Finding the saddlepoint faster than sorting

Justin Dallant ✉ 📧
Department of Computer Science, Université libre de Bruxelles, Belgium

Frederik Haagensen ✉ 📧
Department of Computer Science, IT University of Copenhagen, Denmark

Riko Jacob ✉ 📧
Department of Computer Science, IT University of Copenhagen, Denmark

László Kozma ✉ 📧
Institut für Informatik, Freie Universität Berlin, Germany

Sebastian Wild ✉ 📧
Department of Computer Science, University of Liverpool, UK

—— **Abstract** ——

A *saddlepoint* of an $n \times n$ matrix A is an entry of A that is a maximum in its row and a minimum in its column. Knuth (1968) gave several different algorithms for finding a saddlepoint. The worst-case running time of these algorithms is $\Theta(n^2)$, and Llewellyn, Tovey, and Trick (1988) showed that this cannot be improved, as in the worst case all entries of A may need to be queried.

A *strict saddlepoint* of A is an entry that is the strict maximum in its row and the strict minimum in its column. The strict saddlepoint (if it exists) is unique, and Bienstock, Chung, Fredman, Schäffer, Shor, and Suri (1991) showed that it can be found in time $O(n \lg n)$, where a dominant runtime contribution is sorting the diagonal of the matrix. This upper bound has not been improved since 1991. In this paper we show that the strict saddlepoint can be found in $O(n \lg^* n) \subset o(n \lg n)$ time, where $\lg^*$ denotes the very slowly growing *iterated logarithm* function, coming close to the lower bound of $\Omega(n)$. In fact, we can also compute, within the same runtime, the *value* of a non-strict saddlepoint, assuming one exists. Our algorithm is based on a simple recursive approach, a feasibility test inspired by searching in sorted matrices, and a relaxed notion of saddlepoint.

1 Introduction

Saddlepoints are a central concept of mathematical analysis and numerical optimization. Informally, a saddlepoint of a function (or a surface) is a point where the derivatives (slopes) in orthogonal directions vanish, yet the point is not a local minimum or maximum. In this paper we are concerned with a discrete analogue: an entry of a matrix A is a saddlepoint, if it is simultaneously the maximum in its row and the minimum in its column.

If A represents the payoff matrix of a two-player zero-sum game, then saddlepoints of A give the *value* of the game, corresponding exactly to the pure-strategy Nash equilibria (see e.g. [9, § 4]). Thus, finding the saddlepoint of a matrix (as fast as possible) is a natural and fundamental algorithmic question.

$$A = \begin{pmatrix} -0.08 & 0.55 & 0.98 & 1.21 & 1.24 & 1.07 & 0.7 & 0.13 & -0.64 \\ -0.69 & -0.06 & 0.37 & 0.6 & 0.63 & 0.46 & 0.09 & -0.48 & -1.25 \\ -1.1 & -0.47 & -0.04 & 0.19 & 0.22 & 0.05 & -0.32 & -0.89 & -1.66 \\ -1.31 & -0.68 & -0.25 & -0.02 & 0.01 & -0.16 & -0.53 & -1.1 & -1.87 \\ -1.32 & -0.69 & -0.26 & -0.03 & 0 & -0.17 & -0.54 & -1.11 & -1.88 \\ -1.13 & -0.5 & -0.07 & 0.16 & 0.19 & 0.02 & -0.35 & -0.92 & -1.69 \\ -0.74 & -0.11 & 0.32 & 0.55 & 0.58 & 0.41 & 0.04 & -0.53 & -1.3 \\ -0.15 & 0.48 & 0.91 & 1.14 & 1.17 & 1.0 & 0.63 & 0.06 & -0.71 \\ 0.64 & 1.27 & 1.7 & 1.93 & 1.96 & 1.79 & 1.42 & 0.85 & 0.08 \end{pmatrix}$$
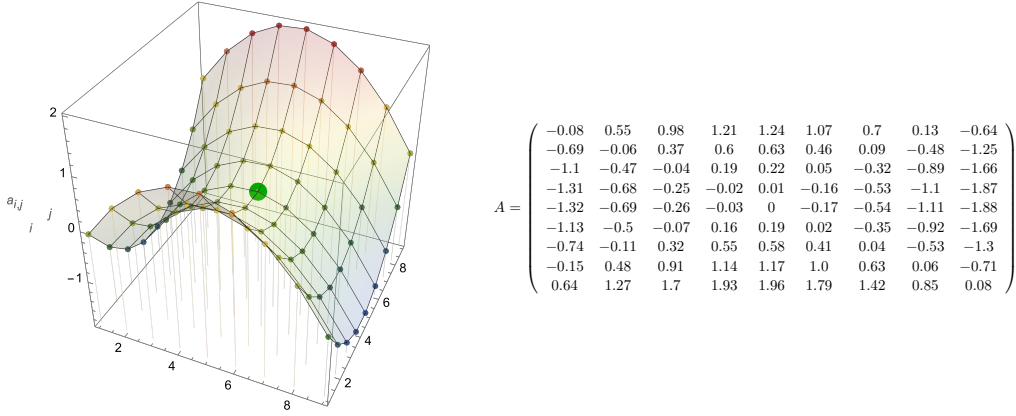
Figure 1 (*right*) A 9×9-matrix A with a (strict) saddlepoint at $a_{5,5} = 0$. (*left*) A 3D plot of the matrix entries with the saddlepoint highlighted in green.

Knuth considered the saddlepoint problem [7, §1.3.2] already in 1968, observing that saddlepoints of A (if they exist) must equal both the minimum of all row-maxima and the maximum of all column-minima, and thus, all saddlepoints of a matrix have the same value. Knuth also gave a number of algorithms [7, pg. 512–515] for finding saddlepoints (or reporting their absence). The runtimes of these algorithms may differ significantly on concrete instances, yet in the worst case they all perform $\Theta(n^2)$ operations on an $n \times n$ square input matrix A.[1] In fact, such a runtime is necessary, as in the worst case all entries of A must be inspected; this can be seen through a simple adversary argument, as shown by Llewellyn, Tovey, and Trick [8].

The situation changes considerably if we require the saddlepoint to be the *strict maximum* in its row and the *strict minimum* in its column: we refer to such an entry as a *strict saddlepoint*. Note that we cannot transform the first problem into the second through a simple perturbation: adding noise to a matrix with non-strict saddlepoints may well create a matrix with no saddlepoints at all! It is not hard to see (e.g. by the above observation of Knuth), that at most one entry of a matrix can be a strict saddlepoint.

The first nontrivial algorithm for finding a strict saddlepoint was given by Llewellyn, Tovey, and Trick [8] in 1988, with a runtime of $O\!\left(n^{\lg 3}\right) \subset O\!\left(n^{1.59}\right)$, which they conjectured to be essentially optimal[2], see also [4]. This conjecture turned out to be false, and an algorithm with runtime $O(n \lg n)$ was given by Bienstock, Chung, Fredman, Schäffer, Shor, and Suri [1] in 1991. Independently and around the same time, Byrne and Vaserstein [3] obtained a similar result.

The bound of $O(n \lg n)$ on the complexity of the problem is a natural barrier and the algorithms achieving it are surprisingly simple.[3] They are deterministic, operate on the matrix A only via constant-time comparisons of entries, and need to inspect only $O(n)$ (adaptively queried) entries. In its original presentation, the algorithm of Bienstock et al. involves $\Theta(n)$ operations on a *heap* that holds $\Theta(n)$ keys – alternatively it can be seen as

---

[1] A running time of the form $O(n \cdot f(n))$ for $n \times n$ input matrices also implies a running time of $O(m \cdot f(n))$ for $m \times n$ or $n \times m$ input matrices with $m \geq n$, as shown in §2.

[2] Throughout the paper, $\lg x$ denotes the base-2 logarithm of x.

[3] We review the algorithm of Bienstock et al. in a slightly modified form in §3.

executing an initial sorting step on $\Theta(n)$ entries that dominates the runtime.

Algorithms based on such a step clearly cannot avoid making at least $\Theta(n \lg n)$ comparisons. But is sorting necessary for finding a saddlepoint? Informally, sorting and saddlepoint-finding do seem related, although in a non-obvious way. If the input matrix A does *not* have a strict saddlepoint, then any correct algorithm must certify this (at least implicitly) for each entry q of A. A sufficient certificate for q is a pair (x, y) with $x \geq y$, where x is an entry in the same row as q, and y is an entry in the same column as q (one of x and y may be q itself); see Figure 2. A moment of thought reveals that such a certificate is also necessary: without it, q may still be the strict saddlepoint.
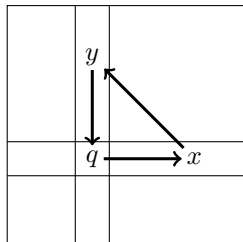
■ **Figure 2** Certificate against q being a strict saddlepoint. Arrows point from larger to smaller entries, inequalities involving q are strict. The conditions of a saddlepoint and $x \geq y$ imply a cycle.

The process of collecting such certificates for all n^2 row-column pairs (through $O(n \lg n)$ comparisons) now appears very similar to sorting n items, i.e. collecting "ordering certificates" for all $\binom{n}{2}$ pairs of items. Given these observations, and the fact that the bound of $O(n \lg n)$ has not been improved in over three decades, it is natural to conjecture that a "sorting barrier" holds, rendering the $O(n \lg n)$ runtime optimal.

Surprisingly, this is not the case. In this paper we show that the strict saddlepoint of an $n \times n$ matrix (or a certificate of its absence) can be found in *almost linear*, $O(n \lg^* n)$ worst-case time, where $\lg^*$ denotes the very slowly growing *iterated logarithm* function. Thus, the runtime is in $o(n \lg \lg \ldots \lg n)$ with the lg function iterated any fixed number of times.

The result is based on the observation that finding a certain *pseudo-saddlepoint* (PSP) of a matrix is sufficient. A PSP always exists, but may not be unique, and may not correspond to a strict saddlepoint (SSP) or even to a general saddlepoint (SP). However, if A has a SP or SSP, then its *value* equals the values of all PSPs of A. Finding a PSP (through a recursive approach) appears to be easier than finding a SSP directly, and having found a PSP, it is easy to locate a SSP with the same value, or to rule out its existence. The recursion can be bootstrapped starting from the Bienstock et al. $O(n \lg n)$ algorithm, obtaining a sequence of algorithms that eventually yield the following general result.

▶ **Theorem 1.** *Given an $m \times n$ or $n \times m$ matrix A, where $m \geq n$, we can determine whether A has a strict saddlepoint, and report such an entry, in $O(m \lg^* n)$ time.*

Our approach is deterministic, simple (despite the subtle runtime bound) and operates on the input matrix A only via queries and comparisons between entries. The result is close to optimal: to find the strict saddlepoint, its row and column must be fully queried, yielding an $m + n - 1$ lower bound on the number of operations. Whether an algorithm (perhaps randomized) with a linear runtime exists is an intriguing open question.

The result is also relevant to the general saddlepoint problem. Locating a (non-strict) saddlepoint, or even deciding if one exists, are subject to the $\Omega(n^2)$ lower bound of Llewellyn et al. [8]. However, if it is known that a saddlepoint exists, our algorithm can compute its *value* in the time given in Theorem 1.

After some preliminaries in §2, we introduce our main algorithm (proving Theorem 1) in successive steps, in §3, §4, and §5. In §6 we also describe an alternative (perhaps even simpler) approach that already improves upon the result of Bienstock et al., yielding a runtime of $O(m \lg \lg n)$ for $m \times n$ or $n \times m$ matrices, with $m \geq n$.

Further related work.   Hofri and Jacquet [6] study the complexity of Knuth's algorithms for matrices with distinct entries that are *randomly permuted*, and Hofri [5] also studies the distribution of the saddlepoint value and the probability of its existence in matrices with random entries; see also Knuth [7, §1.3.2].

2   Preliminaries

Denote $[n] = \{1, \ldots, n\}$ and $[a, b] = \{a, a + 1, \ldots, b\}$. The binary iterated logarithm $\lg^* n$ is defined as

$$\lg^* n = \begin{cases} 0 & \text{if } n \leq 1, \text{ and} \\ 1 + \lg^* (\lg n) & \text{if } n > 1. \end{cases}$$

Let A be a matrix with m rows and n columns, and let $a_{i,j}$ be the entry of A in row i and column j, for all $i \in [m]$ and $j \in [n]$.

A *saddlepoint* (SP) of A is an entry $a_{i,j}$ such that $a_{i,j} \geq a_{i,k}$ for all $k \in [n] - \{j\}$, and $a_{i,j} \leq a_{k,j}$, for all $k \in [m] - \{i\}$. In words, $a_{i,j}$ is the maximum in its row, and the minimum in its column. If $a_{i,j}$ is a SP with all inequalities being strict, then we call $a_{i,j}$ a *strict saddlepoint* (SSP) of A.

Note that the only assumption we make on the entries of the matrix A is that they are from an ordered set, allowing constant-time pairwise comparisons. In particular, we do not require matrix entries to be pairwise distinct.

Not all matrices admit a SSP or SP (take, for instance, the identity matrix of size $n > 1$), but if it exists, the SSP must be unique. Indeed, suppose that $a_{i,j}$ and $a_{i',j'}$ are two distinct SSPs. The cases $i = i'$ or $j = j'$ result in an immediate contradiction. Otherwise, from the definition of SSP, $a_{i,j} > a_{i,j'} > a_{i',j'} > a_{i',j} > a_{i,j}$, again, a contradiction.[4]

Pseudo-saddlepoint.   We next define a concept that is essential for our algorithms.

Let $a_{i,r(i)}$ denote a maximum of row i in A, i.e. $a_{i,r(i)} = \max\{a_{i,1}, \ldots, a_{i,n}\}$ for all $i \in [m]$, and let $a_{c(j),j}$ denote a minimum of column j in A, i.e. $a_{c(j),j} = \min\{a_{1,j}, \ldots, a_{m,j}\}$, for all $j \in [n]$ (breaking ties arbitrarily).

A *pseudo-saddlepoint* (PSP) of A is an entry $a_{i,j} = v$ of A such that every row of A has an entry larger or equal to v and every column of A has an entry smaller or equal to v. Equivalently, $a_{i,j} \leq a_{k,r(k)}$ for all $k \in [m]$, and $a_{i,j} \geq a_{c(k),k}$ for all $k \in [n]$. It follows that PSPs are exactly the entries with value in $[C, R]$, where C is the *maximum* of the column-minima $a_{c(k),k}$ and R is the *minimum* of the row-maxima $a_{k,r(k)}$.

▶ **Observation 1.** *Every matrix has at least one PSP.*

Proof. It is enough to show $C \leq R$. Indeed, let $C = a_{c(j),j}$ and $R = a_{i,r(i)}$. Then, since C (R) is the minimum (maximum) of its respective column (row), $a_{c(j),j} \leq a_{i,j} \leq a_{i,r(i)}$.   ◀

---

[4] Note that while at most one entry can be the SSP, its value may appear multiple times in the matrix.

The example $\left(\begin{smallmatrix} 0 & 7 & 5 \\ 6 & 4 & 2 \\ 3 & 1 & 8 \end{smallmatrix}\right)$ shows that PSPs of a matrix may have different values (here, $C = 2$ and $R = 6$, so all entries with value in $[2, 6]$ are PSPs, but the matrix admits no SSP, and in fact, no SP). The existence of a saddlepoint (strict or not), however, determines the value of all PSPs, as we show next.

▶ **Lemma 2.** *If a matrix A has a SP of value s, then every PSP of A has value s.*

Proof. Let $a_{i,j} = s$ be a SP of A. Then, $s \le a_{c(j),j} \le C$, by the definition of a SP and by the fact that C is the maximum of the column-minima. Thus, no value $v < s$ can correspond to a PSP. Symmetrically, $R \le a_{i,r(i)} \le s$, so no value $v > s$ can correspond to a PSP. ◀

Testing for a strict saddlepoint. The next ingredient of our approach is a feasibility test.

▶ **Lemma 3.** *Given an $m \times n$ matrix A and a value s, we can find, in time $O(m + n)$, a SSP of A of value s, or report that no such SSP exists.*

Proof. We perform two searches, both starting at location $(1, 1)$ of the matrix and proceeding in a monotone staircase fashion as follows. We call the first the *horizontal search*, and the second the *vertical search*, illustrated in Figure 3.
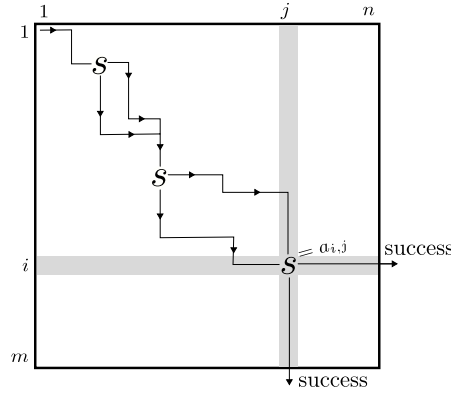
Figure 3 Horizontal and vertical searches in the proof of Lemma 3, for value s, with SSP $a_{i,j} = s$. Observe that the two paths can meet at arbitrary entries, but diverge only at entries with value s. After finding the SSP, the two searches only proceed horizontally, resp. vertically.

Horizontal search: Suppose we are at (i, j). If $i \in [m]$ and $j \in [n]$ then let $q = a_{i,j}$: If $s < q$, set $i \leftarrow i + 1$, and continue the search. Otherwise, set $j \leftarrow j + 1$, and continue the search. If $i > m$, halt with *failure*, if $j > n$, halt with *success*.

Vertical search: Suppose we are at (i, j). If $i \in [m]$ and $j \in [n]$ then let $q = a_{i,j}$: If $s \le q$, set $i \leftarrow i + 1$, and continue the search. Otherwise, set $j \leftarrow j + 1$, and continue the search. If $i > m$, halt with *success*, if $j > n$, halt with *failure*.

Observe that the two searches differ only in the tie-breaking in the case $s = q$ and whether they report success on exiting on the horizontal or vertical end of the matrix.

If any of the two searches halt with failure, return *false*. Otherwise, suppose the horizontal search exited at $(i, n + 1)$ and the vertical search exited at $(m + 1, j)$. Then, compare $a_{i,j}$ with all other entries in row i and in column j, returning *true* if $a_{i,j}$ is a SSP, and otherwise returning *false*.

Clearly, the runtimes of both searches and the final verification are $O(m + n)$. It remains to show, towards establishing Lemma 3, that the algorithm correctly identifies the SSP.

Due to the final verification step, the algorithm cannot falsely report a SSP. So suppose that there is a SSP $a_{i,j}$ of A. Then, both searches eventually reach an entry (i, j') with $j' \leq j$ or an entry (i', j) with $i' \leq i$. From then on, the searches proceed directly to $a_{i,j}$. The horizontal search then executes only $j \leftarrow j + 1$ steps, eventually returning success, and the vertical search executes only $i \leftarrow i + 1$ steps, eventually returning success. As the verification succeeds, the algorithm correctly identifies $a_{i,j}$ as the SSP.   ◀

We remark that in the algorithm of Lemma 3, when searching for a value s, the horizontal and vertical searches cannot both fail. Indeed, if the horizontal search exits "at the bottom" (at $(m + 1, j)$) and the vertical search exits "to the right" (at $(i, n + 1)$), then the two paths must diverge at an entry of value s "in the wrong way", i.e. the horizontal search moving vertically, and the vertical search moving horizontally, which is impossible. By distinguishing the other cases, we can turn the test into a parametric search tool, that will be useful in §6.

▶ **Observation 4.** *If in the algorithm of Lemma 3, when searching for a value s, the horizontal search fails, then the SSP (if exists) must be greater than s. If the vertical search fails, then the SSP (if exists) must be smaller than s. If both searches succeed and the final test fails, then the matrix has no SSP.*

Proof. If the horizontal search fails, then it has identified in each row an entry of value greater than s, thus the SSP must be greater than s. The second case is symmetric. In the third case, we learn that each row has an entry of value at least s and each column has an entry of value at most s. Thus, the SSP could only have value s.[5]   ◀

Reduction to square matrices.   We briefly argue that when computing PSPs, it is sufficient to focus on square matrices. Suppose, more generally, that the input matrix A is of size $m \times n$. Assume, w.l.o.g. that $m \geq n$, otherwise let $A = -A^T$, without affecting the structure of SSP or PSPs. (Explicitly transposing and negating A would of course be too costly, but we can do this "on demand", only for the queried entries.)

We reduce the computation of a PSP of A to computations on square matrices (similar arguments were used by Llewellyn et al. [8] and Bienstock et al. [1] for the SSP).

▶ **Lemma 5.** *Computing a PSP of an $m \times n$ matrix with $m > n$ can be reduced to the computation of PSPs of $\lceil m/n \rceil$ matrices of size $n \times n$, with an additive overhead of $O(m/n)$ on the runtime.*

Proof. Let A be an $m \times n$ matrix with $m > n$. Divide A into $\lceil m/n \rceil$ (possibly overlapping) matrices of size $n \times n$, compute a PSP for each of them, and return the one with minimum value v. Because that entry is a PSP of its corresponding matrix A', every column of A' (and thus every column of A) has an entry smaller or equal to v. Every row of A has a value larger or equal to the computed PSP of one of the smaller matrices containing this row, which is itself larger or equal to v. Thus, every row of A contains a value larger or equal to v.   ◀

▶ **Corollary 6.** *Given an algorithm that finds a PSP in an $n \times n$ matrix in time $O(n \cdot f(n))$ for arbitrary $f(n) \geq 1$, we can compute a PSP in an $m \times n$ or $n \times m$ matrix with $m \geq n$ in time $O(m \cdot f(n))$.*

---

[5] We also mention the somewhat counter-intuitive fact that for every matrix there is exactly one value for which both searches succeed (regardless of whether the matrix has a SSP or SP), and the procedure can be seen as searching for this value. As we lack an immediate use for this fact, we omit the (easy) proof.

3   A baseline algorithm for pseudo-saddlepoints

In this section we adapt the algorithm of Bienstock et al. [1] for finding the SSP, to find a PSP. For completeness, we repeat the analysis, although the modifications needed are minor.

Let $H = \{(r_1, c_1, v_1), (r_2, c_2, v_2), \ldots, (r_q, c_q, v_q)\}$ be a set of q triplets of the form (*row, column, value*), corresponding to entries in the input matrix A, with $v_1 \leq v_2 \leq \cdots \leq v_q$, and satisfying the following three properties:

- P1. H has at most one entry from each row or column of A.
- P2. Every row which does not appear in H has an entry larger or equal to v_q.
- P3. Every column which does not appear in H has an entry smaller or equal to v_1.

▶ **Lemma 7.** *Let $(i, j, a_{i,j}) = (r_1, c_1, v_1)$ and $(k, \ell, a_{k,\ell}) = (r_q, c_q, v_q)$ be the elements of H with minimum and maximum value respectively. By querying $a_{i,\ell}$ and doing a constant number of comparisons and insertions/deletions in H, we can reduce the size of H by one while preserving properties* P1*,* P2*, and* P3*.*

Proof. We know from P1 that $i \neq k$ and $j \neq \ell$. We query $a_{i,\ell}$ and distinguish three cases based on the comparisons with $a_{i,j}$ and $a_{k,\ell}$. Recall that $a_{i,j} = v_1 \leq v_q = a_{k,\ell}$.

Case 1: $a_{i,\ell} \leq a_{i,j} \leq a_{k,\ell}$. Then row k has an entry larger or equal to v_{q-1} (namely $v_q = a_{k,\ell}$). By P2 and the fact that $v_{q-1} \leq v_q$, every row which does not appear in H also has an entry larger or equal to v_{q-1}. Column ℓ has an entry smaller or equal to v_1 (namely $a_{i,\ell}$) and every column which does not appear in H also has an entry smaller or equal to v_1 (by P3). Thus, we can delete (r_q, c_q, v_q) from H while preserving all three properties.

Case 2: $a_{i,j} \leq a_{k,\ell} \leq a_{i,\ell}$. By a symmetric argument we can delete (r_1, c_1, v_1) from H while preserving all three properties.

Case 3: $a_{i,j} < a_{i,\ell} < a_{k,\ell}$. Let $u = \min\{v_2, a_{i,\ell}\}$ and $v = \max\{v_{q-1}, a_{i,\ell}\}$. The row k has an entry larger or equal to v (namely $a_{k,\ell}$). By P2 this is also the case for every row which does not appear in H. The column j has an entry smaller or equal to u (namely $a_{i,j}$). By P3 this is also the case for every column which does not appear in H. Thus, we can insert $(i, \ell, a_{i,\ell})$ and delete both (r_1, c_1, v_1) and (r_q, c_q, v_q) from H while preserving all three properties.   ◀

Given an $n \times n$ matrix A, we start by setting $H = \{(1, 1, a_{1,1}), (2, 2, a_{2,2}), \ldots, (n, n, a_{n,n})\}$. We then repeatedly apply Lemma 7 while maintaining H as a heap or a dynamically balanced binary search tree ordered by entry values (to guarantee $O(\log n)$ time per application of the lemma), until H has only one entry. By properties P2 and P3, this entry will be a PSP. We obtain the following theorem.

▶ **Theorem 8.** *Given an $n \times n$ matrix A, we can report a pseudo-saddlepoint (PSP) of A in time $O(n \log n)$, by querying at most $2n - 1$ entries of A.*

4   Bootstrapping the algorithm

We now improve the algorithm of Theorem 8, by embedding it into a recursive approach. The following lemma allows the decomposition of the input matrix, which is the key step in our subsequent algorithm.

▶ **Lemma 9.** *Let A be an $n \times n$ matrix. Let $R_1, R_2, \ldots, R_k$ and $C_1, C_2, \ldots, C_\ell$ be sets of indices, so that $R_1 \cup \cdots \cup R_k = [n] = C_1 \cup \cdots \cup C_\ell$. For all $i \in [k]$ and all $j \in [\ell]$, let A_{R_i, C_j} denote the submatrix of A obtained by taking the rows and columns of A with indices in R_i, resp. C_j. Let A' be a $k \times \ell$ matrix whose entry $a'_{i,j}$ is a PSP of A_{R_i, C_j} for all $i \in [k]$ and $j \in [\ell]$. Then all PSPs of A' are PSPs of A.*

Proof. Let v be the value of a PSP of A'. By definition, for every $i \in [k]$, there is some $j \in [\ell]$, so that $a'_{i,j} \geq v$. Because $a'_{i,j}$ is a PSP of A_{R_i,C_j}, every row of A with index in R_i has an entry larger or equal to $a'_{i,j}$. Thus, every row of A has an entry larger or equal to v. A symmetric argument shows that every column of A has an entry smaller or equal to v. ◄

We are ready to describe the main subroutine of our algorithm. For ease of presentation we start with a simpler version that already has an almost linear runtime bound.

▶ **Theorem 10.** *Given an $n \times n$ matrix A, we can report a pseudo-saddlepoint (PSP) of A in $O\big(n \cdot 2^{\lg^* n}\big)$ time.*

Proof. The algorithm works as follows.

- If $n = 1$, return the only entry of A.
- Otherwise, let $\ell = \lceil \lg n \rceil$, and divide the rows and columns of A each into $\left\lceil \frac{n}{\ell} \right\rceil$ (possibly overlapping) intervals of size ℓ. This divides A into $\left\lceil \frac{n}{\ell} \right\rceil^2$ (possibly overlapping) square matrices of size $\ell \times \ell$. We obtain a new matrix A' by conceptually replacing each smaller matrix with a PSP of that matrix.
- Run the algorithm of Theorem 8 on A'. Each time a new entry of A' is queried, run the current algorithm recursively on the corresponding submatrix of A to obtain the sought value.

The correctness of the algorithm follows directly from Lemma 9. Let us turn to the runtime analysis. In the first level of the recursion, the current algorithm runs the algorithm of Theorem 8 on a $\left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil \times \left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil$ matrix (costing $O(n)$ time) and makes $2\left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil - 1$ recursive calls on matrices of size $\lceil \lg n \rceil \times \lceil \lg n \rceil$. Thus, for all $n \geq 2$ and a large enough constant c, the runtime $T(n)$ of the algorithm obeys

$$T(n) \leq cn + \left(2 \left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil - 1\right) T(\lceil \lg n \rceil).$$

We show by induction that $T(n) \leq 2cn \cdot 2^{\lg^* n} - 3cn \lg^* n$ for all $n \geq 1$, and thus $T(n) \in O\big(n \cdot 2^{\lg^* n}\big)$. For $1 \leq n \leq 16$, the bound holds assuming c is large enough. Now assume that $n > 16$ and that the result is true for all values smaller than n. We have:

$$T(n) \leq cn + \left(2 \left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil - 1\right) T(\lceil \lg n \rceil)$$

$$\leq cn + \left(2 \left\lceil \frac{n}{\lceil \lg n \rceil} \right\rceil - 1\right) \left(2c\lceil \lg n \rceil \cdot 2^{\lg^*(\lceil \lg n \rceil)} - 3c\lceil \lg n \rceil \lg^*(\lceil \lg n \rceil)\right) \tag{1}$$

$$\leq cn + \left(\frac{2n}{\lceil \lg n \rceil} + 1\right) \left(2c\lceil \lg n \rceil \cdot 2^{\lg^*(n)-1} - 3c\lceil \lg n \rceil (\lg^*(n) - 1)\right) \tag{2}$$

$$\leq 2cn \cdot 2^{\lg^* n} - 3cn \lg^* n - 3cn \lg^* n + 7cn + c\lceil \lg n \rceil \cdot 2^{\lg^* n} \tag{3}$$

$$\leq 2cn \cdot 2^{\lg^* n} - 3cn \lg^* n - 12cn + 7cn + 5cn \tag{4}$$

$$\leq 2cn \cdot 2^{\lg^* n} - 3cn \lg^* n.$$

Here,

- (1) follows by induction,
- (2) uses that $\lceil x \rceil \leq x + 1$ for all x, and that $\lg^*(\lceil \lg n \rceil) = \lg^* n - 1$ for $n \geq 2$,
- (3) follows via simple manipulation and dropping a negative term,
- (4) uses the facts that $\lceil \lg n \rceil \cdot 2^{\lg^* n} < 5n$ for $n \geq 1$, and $\lg^* n \geq 4$ for $n > 16$. ◄

We remark in passing that an early stopping of the recursion would yield for all $k \in O(\lg^* n)$, a runtime of $n \cdot 2^{O(k)} \lg^{(k)} n$ with only $n \cdot 2^{O(k)}$ entries of A queried.

Our overall algorithm is as follows: Given an input matrix A of size $m \times n$ with $m \geq n$, first find a PSP s of A (via Theorem 10 and Corollary 6) in time $O\big(m \cdot 2^{\lg^* n}\big)$. Then, verify whether A admits a SSP of value s (via Lemma 3) in time $O(m + n)$. If yes, then report it, if not, then conclude (by Lemma 2) that A has no SSP. This yields an overall runtime of $O\big(m \cdot 2^{\lg^* n}\big)$.

Remark. Our algorithm can also be used for computing the *value* of the (non-strict) saddlepoint (SP), assuming that it exists, within the same runtime: we simply find a PSP s of A (via Theorem 10 and Corollary 6), and conclude (by Lemma 2) that the SP value is s. *Locating* a SP entry requires quadratic time in the worst case [8]. We can improve this, however, if the SP value s appears only few times in A. More precisely, we can locate a SP of value s in an $m \times n$ matrix A with k entries of value s in $O(k(m + n))$ additional time.

The approach is as follows: Run the horizontal search of Lemma 3 with value s. The search necessarily succeeds, finding in each column an entry of value at most s. The SP must be in a column where we encountered the value s. There are at most k such columns, so look through all of them in $O(mk)$ time to collect all candidate entries of value s (again, at most k of them). Test the candidates in $O(m + n)$ time each, for a total of $O(k(m + n))$.

5   Improved runtime

A closer look at the baseline algorithm of Theorem 8 reveals that only $n - 1$ of the $2n - 1$ queried entries are chosen adaptively during runtime; the remaining n entries are on the main diagonal $(a_{i,i})_{i \in [n]}$ of the input matrix A. A similar observation applies to the algorithm of Theorem 10 described in §4. Denoting $\ell = \lceil \lg n \rceil$, here $\lceil \frac{n}{\ell} \rceil$ of the $2\lceil \frac{n}{\ell} \rceil - 1$ recursive calls are for submatrices whose position is fixed upfront.

This suggests an improvement to the algorithm of Theorem 10, by solving $\lceil \frac{n}{\ell} \rceil$ of the subproblems directly, in an $O(n)$ time preprocessing step, and thereby reducing the number of recursive calls.

Let us first fix the decomposition of the input matrix A into submatrices of size $\ell \times \ell$ as follows. For $i = 0, \ldots, \lfloor n/\ell \rfloor - 1$, let $R_i = C_i = [i \cdot \ell + 1, (i + 1) \cdot \ell]$, and set the last (possibly overlapping) interval $R_{\lfloor n/\ell \rfloor} = C_{\lfloor n/\ell \rfloor} = [n - \ell + 1, n]$. We have $\cup_i R_i = \cup_i C_i = [n]$.

▶ **Lemma 11.** *Given an $n \times n$ matrix A, we can transform it in time $O(n)$ into an $n \times n$ matrix B, so that a PSP of value q of B implies a PSP of value q of A. Moreover, in $O(n)$ time we can compute a PSP of every "diagonal box" B_{R_i, C_i}.*

Lemma 11 serves as a preprocessing step for each call of the algorithm of Theorem 10. With this preprocessing, the algorithm of Theorem 10 needs to make only $\lceil \frac{n}{\ell} \rceil - 1$ recursive calls on matrices of size $\ell \times \ell$. As far as these recursive calls are concerned, the preprocessed matrix B is identical to A, up to permuting rows and columns, which can be maintained using straightforward bookkeeping.

Thus, for all $n \geq 2$ and a large enough constant c', the recurrence for the runtime becomes

$$T(n) \leq c'n + \left(\left\lceil \frac{n}{\ell} \right\rceil - 1 \right) T(\ell).$$

We show by induction that $T(n) \leq 2c'n \lg^* n$ for all $n \geq 1$, and thus $T(n) \in O(n \lg^* n)$:

$$
\begin{aligned}
T(n) &\leq c'n + \left(\left\lceil \frac{n}{\ell} \right\rceil - 1 \right) T(\ell) \\
&\leq c'n + \left(\left\lceil \frac{n}{\ell} \right\rceil - 1 \right) (2c'\ell \lg^* \ell) \\
&\leq c'n + 2c'n(\lg^*(n) - 1) \\
&\leq 2c'n \lg^* n.
\end{aligned}
$$

Together with Corollary 6 and Lemma 3 and Lemma 2, this implies our main result.

▶ **Theorem 1.** *Given an $m \times n$ or $n \times m$ matrix A, where $m \geq n$, we can determine whether A has a strict saddlepoint, and report such an entry, in $O(m \lg^* n)$ time.*

Here again one could stop the recursion early, yielding for all $k \in O(\lg^* n)$, a runtime of $O(n \lg^{(k)} n + nk)$ with only $O(nk)$ entries of A queried.

It remains to describe and analyze the preprocessing step.

Proof of Lemma 11. Given a square input matrix A, consider the following transformation, written as an in-place procedure, that results in a matrix B of the same size.

> **Transform**(A, t)
> **Input:** an $n \times n$ matrix A and stopping threshold t.
> 1: **if** $n \leq t$ **then halt**
> 2: $v \leftarrow$ **Select**$(\{a_{n,1}, a_{n-1,2}, \ldots, a_{1,n}\}, \lceil n/2 \rceil)$
> 3: **Partition** $(a_{n,1}, a_{n-1,2}, \ldots, a_{1,n})$ around v
> 4: $a_{i,i} \leftarrow v$ for $i = 1, \ldots, \lceil n/2 \rceil$
> 5: **Transform**$(A_{[\lceil n/2 \rceil + 1, n], [\lceil n/2 \rceil + 1, n]}, t)$

The transformation works as follows: select the median (element of rank $\lceil n/2 \rceil$) v of the antidiagonal $\{a_{n,1}, a_{n-1,2}, \ldots, a_{1,n}\}$. Then, partition the antidiagonal around v as detailed below, so that v goes into position $a_{n-\lceil n/2 \rceil + 1, \lceil n/2 \rceil}$ of the matrix, with entries smaller on its left and entries larger on its right.

Then, set the first $\lceil n/2 \rceil$ entries on the main diagonal to v. Finally, repeat the transformation recursively on the bottom right quadrant of the matrix, starting from an entry of the main diagonal. Stop when the matrix size falls below a stopping threshold t. The initial call is Transform$(A, 2 \lg n)$, to preprocess a matrix A of size $n \times n$, with stopping threshold $t = 2 \lg n$; the early stopping is to avoid affecting the rightmost (overlapping) boxes of A. The effect of the transformation is illustrated in Figure 4(a).

Let us first argue that the transformation takes linear time. Indeed, line 2 can employ linear time selection (e.g. [2]), and line 3 can be achieved by simulating a standard partitioning procedure on the antidiagonal, swapping pairs of entries by swapping the corresponding pairs of rows and columns. Note that when running line 3 in a recursive call of Transform, we still swap pairs of rows or columns of the full matrix. Notice that these operations can be implemented with simple bookkeeping in constant time per swap. Line 4 requires to copy the median element along the diagonal in $\lceil n/2 \rceil$ locations (this can be achieved e.g. by saving the diagonal into an array). Thus, lines 1–4 take cn time, for a sufficiently large c. With the recursive call in line 5, the total runtime of Transform(A) can be bounded as $c(n + n/2 + n/4 + n/8 + \cdots) \leq 2cn \in O(n)$.

We next argue that a PSP of the transformed matrix B implies a PSP of A of the same value. Recall that q is a PSP-value if and only if $q \in [C, R]$, where C is the maximum of the column-minima and R is the minimum of the row-maxima.
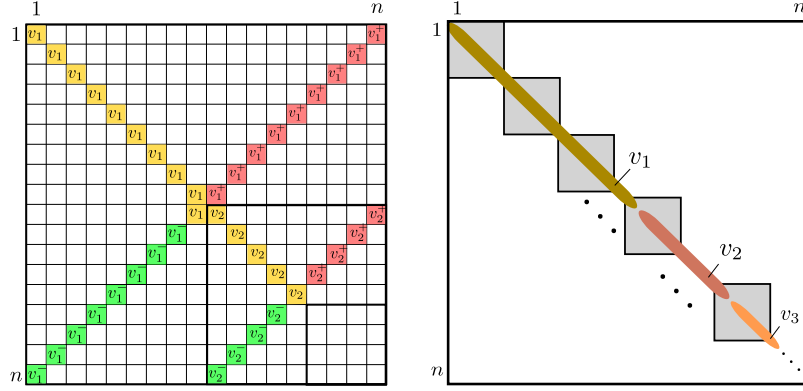
■ **Figure 4** (*a*) The effect of procedure Transform on an input matrix A of size 18×18: the antidiagonal is partitioned around the median, and the median value is copied to half of the main diagonal. The procedure is repeated recursively on the lower right quadrant. Values $v_1, v_2, \ldots$ denote the medians found in subsequent calls, and v_i^-, v_i^+ indicate values $\leq v_i$, resp. $\geq v_i$.
Note that swaps during the partitioning step of recursive call i may move elements away from the antidiagonal from call $j < i$. For simplicity, this is not reflected in the figure.
(*b*) Solving the diagonal-subproblems in the algorithm of Theorem 10. Squares indicate $\lceil \lg n \rceil \times \lceil \lg n \rceil$ size subproblems. All, but $\lg n + O(1)$ of these have uniform diagonal.

Initially $B = A$, so the PSPs are the same. Consider a call of Transform at an arbitrary level of recursion. Swapping pairs of rows or columns does not affect the PSP values, thus the claim holds for lines 1–3. Copying the median value v in line 4 cannot decrease C, since the affected column already contains a value at most v due to the partitioning step. Similarly, it cannot increase R, since the affected row already contains a value at least v (see Figure 4(a)). Thus, denoting by C', R' the new maximum of column-minima, resp. minimum of row-maxima, we have $[C', R'] \subseteq [C, R]$, so no new PSPs are created.

It remains to compute a PSP for each diagonal box B_{R_i, C_i}. Notice that the diagonals of these boxes coincide with the diagonal of the full (preprocessed) matrix B. Moreover, for all but at most $\lg n + O(1)$ of the boxes B_{R_i, C_i}, their diagonal contains a single value. This is because, after preprocessing, the diagonal of B consists of at most $\lg n$ contiguous uniform sections (corresponding to calls of Transform), and a last section of length at most $2 \lg n$, that was unaffected by Transform. Only boxes that intersect with the boundaries between sections, and a constant number of boxes at the end will have a non-uniform diagonal; see Figure 4(b) for an illustration.

Notice that if every diagonal entry of a matrix is v, then v is a PSP of the matrix. Thus, all but $\lg n + O(1)$ of the diagonal boxes have their PSP readily available. For the remaining diagonal boxes, we call the baseline algorithm of Theorem 8, adding a term $O(\lg n \cdot \lg n \lg \lg n) \subset O(n)$ to the runtime. The total runtime is $O(n)$, finishing the proof.   ◄

6   An alternative approach

In this section we briefly describe an algorithm for the SSP problem with a runtime of $O(n \lg \lg n)$ on an $n \times n$ matrix A. While the bound is weaker than the previous ones, we find the approach worth mentioning due to its simplicity, and since it can run faster on certain inputs.

The algorithm has two phases. The *first phase* makes use of Observation 4 to find progressively better upper and lower bounds on the value of the SSP (should it exist), and

thereby eliminate rows or columns of A that cannot contain the SSP. The *second phase* begins when one side-length of the matrix has been reduced to at most $\frac{n}{\lg n}$. It makes use of a heap, similarly to the algorithm of Theorem 8, but with a different purpose: to also reduce the *longer side* of the matrix to about $\frac{n}{\lg n}$. When both sides of the matrix have length $O(\frac{n}{\lg n})$, we can finish the job with the baseline algorithm of Theorem 8, with a total runtime of $O(n)$. If a SSP is found, we perform an additional $O(n)$ time test with it on the original matrix, to rule out a false positive.

Recall that the search procedure of Lemma 3 returns (by Observation 4), in $O(m + n)$ time, for an $m \times n$ or $n \times m$ matrix A and a search value s, one of four answers: (1) A has the SSP $a_{i,j} = s$; (2) A has no SSP; (3) the SSP of A (if exists) is $> s$; or (4) the SSP of A (if exists) is $< s$.

First phase. Let A denote the current $m' \times n'$ matrix, where $n \geq m' \geq n'$ (the other case is symmetric). Assume that entries of A are $a_{i,j}$ with $i \in [m']$, resp. $j \in [n']$.

Compute the median v of the set of elements $D = \{a_{i,\lceil \frac{i \cdot n'}{m'} \rceil} \mid i \in [m']\}$ and search A with v using the procedure of Lemma 3. If we learn that the SSP can only be larger than v, then recurse on $A_{[m'],[n']-C'}$ where $C' = \{j \mid a_{i,j} \in D \wedge v \geq a_{i,j}\}$. If we learn that the SSP can only be smaller than v, then recurse on $A_{[m']-R',[n']}$ where $R' = \{i \mid a_{i,j} \in D \wedge v \leq a_{i,j}\}$. If we find the SSP, or learn that a SSP does not exist, then halt accordingly.

Observe that the reductions are justified. Indeed, if the SSP must be larger than v, then it cannot be in columns that contain entries smaller or equal to v, and hence, these columns can be deleted. If the SSP must be smaller than v, then it cannot be in rows that contain entries larger or equal to v, and hence, these rows can be deleted.

If either side-length of the matrix is reduced to $\frac{n}{\lg n}$, then proceed to the next phase.

Second phase. Let A denote the current $m' \times n'$ matrix, where $n \geq m' \geq n'$ (the other case is symmetric). Assume $n' \leq \frac{n}{\lg n}$, and $m' > 4n'$ (otherwise we can stop). For each column $j \in [n']$, select $\lfloor \frac{m'}{2n'} \rfloor$ rows $R_j \subseteq [\lfloor m'/2 \rfloor]$, so that $R_j \cap R_{j'} = \emptyset$ whenever $j \neq j'$. Observe that initially only the first half of the m' rows are picked. For each column j calculate the minimum $m_j = \min\{a_{i,j} \mid i \in R_j\}$.

Insert the minima $m_1, \ldots, m_{n'}$ into a max-heap. For n' iterations, extract the maximum element from the heap. Suppose the currently extracted maximum is $m_j = a_{i,j}$. Then for all $i' \in R_j - \{i\}$, delete row i' of the matrix A.

This is justified, since a SSP cannot exist in row i' of the matrix. Indeed, if such a SSP $a_{i',k}$ existed, then either (1) $k = j$ and we have a contradiction since $m_j \leq a_{i',j}$ (by the choice of m_j as the minimum), or (2) $k \neq j$ which is a contradiction since $m_k \leq m_j \leq a_{i',j} < a_{i',k}$ (by the maximality of m_j in the heap and the row condition of SSPs). Either case contradicts that $a_{i',k}$ is a SSP since it is not strictly smaller than some value in its column.

Now, in the column j, select $\lfloor \frac{m'}{2n'} \rfloor$ new elements with row index set $R_j \subseteq [m']$ disjoint from all other $R_{j'}$ with $j \neq j'$. Compute the new minimum $m_j = \min\{a_{i,j} \mid i \in R_j\}$, and reinsert m_j into the max-heap.

In n' iterations, the process removes a constant fraction of rows. Repeat the phase $O(\lg \lg n)$ times, to reduce the number of rows to $m' \leq 4n'$.

Running time. Starting with an initial $n \times n$ matrix, every iteration of the first phase runs in $O(n)$ time and removes at least a constant fraction of the remaining rows or columns,

which implies that in $O(n \lg \lg n)$ time at least one side will be reduced to length $\frac{n}{\lg n}$.[6]

In the second phase, initializing R_j and m_j for each column j takes $O(n)$ total time. Then, each of the n' iterations involve a constant number of heap operations of cost $O(\lg n)$, and a constant cost per the removal of each row. Since $n' \in O(\frac{n}{\lg n})$, the execution of the phase takes $O(n)$ total time. Repeated $O(\lg \lg n)$ times, this yields the total runtime of $O(n \lg \lg n)$.

---- **References** ----

1   Daniel Bienstock, Fan Chung, Michael L. Fredman, Alejandro A. Schäffer, Peter W. Shor, and Subhash Suri. A note on finding a strict saddlepoint. *Am. Math. Monthly*, 98(5):418–419, April 1991. `doi:10.2307/2323858`.

2   Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. `doi:10.1016/S0022-0000(73)80033-9`.

3   Christopher C. Byrne and Leonid N. Vaserstein. An improved algorithm for finding saddlepoints of two-person zero-sum games. *Int. J. Game Theory*, 20(2):149–159, June 1991.

4   Stephen Hedetniemi. Open problems in combinatorial optimization. `https://people.computing.clemson.edu/~hedet/algorithms.html`. Accessed: 2023-08-07.

5   Micha Hofri. On the distribution of a saddle point value in a random matrix. *Department of Computer Science, WPI*, 100, 2006.

6   Micha Hofri and Philippe Jacquet. Saddle points in random matrices: Analysis of Knuth search algorithms. *Algorithmica*, 22(4):516–528, 1998. `doi:10.1007/PL00009237`.

7   Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., USA, 1997.

8   Donna Crystal Llewellyn, Craig Tovey, and Michael Trick. Finding saddlepoints of two-person, zero sum games. *The American Mathematical Monthly*, 95(10):912–918, 1988. `doi:10.1080/00029890.1988.11972116`.

9   Michael Maschler, Shmuel Zamir, and Eilon Solan. *Game theory.* Cambridge University Press, 2020.

---

[6]   Note that if iterations alternate between removing rows and columns, then the runtime can be described by a geometric series that evaluates to $O(n)$.