultaneously is as small as possible.

We assume that the n execution modes for each task are given sorted in such a way that mode (i, j), corresponding to the i^{th} mode for the first task and the j^{th} mode of the second task, is as least as costly as modes $(i - k, j), 0 \leq k < i$ and $(i, j - k), 0 \leq k < j$. Also, we are given an oracle function $f(i, j)$ that returns TRUE if and only if the i^{th} mode for the first task is compatible with the j^{th} mode of the second task.

As a real application consider a Berth and Crane Allocation Problem (BCAP) [6] in which two vessels need to use port facilities to deliver a number of containers and pick-up some others. Both vessels need space, time, manpower and equipment to complete their tasks. The amount and quality of the resources assigned to the tasks (execution modes) determine a financial cost that should be paid to the port operator and the resources available at the

port must be shared by the two vessels. In this case, the vessels owner would like to find a pair of compatible modes (one for each task) with minimum cost in the sorted matrix M that stores the cost of every pair of execution modes.

2 Formalization

Let M be a non-decreasingly sorted matrix of size $n \times n$ and $f : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ a boolean function (oracle function) that takes a pair of valid indices of the matrix as parameters. The problem consists in determining a pair of indices (i, j) such that $f(i, j)$ is TRUE and $M[i, j]$ is as small as possible. If no such pair exists the problem has no solution. To the best of our knowledge, such problem has not been previously studied in the literature.

Function f is treated here as a black box that indicates if a pair of elements, denoted by their indices, is *acceptable* or not. There is no other way to determine if a pair of indexes is acceptable, we assume no knowledge on the nature of f. For our theoretical results we will assume that function f takes $O(1)$ time to compute. Since in practice the time to compute f can be of relevance, in our experimental analysis we will also consider oracle functions with different computational costs.

Let us define s as an array of pairs of indices of M sorted non-decreasingly by $M[i, j]$. Then, we define z as an unknown parameter indicating the position of the first acceptable pair of indexes in s, that is, $f(s[z]) = True$ and $f(s[\ell]) = False$ for every $1 \leq \ell < z$. If there is no pair (i, j) with $f(i, j) = True$ we set $z = n^2$.

In the case of ties, z may take any value among the positions of the tied pairs. In that case, the value of z is not well defined. This fact may complicate the theoretical analysis of the algorithms performed in Section 4. For that reason, we assume in that section that there are not repeated values in M, for simplicity. Observe that the situation can be handled by using the indices i and j for tie breaking, or considering that in case of ties acceptable elements occur after the non-acceptable ones of the same cost.

One special case of sorted matrices are the so-called $X + Y$ matrices [3, 4, 5]. Let X, Y be two vectors of dimension n sorted in non-decreasing order. The matrix $X + Y$ resulting from the Cartesian sum of X and Y is the matrix A in which $A[i, j] = X_i + Y_j$. In many applications, including versions of BCAP [6], the cost associated with a pair of tasks is just the sum of their individual costs, which can be modelled as $X + Y$ matrices.

3 Related works

Sorted matrices have been considered in the literature and many natural problems have been studied, including sorting, selection and search problems.

The complexity of *sorting* a sorted square matrix into a linear array has not been shown lower than the complexity of sorting a general matrix, that is, the best known algorithms take time $\Omega(n^2 \log(n))$. For the special case of $X + Y$ matrices, [3] gives a non-constructive proof showing that the elements of these matrices can be sorted with just $O(n^2)$ comparisons. In [5] an algorithm with such number of comparisons is presented, nevertheless the time complexity of the algorithm is still $\Theta(n^2 \log(n))$.

The computation of statistics such as the median value or, more generally, obtaining the t^{th} smallest value in a sorted matrix has been studied, and is sometimes called *selection* problem. Algorithms for this problem have been presented in [2, 7] with time complexity $O(n)$. Simpler algorithms with the same complexity are obtained in [4].

The *search* problem consists in, given a value v, find a pair of indices (i, j) such that $M[i, j] = v$. In [1] the authors show that the problem can be solved in $\Theta(n)$ and that the complexity is optimal.

4 Algorithms

In this section we present six algorithms for the problem introduced in the previous sections. We start from scratch with a naive algorithm and then we exploit the structure of the input in order to find more efficient algorithms. It is important to remark that, since function f is given as a black box, a time complexity $\Omega(n^2)$ is unavoidable in some instances. For instance, if there is a unique pair (i, j) for which $f(i, j)$ is TRUE, any algorithm that ignores (i, j) could not be correct. Also, finding an acceptable pair does not ensure its optimality, implying a lower bound of $\Omega(z)$. Hence, in many algorithms we will be interested in strategies that could work well in practice, even if their worst case complexity are still as bad as the naive algorithm.

4.1 A naive algorithm

The simplest way to deal with the problem is to evaluate all elements of M requiring $\Theta(n^2)$ operations.

One can evaluate $f(i, j)$ for all the n^2 pairs of valid indices and return the acceptable pair with the smallest value in M. This algorithm takes $\Theta(n^2)$ time.

The procedure can be easily modified by not testing any pair of indices dominated by an already known acceptable pair. That is, if (i, j) is known to be acceptable, there is no need to test the acceptability of any pair (ℓ, p) such that $M[\ell, p] \geq M[i, j]$. Depending on how the pairs are scanned (fixing the first index first, fixing the second index first, etc) and the distribution of acceptable pairs, the modification may not improve the computational time of the algorithm and may worsen it. For that reason, this kind of modification will not be further analysed.

4.2 A sort-based algorithm

The second natural approach to the problem would be to sort the elements of M in order to evaluate them sorted by cost and stopping the evaluation when the first pair of acceptable indices is found. As mentioned before, even using the best known algorithm for sorted matrices, the sorting step would require $\Theta(n^2 \log n)$ time, which would dominate the time complexity. Note that, after the sorting step, it is possible to stop the algorithm as soon as the first acceptable pair is found.

Instead of sorting the whole input, it is possible to exploit the structure of the input, which is partially sorted, in order to make use of the previous observation. Recall that we denote by s the array containing pairs of indices sorted non-decreasingly by $M[i, j]$ and that z is the position in s for the first pair p for which $f(p)$ is TRUE. By using a priority queue, it is possible to devise an algorithm requiring $O(z \log z)$ operations.

In the sort-based algorithm, the priority queue will ensure that the pairs are evaluated in order of their value in M, to make it possible to stop the execution as soon as a viable pair is found. The queue starts with the pair $(1, 1)$ and every time the pair (i, j) is taken out of the queue for testing its acceptability, the pairs $(i + 1, j)$ and $(i, j + 1)$ are added to the queue if they are valid indices and were not previously inserted in the queue. Implementing the priority queue with a binary heap, the algorithm runs in $O(z \log z)$.

A technical issue must be addressed in order to be able to check whether a pair was already inserted in the queue without using too much additional space. By breaking ties in the priority queue by the sum of the indices of the pairs, one can be sure that, if the pair (i, j) is in the queue, no pair $(i + t, j + u)$ with both t and u positive is in the queue. Therefore, the elements on the queue form the border of an orthogonal convex region. Then, no more than $2n - 1$ pairs are in the priority queue at any point in the algorithm execution. Also, note that due to this fact, in order to know if a pair (i, j) that is about to be added to the queue is already there, we need only to compare j with the maximum value k such that (i, k) is in the queue. In consequence the algorithm can be implemented using no more than $O(n)$ additional space: $O(n)$ for the binary heap and $O(n)$ for storing the maximum second index in the queue for each possible value of the first index.

Observe that the $O(z \log z)$ running time could be much better than the $\Theta(n^2)$ time of the naive approach, since z can be small when compared to n^2 or even compared to n. Indeed, if $z = O(n^{2-\epsilon})$ for any $\epsilon > 0$, the sort-based approach would be asymptotically better. However, if $z = \Theta(n^2)$, the complexity of the algorithm becomes $O(n^2 \log(n))$ which can be worse than the naive approach.

4.3 Exploiting the main diagonal of the matrix

The next two algorithms to be presented here exploit the main diagonal of M and require $O(n\sqrt{z})$ operations. Asymptotically, these algorithms dominate the trivial algorithm as their complexity is $O(n^2)$ even in the worst case when $z = \Theta(n^2)$ and are better when the value of z is small.

The algorithms are based on evaluating with function f up to iteration ℓ, at least all the pairs of indices (i, j) such that $M[i, j] \leq M[\ell, \ell]$.

The following simple theorem form the base of the two next algorithms.

▶ **Theorem 1.** *Let $i, j, k \in \{1, \ldots, n\}$ such that $i \times j \geq k$. Then $M[i, j] \geq M[s[k]]$*

Proof. Since M is sorted, $M[i, j] \geq M[i', j']$ for $1 \leq i' \leq i$ and $1 \leq j' \leq j$. There are $i \times j \geq k$ possible distinct values for i' and j'. Then, the position of the pair (i, j) in s is at least k. ◀

An immediate consequence of the theorem is that, when our following algorithms finish iteration $\lceil \sqrt{z} \rceil$ one can be sure that all the first z elements of s has been evaluated with function f.

4.4 L-shape algorithm

As the name may suggest, L-shape algorithm will proceed by iterating through lines and columns of M in increasing order. From now on, in all of our algorithms we will assume that algorithms will be implemented with lazy-evaluation, meaning that if there are two or more conditions in a loop of control sequence, they will be evaluated in order and only if all previous conditions hold.

As shown in Algorithm 1 this approach evaluates in iteration ℓ all the not previously evaluated elements of M in the line $\ell - 1$ and in column $\ell - 1$ that are better than the best known acceptable pair, or all elements, if no acceptable element is known so far. By the end of one iteration of the main loop, for an integer ℓ all elements in lines or columns smaller than ℓ will have been visited (or ignored if a better acceptable solution was already found).

Note that the algorithm does not necessarily stop after the iteration in which it first finds an acceptable pair. It stops when the element in the main diagonal is larger than the best (smallest) acceptable pair found. At this point, we know that no better element would be evaluated by continuing the algorithm execution, and the correctness follows.

Algorithm 1 L-shape algorithm. It evaluates a line and a column of M at each iteration.

```
Data: M, f
if f(1,1) = True then
    return (1,1)
i ← 2, best ← +∞
while i ≤ n + 1 and best > M[i − 1, i − 1] do
    j ← i
    while j ≤ n and M[i − 1, j] < best do
        if f(i − 1, j) = True then
            best ← M[i − 1, j]
            bestpair ← (i − 1, j)
        j ← j + 1
    j ← i
    while j ≤ n and M[j, i − 1] < best do
        if f(j, i − 1) = True then
            best ← M[j, i − 1]
            bestpair ← (j, i − 1)
        j ← j + 1
    if M[i, i] < best and f(i, i) = True then
        best ← M[i, i]
        bestpair ← (i, i)
    i ← i + 1
if best < +∞ then
    return bestpair
return "no solution"
```

Let *best* be smallest cost of an acceptable element and let ℓ be the largest integer such that $M[\ell, \ell] < best$. From Theorem 1, $z \geq \ell^2$. Since *bestpair* have at least one coordinate bounded by $\ell + 1$, the algorithm main loop is executed at most $\ell + 1 = O(\sqrt{z})$ times, with each iteration performing $O(n)$ steps, resulting in a $O(n\sqrt{z})$ running time. Also note that this is an in-place algorithm, that is, it requires just constant extra space.

4.5 Contour-line algorithm

As in the previous algorithm the Contour-line algorithm exploits elements in the main diagonal to bound its running time. However, instead of visiting a whole line (column) at once, in iteration ℓ, only elements that are smaller or equal to $M[\ell, \ell]$ are evaluated. This way, at the end of iteration ℓ, the elements already evaluated form a contour line in the matrix bounded by the value of $M[\ell, \ell]$. Let t_ℓ be the number of elements of M with value at most $M[\ell, \ell]$. Then, by the end of iteration ℓ the algorithm has evaluated exactly the first t_ℓ elements of s. Once an acceptable solution is found, which happens for the smallest ℓ for which $t_\ell \geq z$, the algorithm stops. In that regard it is similar to the sort-based algorithm, evaluating elements bounded by certain value at each step, while it avoids the extra computations related to the upkeep of the heap.

Notice that, contrary to the previous algorithm, the Contour-line stops at the end of the iteration in which an acceptable pair of indices is found as no better pair would be evaluated in future iterations. Also, whenever the best acceptable pair is in the main diagonal of M, the Contour-line algorithm performs as few evaluations of the oracle function as the sort-based algorithm (assuming no ties), that is, as few as possible.

Algorithm 2 Contour-line Algorithm. Evaluates just pairs with value smaller than or equal to pair (ℓ, ℓ) up to iteration ℓ.

```
Data: M, f
for j ← 1 to n do
    contourline[j] ← 1
i ← 1, best ← +∞
while i ≤ n and best = +∞ do
    for j ← 1 to n do
        while contourline[j] ≤ n and M[j, contourline[j]] ≤ M[i, i] do
            if M[j, contourline[j]] < best and f(j, contourline[j]) = true then
                best ← M[j, contourline[j]]
                bestpair ← (j, contourline[j])
            contourline[j] ← contourline[j] + 1
    i ← i + 1
if best < +∞ then
    return bestpair
else
    return "no solution"
```

Let ℓ be the number of iterations of the main loop of the algorithm. As in the analysis of Algorithm 1, no more than $\ell = O(\sqrt{z})$ iterations are needed. The cost of iteration i, is proportional to n plus the number of times that an element of *contourline* is incremented. But notice that the value of $contourline[j]$ cannot exceed i when $j > i$, implying that by the end of the algorithm elements of *contourline* were incremented at most $\ell n + (n - \ell)\ell$ times, resulting in a $O(n\sqrt{z})$ complexity.

In contrast to L-Shape algorithm, Contour-line requires $\Theta(n)$ additional space to store the contour line.

4.6 Expanding Square Algorithm

The next algorithm consists in two stages.

In the first stage, only elements in the square defined by the pairs $(1, 1)$ and (k, k) are evaluated up to iteration k. This first stage ends after finishing an iteration in which an acceptable pair is found. Let (i, j) be the best pair for which $f(i, j)$ tested *true* during this first stage. If there is an acceptable pair better than (i, j), it must be of the form $(i - t, j + u)$ or $(i + t, j - u)$ with both t and u positive.

In the second stage, just elements outside of the sub-matrix already evaluated that are smaller than the best pair found are evaluated.

Let $\ell = \max\{i, j\}$ and suppose first that the pair in position z of vector s is not in the $\ell \times \ell$ sub-matrix the algorithm visited. Let $(i_{opt}, j_{opt}) = s[z]$ and $\ell_{opt} = \max\{i_{opt}, j_{opt}\}$.

The number of positions of M visited during the execution of Algorithm 3 is bounded by ℓ^2 in the first stage, plus a maximum of $2(\ell(\ell_{opt} - \ell))$ until finding (i_{opt}, j_{opt}), plus at most z elements that are smaller than $M[i_{opt}, j_{opt}]$ and are evaluated after (i_{opt}, j_{opt}) is found.

We have that $\ell^2 + 2(\ell(\ell_{opt} - \ell)) + z = \ell^2 + 2\ell\ell_{opt} - 2\ell^2 + z - = 2\ell\ell_{opt} - \ell^2 + z < 2\ell\ell_{opt} + z$. Since $s[z]$ is not in the $\ell \times \ell$ submatrix, $\ell_{opt} > \ell$. Clearly, we also have $z \geq \ell_{opt}$. Then, $2\ell\ell_{opt} + z \leq \min\{3n^2, 3z^2\}$.

Algorithm 3 Expanding square algorithm.

```
Data: M, f
if f(1,1) = true then
    return (1,1)
i ← 2, best ← +∞
while i ≤ n and best = +∞ do
    for j ← 1 to i do
        if M[i,j] < best and f(i,j) = true then
            best ← M[i,j]
            bestpair ← (i,j)
        if M[j,i] < best and f(j,i) = true then
            best ← M[j,i]
            bestpair ← (j,i)
    i ← i + 1
if best = +∞ then
    return "no solution"
if i ≤ n then
    ℓ ← i − 1
    while i ≤ n and (M[i,1] < best or M[1,i] < best) do
        j ← 1
        while j < ℓ and (M[i,j] < best or M[j,i] < best) do
            if M[i,j] < best and f(i,j) = true then
                best ← M[i,j]
                bestpair ← (i,j)
            if M[j,i] < best and f(j,i) = true then
                best ← M[j,i]
                bestpair ← (j,i)
            j ← j + 1
        i ← i + 1
return bestpair
```

When the pair (i, j) found in the first stage is the best acceptable pair, the value of $\ell = \max\{i, j\}$ equals the size of the square. Since $z \geq \ell$, the number of elements visited by the algorithm during the first stage is $\ell^2 \leq z^2$. In the second stage only elements smaller than $M[i, j]$ are visited and those are limited by z. Then the total number of elements evaluated are limited by $2z^2$.

The algorithm never evaluates a position more than once and then the complexity of this in-place algorithm is $O(\min\{n^2, z^2\})$. Also, the algorithm requires just constant additional space.

4.7 Level curves algorithm

One implication of Theorem 1 is that if the cheapest acceptable pair is in position (i_{opt}, j_{opt}), then $z \geq i_{opt} \times j_{opt}$. The algorithm presented in this subsection explores this idea by splitting elements in sets $S_1, \ldots, S_n$ such that $S_k = \{(i, j) \mid (k-1)n < ij \leq kn\}$ and evaluating these sets in order. Once an acceptable pair is found, its cost is used to prune the exploration, similarly as in the contour-line algorithm.

More precisely, in the k-th iteration the algorithm evaluates elements of S_k, in increasing order of coordinates. As in previous algorithms, there is a variable *best* containing the cost of the best acceptable pair found so far. Whenever evaluating a pair (i, j), if $M[i, j] > best$, no further elements in the same line are considered in the current iteration or in further ones. We remark that $\Theta(n)$ information is stored to make this pruning possible, by storing a list of indices i (lines of the matrix M) which can still be searched.

Let (i_{opt}, j_{opt}) be the optimal pair and let $k_{opt} = \lceil i_{opt}.j_{opt}/n \rceil$. By construction, the pair (i_{opt}, j_{opt}) will be found in iteration k_{opt}. Up to that point, exactly $\lfloor k_{opt}.n/i \rfloor$ elements with first coordinate i are visited. So, the number of visited pairs is bounded by

$$\sum_{i=1}^{n} \min(\frac{k_{opt}.n}{i}, n) \leq \sum_{i=1}^{n} \frac{k_{opt}.n}{i} = k_{opt}.n \sum_{i=1}^{n} \frac{1}{i} = O(k_{opt}.n \log n).$$

Since $(k_{opt} - 1)n < i_{opt}.j_{opt}$ and $i_{opt}.j_{opt} \leq z$, then the running time up to iteration k_{opt} can be bounded by $O(z \log n)$ when $k_{opt} > 1$ and by $O(n \log n)$ in the first iteration. Putting everything together, we get an upper bound of $O((n + z) \log n)$. In case k_{opt} is very large, this bound is not tight, since the min operation in the inequality becomes more relevant. It is not hard to see that the algorithm never performs more than $O(n^2)$ operations.

Notice, however, that the algorithm has no way to identify that the desired acceptable element has been found. It remains to bound the running time of the rest of the execution.

During further iterations, we still consider sets S_k in order. Whenever visiting a pair (i, j), either it has smaller cost than (i_{opt}, j_{opt}), or we update a auxiliary data structure to never consider elements with the same first coordinate. Hence, the total number of steps after finding the solution is bounded by $O(n + z)$.

Putting all together, we have a $O(\min((n + z) \log n), n^2)$-time $\Theta(n)$-space algorithm.

4.8 Comparison and summary of the algorithms

With the exception of the sort-based algorithm, none of the algorithms we discussed have asymptotically worse running time than the Naive algorithm, as illustrated in Table 1. As for space, three algorithms can be implemented using only additional constant space. Contour-line, keeps a vector of n positions to track the contour line, the sort-based, that requires a heap or another non-trivial data structure and the level curves algorithm stores a linear array indicating where to stop the evaluation in each line of M.

Table 1 Comparison of the algorithms in terms of time and space complexity.

Algorithm	Complexity	Space
Naive	$O(n^2)$	$O(1)$
S-based	$O(z \log z)$	$O(n)$
L-shape	$O(n\sqrt{z})$	$O(1)$
Contour-line	$O(n\sqrt{z})$	$O(n)$
Expanding square	$O(\min(n^2, z^2))$	$O(1)$
Level curves	$O(\min((n + z) \log n), n^2)$	$O(n)$

Many of the times complexities are incomparable. Besides, it is not reasonable to assume the value of z is known beforehand. If some estimation of the value of z is available in a specific setting, one algorithm can be preferable over another. But besides the worst-case analysis, the algorithms present differences and subtleties, some of which are not captured by Table 1, and the number of steps performed for a given instance could be much smaller than the worst case analysis suggests. We discuss a few of them in the rest of this section and other aspects will be brought to light when analysing the experiments, in Section 5.

L-shape and contour-line have the same worst-case time complexity, but different space complexity. But it is not hard to conceive instances for which contour-line would run faster than L-shape, implying that the extra space bring advantages in some cases. On the other hand L-shape could make best use of cache, by visiting long sequences of consecutive elements, in particular in the case of $X + Y$ matrices, stored implicitly.

Expanding square algorithm shares many similarities with contour-line algorithm, in the sense that in both cases there is a value ℓ such that all elements "inside the ℓ-square", i.e. elements (i, j) with $i, j \leq \ell$, are evaluated and all other elements considered satisfy $\min(i, j) \leq \ell$. Among elements outside the square, while contour-line only considers elements smaller than $M[\ell, \ell]$, Expanding square consider elements smaller than the element found, which is potentially smaller. However, this implicit square could be different in each algorithm: Contour-line is able to find an acceptable element outside the square, aborting the execution of the algorithm with a much smaller value of ℓ. Finally, as the complexity analysis suggests, Expanding square may be fast when the cheapest acceptable element is close to the main diagonal, since most of the calls to oracle function f were made to pairs of smaller cost the the optimal, and hence unavoidable.

As the previous discussion suggests, the position of the optimal element, in addition to the value of k, may impact the running time of the algorithms. Another instance of this can be observed when comparing algorithms L-shape and level curves. It is not hard to convince oneself that, after k iterations of the main loop of both algorithms, the set of positions considered by L-shape is a subset of the ones considered by level curves. Besides, L-Shape is a simpler algorithm and it is likely that in any implementation it would have smaller constants. Consider the optimal element is in position $(c, n - d)$ for small constants c and d. Then both algorithms will visit this element around the same iteration and, from the previous discussion, L-Shape would find it faster. On the other hand, if the element is close to the main diagonal, level curves algorithm has a chance to be much faster than L-Shape. Suppose, for example, that the desired element is in position $(\sqrt{n}, \sqrt{n})$. Level curves would find it in the first iteration and spend at most $O(n + z)$ additional steps before finishing its execution, to a total of $O(n \log n + z)$ steps, while L-Shape run $\sqrt{n}$ iterations, possibly spending $O(n\sqrt{n})$ steps, which could be much bigger if $z = o(n\sqrt{n})$.

5 Experiments

In this section we discuss our computational tests of the algorithms developed in the previous sections. We first describe how we constructed our benchmark instances and the computational environment in which the experiments were run. Then, we show the results from the experiments.

5.1 Instances

For constructing instances we define a set of parameters detailed in Table 2. By combining different values of the parameters we defined a set of 60 problem instances for which the algorithms where tested.

The instances created are denoted by the value of their parameters. The first number corresponds to the size of the matrix, the second and third refer to the pair of indexes reporting the position of the optimal acceptable element, the forth indicates which is the probability of elements with cost larger than the optimal to be acceptable, the fifth defines the matrix construction method and the last the oracle speed. For example, an instance named I.500.300.310.1.0.1 is an instance with a 500×500 matrix in which the optimal acceptable pair is $(300, 310)$, elements with larger cost have probability of $1/500$ of being acceptable, the matrix is constructed with the 0 method and a slow oracle is used.

Table 2 Parameters for instance construction.

Parameter	Domain	Description
n	Positive integer	Matrix M is of size $n \times n$.
(i^*, j^*)	Pair of valid indices of M	Position of the optimal acceptable element.
e	$\{0, 1, 2\}$	Probability p of a pair (i, j) such that $M(i, j) > M(i^*, j^*)$ being acceptable: 0: $p = 0$, 1: $p = 1/n$, 2: $p = 1$.
v	$\{0, 1, 2, 3\}$	Construction of M: 0: $X + Y$, X and Y with elements increasing by a random amount between 0 and 10 (inclusive) from one element to the next. 1: $X + Y$, X and Y with each element being twice the previous one. 2: $X + Y$, X and Y where the difference between consecutive elements at positions $i+1$ and i is half the difference between elements at positions i and $i-1$. 3: all elements have the same value.
t	$\{0, 1\}$	Complexity of oracle: 0: Fast oracle, returns the answer in $O(1)$ time. 1: Slow oracle, perform $O(n)$ dummy operations before returning the answer.

We notice that all matrices we use in our experiments are $X + Y$ matrices. This allow us to compute the cost of any position on the matrix as the sum of the two respective position in the vectors. In that way we avoid dealing with very big matrices in memory, which would be a bottleneck in some experiments. Another reason for the choice is that in real problems (in particular in the one that inspired this work) the sorted matrix frequently is a $X + Y$ matrix.

5.2 Computation Enviroment

All algorithms were implemented in C and compiled with gcc distributed by Apple (Apple clang version 14.0.0, clang-1400.0.29.102) with no compilation flags. The experiments were run on a MacBook Air running a Mac OS 12.7.2, with a 2,2 GHz Intel Core i7 Dual-Core processor and 8GB 1600MHz DDR3 memory.

5.3 Results

In Tables 3, 4 and 5 we show the obtained computational results for the algorithms developed in this work. Tables 3 and 4 report on the number of evaluations of the oracle for all the six algorithms using several instances with $n = 50$ in Table 3 and $n = 500$ in Table 4. Finally, Table 5 shows computational times obtained for all the algorithms except for the naive one. We show results with $n = 5000$ and a slow oracle and with $n = 20000$ and a fast oracle. Computational time is displayed in seconds. In all the tables, best results are written in bold.

Several interesting facts regarding the number of evaluations of the oracle can be noted from Table 3 and confirmed from Table 4:

Table 3 Number of calls of the oracle function for small instances.

Instance	Naive	S-based	L-shape	C.-line	E. Square	Level curves
I.50.25.25.0.0.0	2500	**1288**	1825	1296	1309	1461
I.50.5.5.0.0.0	2500	**45**	385	**45**	48	115
I.50.5.50.0.0.0	2500	1323	1330	**1314**	2459	1330
I.50.25.25.1.0.0	2500	**1288**	1341	1296	1309	1340
I.50.5.5.1.0.0	2500	**45**	188	**45**	48	112
I.50.5.50.1.0.0	2500	1323	1330	**1314**	1416	1330
I.50.25.25.2.0.0	2500	**1288**	1303	1296	1309	1302
I.50.5.5.2.0.0	2500	**45**	48	**45**	48	48
I.50.5.50.2.0.0	2500	1323	1319	**1314**	1349	1319
I.50.25.25.0.1.0	2500	**625**	1825	**625**	648	1459
I.50.5.5.0.1.0	2500	**25**	385	**25**	28	109
I.50.5.50.0.1.0	2500	2411	2411	**2410**	2459	2411
I.50.25.25.1.1.0	2500	**625**	792	**625**	648	789
I.50.5.5.1.1.0	2500	**25**	186	**25**	28	106
I.50.5.50.1.1.0	2500	2411	2411	**2410**	2459	2411
I.50.25.25.2.1.0	2500	**625**	626	**625**	648	627
I.50.5.5.2.1.0	2500	**25**	26	**25**	28	27
I.50.5.50.2.1.0	2500	2411	2411	**2410**	2459	2411
I.50.25.25.0.2.0	2500	**1825**	**1825**	**1825**	1848	1826
I.50.5.5.0.2.0	2500	**385**	**385**	**385**	388	386
I.50.5.50.0.2.0	2500	**474**	475	**474**	2459	686
I.50.25.25.1.2.0	2500	**1825**	**1825**	**1825**	1848	1826
I.50.5.5.1.2.0	2500	**385**	**385**	**385**	388	386
I.50.5.50.1.2.0	2500	**474**	475	**474**	480	476
I.50.25.25.2.2.0	2500	**1825**	**1825**	**1825**	1848	1826
I.50.5.5.2.2.0	2500	**385**	**385**	**385**	388	386
I.50.5.50.2.2.0	2500	**474**	475	**474**	480	476
I.50.25.25.0.3.0	2500	1202	1825	1225	**648**	1459
I.50.5.5.0.3.0	2500	41	385	205	**28**	109
I.50.5.50.0.3.0	2500	1436	431	**250**	2459	677
I.50.25.25.1.3.0	2500	1202	1825	1225	**648**	1459
I.50.5.5.1.3.0	2500	41	385	205	**28**	109
I.50.5.50.1.3.0	2500	1436	431	**250**	2459	677
I.50.25.25.2.3.0	2500	1202	1825	1225	**648**	1459
I.50.5.5.2.3.0	2500	41	385	205	**28**	109
I.50.5.50.2.3.0	2500	1436	431	**250**	2459	677

1. The Naive algorithm as expected does n^2 evaluations of the oracle for all instances.
2. In many of the instances the algorithms L-shape, contour-line and expanding square perform a number of evaluations similar to the sort-based algorithm which makes exactly z evaluations. Level curves algorithm is also competitive, but when the pair (i^*, j^*) have small coordinates, the algorithm makes a number of unnecessary evaluations in its first (unique) iteration.
3. In some cases, the last four algorithms performed fewer calls of the oracle than the sort-based algorithm. This is only possible due to the presence of elements of the matrix with the same value. This happens more often when method 3 (all elements on the matrix have the same value) is used for the construction of the matrix.

Table 4 Number of calls of the oracle function for medium-sized instances.

Instance	Naive	S-based	L-shape	C.-line	E. Square	Level curves
I.500.250.250.0.0.0	250000	120771	187000	**120757**	120959	148483
I.500.5.5.0.0.0	250000	**42**	3985	**42**	44	1060
I.500.5.500.0.0.0	250000	136165	136152	**136151**	249535	136152
I.500.55.55.0.0.0	250000	**6296**	51085	6305	6348	16877
I.500.250.250.1.0.0	250000	120771	122390	**120757**	120959	122276
I.500.5.5.1.0.0	250000	**42**	1709	**42**	44	852
I.500.5.500.1.0.0	250000	136165	136152	**136151**	138425	136152
I.500.55.55.1.0.0	250000	**6296**	7363	6305	6348	6976

4. The L-shape algorithm gets bad results when the optimal acceptable pair of indices lay in the main diagonal of the matrix and specially if that pair is the only acceptable pair. In that case, until that pair is visited the algorithm evaluates entire lines and columns of the matrix.
5. The expanding square algorithm gets bad results when the optimal acceptable pair of indices lay far away from the main diagonal of the matrix and specially if that pair is the only acceptable pair. In that case, the square in which each element is evaluated has to be expanded to cover almost the whole matrix.
6. Level curves algorithm seems to deal better with the extreme cases of previous items: when simultaneously compared to L-shape and expanding square, it is never worse than both. In particular, in instances for which L-shape performs well among all algorithms, Level curves has a slightly worse performance. However, for instances where L-Shape performed badly, level curves had a much better result.
7. Parameter e controlling if other elements beside the optimal are acceptable only affects algorithms L-shape, expanding square and level curves. By design, the sort-based algorithm never evaluates elements with cost higher than the optimal. Contour-line may evaluate elements higher than the optimal in rare occasions in the last iteration, but it did not happen in our experiments.
8. With matrix construction methods 1 and 2 sort-based algorithm and contour-line perform almost the exact same number of evaluations. The same is true for L-shape only for construction method 2.

Table 5 shows that when checking the oracle is computationally expensive, the sort-based algorithm and the contour-line algorithm are to be preferred. Sort-based algorithm guarantees the lowest number of oracle calls if there are no elements with the same cost as the optimal acceptable element. Contour-line, as shown in the previous tables, performs in practice around the same number of calls to the oracle. The algorithms are basically tied in all instances and are better than the rest.

For the large instances with a fast oracle the results are very different. In fact, the best algorithms are L-shape and expanding square while level curves also performs very well. Contour-line perform reasonably well when the optimal element is close to the main diagonal, but is always slower than expanding square. Observe that as noticed in the previous tables, L-shape is better when the optimal acceptable element is close to a border of the matrix while expanding square is better when the acceptable element is the main diagonal. Level curves performs reasonably well in most instances, outperforming L-shape when the optimal element is close to the main diagonal and expanding square when one of the coordinates is large and the other small. The best case for level curves is illustrated by instance i.20000.1000.20.0.0.0, when the algorithm finds the optimal element in the first iteration, while L-shape requires many, and the element is far from the diagonal, making expanding square waste time.

Table 5 Computational time for large instances.

Instance	S-based	L-shape	C.-line	E. Square	Level curves
I.5000.5.5.0.0.1	**0.002**	1.546	**0.002**	**0.002**	0.414
I.5000.41.500.0.0.1	6.316	19.422	**6.157**	9.523	9.622
I.5000.540.41.0.0.1	6.069	19.358	**5.968**	10.792	9.426
I.5000.541.541.0.0.1	**22.220**	189.546	22.627	22.456	60.554
I.5000.5.5.1.0.1	**0.002**	0.703	**0.002**	**0.002**	0.341
I.5000.41.500.1.0.1	6.141	7.129	**6.109**	9.600	7.090
I.5000.540.41.1.0.1	6.146	6.991	**5.986**	11.006	6.710
I.5000.541.541.1.0.1	22.698	34.331	**21.984**	22.003	30.392
I.20000.10000.10000.0.0.0	64.058	2.964	3.948	**1.863**	3.240
I.20000.10000.50.0.0.0	14.758	**0.508**	0.995	0.867	0.639
I.20000.5.10000.0.0.0	15.236	**0.503**	1.006	0.864	0.661
I.20000.250.250.0.0.0	0.026	0.097	0.003	**0.001**	0.008
I.20000.2400.10000.0.0.0	23.612	1.182	1.614	**0.951**	1.378
I.20000.2500.2500.0.0.0	3.411	1.009	0.329	**0.136**	0.482
I.20000.5000.5000.0.0.0	15.289	1.853	0.983	**0.478**	1.226
I.20000.1000.20.0.0.0	0.113	0.012	0.010	0.009	**0.007**

6 Conclusions

In this work we introduced a simple yet important problem in sorted matrices. The problem asks to find an optimal element of the matrix that satisfies a test performed by a black box oracle. The problem has applications in the assignment of resources to two tasks that must be performed simultaneously.

We showed that an algorithm using the sorted property of the matrix to call the oracle as few times as possible does not dominate in terms of computational complexity the naive algorithm consisting in calling the oracle for each element of the matrix. This observation motivated the development of other algorithms (L-shape, contour-line, expanding square and level curves) that do dominate the naive algorithm having the same complexity in the worst case but being strictly better in other cases.

Each algorithm was described and its complexity was analysed. Different properties including space requirements were discussed.

The algorithms were implemented and computational results were used to compare the number of evaluations of the oracle and the running time of each algorithm. By these experiments we were able to recommend different algorithms depending on the computational cost of the oracle, or on the position of the desired element, when prior knowledge is available.

An open question concerns the case of matrices $X + Y$ in which case it might be possible to use the special structure of those matrices to develop algorithms with lower computational complexity. Also, for some of the proposed algorithms it may be not obvious how to generalize them when the input matrix is not square. We plan to address this issue in a future work.

References

1 Michel Cosnard, Jean Duprat, and Afonso G Ferreira. The complexity of searching in $X + Y$ and other multisets. *Information Processing Letters*, 34(2):103–109, 1990.

2 Greg Frederickson and Donald Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *J. of Comput. and Syst. Sci.*, 24(2):197–208, 1982.

3 Michael L Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361, 1976.

4 Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from heaps, row-sorted matrices, and X + Y using soft heaps. In *Symposium on Simplicity in Algorithms, 2*, pages 5:1–5:21, 2019.

5 Jean-Luc Lambert. Sorting the sums $(x_i + y_j)$ in $O(n^2)$ comparisons. *Theoretical Computer Science*, 103(1):137–141, 1992.

6 Frank Meisel and Christian Bierwirth. Heuristics for the integration of crane productivity in the berth allocation problem. *Transportation Research Part E: Logistics and Transportation Review*, 45(1):196–209, 2009. doi:10.1016/j.tre.2008.03.001.

7 Andranik Mirzaian and Eshrat Arjomandi. Selection in $X + Y$ and matrices with sorted rows and columns. *Information Processing Letters*, 20(1):13–17, 1985.