Finding Maximum and Minimum Size Matrices: The Algorithmic Complexity of Coding Challenges

Abdelrahman Abdelmonsef
Department of Computer Science, Swarthmore College, PA, USA

Xingyu Dong
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA

Daniel Průša
Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic

Michael Wehar
Department of Computer Science, Bryn Mawr College, PA, USA

Chen Xu
Department of Computer Science and Engineering, University at Buffalo, NY, USA

Abstract

We investigate coding challenge problems related to finding a maximum (or minimum) size match for a predetermined two-dimensional pattern. We improve upon known runtime results by introducing new algorithms and refining existing algorithmic techniques. First, we present our main result which introduces a nearly quadratic time algorithm for the problem of finding a rectangular block within a matrix of maximum (or minimum) size containing only positive border entries which improves the prior cubic time solution. Then, we introduce a quadratic time and linear space algorithm for the problem of finding all rectangular blocks containing only positive border and empty interior entries. Finally, we present a $\log(n)$ factor improvement for detecting the largest length such that all square blocks in a matrix have their sums bounded by a given number.

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

Keywords and phrases Pattern Matching, Matrices, Discrete Algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.1

Acknowledgements Preliminary research was pursued by the 1st, 2nd, 4th, and 5th authors in collaboration with the Algorithms Research Group at Swarthmore College during Spring and Fall 2022. We would especially like to thank and recognize the helpful discussions with B. Xiang, D. Yang, E. Brickner, J. Mancini, J. Swernofsky, N. Almadbooh, and all 19 individuals who participated in the Algorithms Research Group. In addition, we would like to thank K. Regan as well as anonymous reviewers for helpful suggestions.

1 Background

1.1 Coding Challenges and 2D Pattern Matching

Coding challenge problems have become increasingly popular in recent years. These challenges typically involve being given an algorithmic problem and having to find and code up an efficient solution. Users of these platforms even compete based on a variety of factors including finding the most efficient solution. In this work, we investigate coding challenge-styled problems that relate to two-dimensional pattern matching. In particular, we focus on finding largest and smallest blocks within matrices that match a predetermined pattern. The patterns that we focus on are simple constraints on either the matrix's corner, border, or interior entries. For example, given an m by n matrix, it is a classic algorithmic problem

to find a maximum area rectangular subblock within the matrix consisting of only 1's. We denote this problem by `Max-Rect-Ones`. This problem commonly appears in online coding challenges [2, 7]. We also consider a variant of this problem denoted by `Max-Square-Ones` where we restrict the subblock to be a square consisting of only 1's. This variant also appears as a coding challenge [1]. It is well known that both of these problems can be solved in $O(mn)$ time [14]. Furthermore, a generalized form of this problem was considered in [15]. In the more general form, we are searching for a maximum area subblock that matches a Border-Unaware Local Picture Language over a binary alphabet. That is, a set of 2 by 2 binary tiles T such that every 2 by 2 subblock of the matrix is in T. If $T = \{\left[\begin{smallmatrix}1 & 1\\ 1 & 1\end{smallmatrix}\right]\}$, then we simply get the `Max-Rect-Ones` problem. For any set of 2 by 2 binary tiles T, finding a maximum area subblock that matches T can be solved in $O(mn)$ time [15].

1.2 Related Problems for Corner Patterns

Even when only considering patterns restricting corner entries, finding maximum and minimum matches can be challenging. Let a 2 by 2 pattern matrix $\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$ be given. The Max Four Corners Problem for Boolean matrices is defined as follows. Given an m by n Boolean matrix, find a maximum area rectangular subblock with upper left corner a, upper right corner b, lower left corner c, and lower right corner d. We denote this problem by `Max-Corners-`$\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$. We also consider a variant of this problem denoted by `Min-Corners-`$\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$ where we are searching for the minimum area subblock matching the corner pattern. It was shown in [18] that `Min-Corners-`$\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$ is solvable in $\tilde{O}(mn)$ time for all corner patterns $\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$. Also, it was shown that `Max-Corners-`$\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$ is triangle finding-hard. Furthermore, it was shown that `Max-Corners-`$\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]$ is solvable in $O(mn(min\{m,n\})^{0.5302})$ time by a reduction to the minimum witness for Boolean matrix multiplication problem.

2 Introduction

2.1 Motivation

Classically, pattern matching in two dimensions has been found to have applications to data queries [21], computational geometry [22], and computer graphics [20]. In this work, we focus on three problems related to maximum and minimum size subblocks within matrices that match simple patterns. We will see that these problems appear in coding competitions (e.g. USACO 2016, KPI-OPEN 2017), online coding platforms (e.g. GeeksforGeeks), and online forums (e.g. StackExchange). For these problems, we introduce efficient algorithms that in some cases improve upon established runtime upper bounds and in others common folklore.

2.2 Problem Statements

First, we investigate the Maximum Rectangle with Positive Border problem denoted by `Max-Area-Border`. This problem is defined as follows. Given an m by n matrix with integer values (typically only 0's and 1's), find a largest area subblock whose border consists of only positive values. This problem appeared in [16] and has a known cubic time solution [19]. We also consider a variant of the problem denoted by `Max-Perimeter-Border` where we are searching for a largest subblock in terms of perimeter rather than area. The perimeter problem appeared in [5] and has a known cubic time solution [4]. In addition, we consider the `Min-Area-Border` and `Min-Perimeter-Border` problems that are concerned with finding smallest area and perimeter (non-trivial) subblocks, respectively.

Second, we investigate a related problem that we refer to as All Rectangles with Positive Border and Empty Interior. We denote this problem by `All-Empty-Interior`. For this problem, we are searching for all rectangular subblocks with only positive border entries and only 0's as interior entries.

Finally, we investigate a problem that we refer to as Maximum Squares with Upper Bounded Sums and denote it by `Max-Bounded-Sums`. In this problem, we are searching for a maximum side length such that all square subblocks with this side length have a sum that is bounded by a given number. A dual variant of this problem is referred to as Minimum Squares with Lower Bounded Sums and is denoted by `Min-Bounded-Sums`.

2.3 Algorithmic Results

In the following, we list the pattern matching problems investigated in this paper along with a brief statement of our primary runtime improvements for each of these problems. In the following, we will use m to denote the number of rows and n to denote the number of columns in the input matrices.

- `Max-Area-Border`: Maximum (or Minimum) Positive Border Problem
  This is the problem of detecting a maximum (or minimum) area subblock within a matrix that contains only positive values along the border. For this task, we achieve a runtime of $\tilde{O}(mn)$. In addition, we show the same time bound for the problem of maximizing perimeter. This improves the known cubic runtime bounds for the area and perimeter cases from [19] and [4], respectively.
- `All-Empty-Interior`: All Positive Border with Empty Interior
  This is a variation on the previous problem where the task is to find and list all subblocks within a matrix that contain only positive values along the border and zeros within the interior. For this task, we achieve a runtime of $O(mn)$ while using at most $O(n)$ space.
- `Max-Bounded-Sums`: Maximum (or Minimum) Squares with Bounded Sums
  This is the problem of finding the maximum side length such that all square subblocks with this side length have a sum of at most k. For this task, we achieve a runtime of $O(mn)$ over input matrices with non-negative entries. This improves the prior $O(mn \log(m+n))$ time algorithm [6].

3 Rectangles with Positive Border

3.1 Maximizing Area and Perimeter

When searching for matches within a matrix, it is natural to seek out a largest or smallest such match. In two dimensions, by largest (smallest) we typically mean a matching rectangular block with maximum (minimum) area unless otherwise stated. A classic example of this (previously mentioned in Section 1.1) is the problem of searching in a Boolean matrix for a maximum area subblock of all 1's (denoted by `Max-Rect-Ones`). This problem is well known to be solvable in $O(mn)$ time for m by n input matrices [14]. In this section, we consider a variant of this classic problem where the border of the subblock contains only 1's rather than the interior. We denote this problem by `Max-Area-Border`. The `Max-Area-Border` problem appeared in the USACO 2016 competition [16] and a cubic time solution is known [19]. In addition, we denote by `Min-Area-Border` a variation of this problem where we are seeking a minimum matching subblock. See Section 2.2 for a more detailed definition.

▶ **Theorem 1.** *`Max-Area-Border` and `Min-Area-Border` are solvable in time*

$O(m \cdot n \cdot \log(min\{m, n\}))$.

Proof. Given an m by n Boolean matrix $\mathcal{M}$. Without loss of generality, $m \geq n$. Therefore, we have $n = min\{m, n\}$. We proceed with a divide and conquer strategy with a similar format to that presented in [18] for solving the `Min-Corners-`$\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$ problem. First, we present the algorithm's overall approach. Then, we focus on the split cases and the map construction process in more detail.

Algorithm Overview. Initially, we spend $O(m \cdot n)$ time to do some preprocessing. Then, we split $\mathcal{M}$ horizontally $\left(\frac{m}{n} - 1\right)$ times so that we have $\frac{m}{n}$ matrices that are n by n. We call it a split case when we are considering positive border rectangles that stretch across the split between two vertically-adjacent or horizontally-adjacent matrices. Notice that these initial $\left(\frac{m}{n} - 1\right)$ horizontal split cases are not entirely independent from each other as a rectangle could span across multiple splits. We will see that each split case takes $O(n^2)$ time to handle. In total, these initial split cases take $O(m \cdot n)$ combined time.

Now that these initial split cases have been handled, we focus on positive border rectangles that are contained solely within one of the n by n matrices. For each n by n matrix M, we proceed by further subdividing the problem with horizontal and vertical splits. For example, if M was horizontally split into M_{top} and M_{bottom}, then a maximum area rectangle with a positive border in M is either in M_{top}, M_{bottom}, or covered by the split case where the rectangle stretches across the boundary between M_{top} and M_{bottom}. Similar to before, each split case will take $O(n^2)$ time. Hence, by combining horizontal and vertical splits we get four subcases each with an $\frac{n}{2}$ by $\frac{n}{2}$ matrix. This leads us to the recurrence relation $t(n) = 4 \cdot t(\frac{n}{2}) + O(n^2)$. Solving the recurrence, we get $t(n) = O(n^2 \cdot \log(n))$ [8]. Therefore, since there are $\frac{m}{n}$ many n by n matrices to consider, the algorithm's total runtime is $O(m \cdot n \cdot \log(n))$.

Split Cases. We proceed by further describing how we efficiently handle split cases. Without loss of generality, we focus on horizontal splits. For an n by n matrix M that is horizontally split into M_{top} and M_{bottom}, our task is to construct maps m_{top} and m_{bottom}. As input, these maps take in pairs of column indexes $\{j_1, j_2\}$. For the `Max-Area-Border` problem, the map m_{top} outputs the smallest row index i in M_{top} such that all entries between $M_{top}(i, j_1)$ and $M_{top}(i, j_2)$ are 1's, all entries below $M_{top}(i, j_1)$ are 1's, and all entries below $M_{top}(i, j_2)$ are 1's. The map m_{bottom} outputs the largest row index i in M_{bottom} such that all entries between $M_{bottom}(i, j_1)$ and $M_{bottom}(i, j_2)$ are 1's, all entries above $M_{bottom}(i, j_1)$ are 1's, and all entries above $M_{bottom}(i, j_2)$ are 1's. If we are instead solving the `Min-Area-Border` problem, then m_{top} outputs the largest row index and m_{bottom} outputs the smallest.

Once we have constructed the m_{top} and m_{bottom} maps, we straightforwardly compute a maximum (minimum) area rectangle with a positive border that stretches across the boundary of M_{top} and M_{bottom}. In particular, for any pair of column indexes $\{j_1, j_2\}$, if defined, the rows

$$\{m_{top}(\{j_1, j_2\}), m_{bottom}(\{j_1, j_2\})\}$$

and columns $\{j_1, j_2\}$ together define a positive border rectangle. We select the maximum (minimum) area rectangle among all column pairs $\{j_1, j_2\}$.

Things are a little more complex for the initial $\left(\frac{m}{n} - 1\right)$ horizontal splits cases. We still construct the m_{top} and m_{bottom} maps for each n by n matrix, but we need to carefully reuse work between vertically-adjacent matrices because a rectangle could extend beyond multiple

horizontal splits. To do this, we construct all of the m_{top} maps in order from top to bottom, reusing the previous vertically-adjacent matrix's m_{top} map for rectangles that might spans across multiple splits. Similarly, we construct m_{bottom} maps in order from bottom to top, reusing the previous matrix's m_{bottom} map.

Map Construction. Given an n by n matrix that is horizontally split into M_{top} and M_{bottom}, all that remains is to describe how to construct the maps m_{top} and m_{bottom} in $O(n^2)$ time. Without loss of generality, we focus on constructing m_{top} for the `Max-Area-Border` problem. First, we preprocess the matrix M_{top} so that for any entry $M_{top}(i,j)$ we have constant time lookup of the closest 0 value stored to the left. That is, we have constant time lookup of the largest $j' < j$, if it exists, such that $M_{top}(i,j') = 0$. Similarly, we need constant time lookup of the closest 0 value stored above any entry.

Next, let a column j_2 be given. We demonstrate how to compute $m_{top}(\{j_1, j_2\})$ for all $j_1 < j_2$ in $O(n)$ total time. We proceed as follows. Start with row index $i = k$ where $k-1$ is the bottom-most row with a 0 value in column j_2. Then, put row index i into a group (or bucket) based on its distance to the closest 0 value to the left of entry $M_{top}(i, j_2)$. Now, increment i and repeat. Row indexes with the same distance will be put into the same group. Within groups, row indexes are in ascending order.

Following this, we create a union-find data structure starting with singleton sets $\{i\}$ for each row index i, adding one extra row to the bottom of M_{top}. We iterate through column indexes $j_1 < j_2$ starting with $j_1 = 1$. We consider the associated distance $d = j_2 - j_1 + 1$. Now, we iterate in order over the rows in the distance d group. For each row i, we perform a union between the set containing i and the set containing $i+1$. Next, we perform a find on value k where $k-1$ is the bottom-most row with a 0 value in column j_1. Whenever we perform this find, we determine the value of $m_{top}(\{j_1, j_2\})$. Then, we increment j_1 and repeat until we have computed $m_{top}(\{j_1, j_2\})$ for all $j_1 < j_2$.

This almost works, the only issue is that the find operation needs to return the minimum element in the set. To fix this, we use an auxiliary array to store the minimum values in each set. Basically, the array will be used to map each set's representative element to its minimum element. Furthermore, with each union operation that we perform, we will also need to perform a find operation to compute and update the new set's minimum element.

The surprising thing about this construction is that for each column j_2, the resulting sequence of unions and finds is predetermined. In other words, we can apply the special case of union-find from [13] to perform these $O(n)$ operations in $O(n)$ total time. This is quite remarkable since union-find usually takes $O(n \cdot \alpha(n))$ time to perform $O(n)$ operations. Therefore, since there are $O(n)$ possible values for j_2, the total runtime is $O(n^2)$ for the split case.[1] For the `Min-Area-Border` problem, we can perform a similar strategy that achieves the same runtime where we map representative elements to maximum elements and always perform finds on the bottom-most row index. ◀

A similar problem, denoted by `Max-Perimeter-Border`, for maximizing perimeter instead of area appeared in the KPI-OPEN 2017 competition [5]. A cubic time solution is known for this problem [4]. We claim that the approach from Theorem 1 can be similarly applied to improve efficiency. The essential part in proving Theorem 1 is to identify a largest rectangle with positive border that crosses the split in at most $O(n^2)$ time which makes the total

[1] We acknowledge an anonymous reviewer who suggested the use of this special union-find strategy for the split cases. Our original approach was to use a binary heap, but this adds a $\log(n)$ factor to the runtime. One could alternatively use a Van Emde Boas Tree instead of a binary heap to improve efficiency.

runtime's recurrence relation work as presented. To find a largest rectangle that crosses the split, we first find all of the rectangles with maximal height that cross the split. We then pick out the rectangle of largest area from these. If instead, we pick out the rectangle with largest perimeter, this will still work because the rectangle with largest perimeter must also be a rectangle of maximal height. Therefore, we have the following corollary.

▶ **Corollary 2.** *`Max-Perimeter-Border` and `Min-Perimeter-Border` are solvable in time*

$O(m \cdot n \cdot \log(min\{m, n\}))$.

3.2 Enumerating Rectangles With Empty Interior

In an n by n matrix, there are $\Theta(n^4)$ rectangular subblocks. Therefore, we can have at most $O(n^4)$ rectangular subblocks that match a given pattern. Furthermore, brute force checking every matching subblock could be a costly approach to find a maximum or minimum size subblock matching the pattern. The runtime could be $\Omega(n^4)$ or greater in the worst case.

However, there are pattern matching problems where the number of possible matches is $o(n^4)$ because of the nature of the pattern. One example is the square of all ones pattern [9], but this is trivial because there are $\Theta(n^3)$ square subblocks in an n by n matrix. Another example of such a pattern is from the All Positive Border with Empty Interior Problem denoted by `All-Empty-Interior`. The goal of this problem is to find all rectangular subblocks with only positive border and 0's as interior entries. For this problem, there can be at most $O(n^2)$ possible matches. In fact, we will see that we can find all such matches in at most $O(n^2)$ time.

The strategy is to scan row by row and update at each moment when we enter, stay within, or exit a potential rectangular matching subblock. We will keep an array to record every "staple"-shape of 1's which consists of a top, left, and right edge containing only 1's with 0's everywhere else. We output the rectangle when the staple is closed by an all 1's interval as a bottom edge.

▶ **Theorem 3.** *`All-Empty-Interior` is solvable in $O(mn)$ time and $O(n)$ space.*

Proof. Consider a non-trivial subblock matching the positive border and empty interior pattern. Suppose that its left and right edges are at column indices (l, r) and its top and bottom edges are at row indices (t, b). We know that from the top of the matrix to the bottom, the interaction of each row with a matching subblock can only take one of four forms:

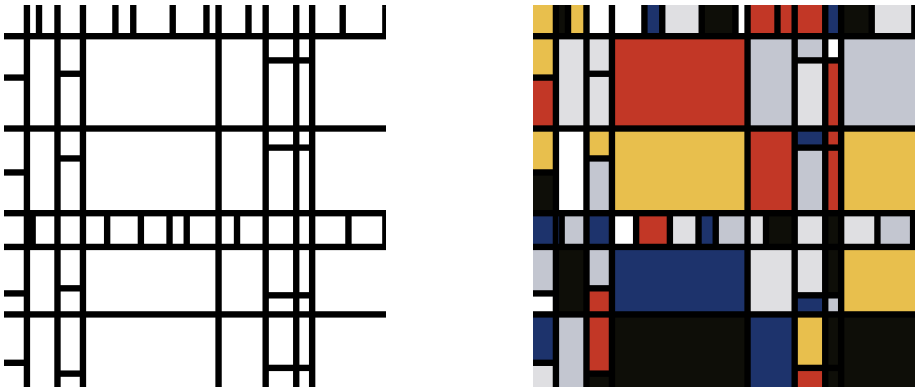
- F_1: The current row is not intersecting with this subblock.
- F_2: The current row is hitting the top edge t of this subblock.
- F_3: The current row is passing through the left and right edges (l, r) of this subblock.
- F_4: The current row is hitting the bottom edge b of this subblock.

It is easy to see that at any moment, there can be at most $O(n)$ subblocks matching with form F_2, F_3, or F_4 because a match cannot overlap within another match. We can therefore maintain a triple (t, l, r) (informally called a "staple") to represent a partial pattern match alongside a row. We can use the intervals of 1's from the current row and the previous row to update the current triples. To handle the transitions from F_1 to F_2, we just record the current row number and the consecutive 1 intervals as (t, l, r) triples for partial pattern matches. To handle the transitions from F_2 to F_3, we utilize the information from the intervals of 1's from the previous row and the current row to break each triple (t, l, r) into multiple triples (t, l_i, r_i), if necessary. To handle the transitions from F_3 to F_4, we match with the intervals

of 1's of the current row to set the bottom edge b for any (t, l, r) and output a pattern match. We reject a partial match when we find that the interval of 1's of the corresponding form cannot lead to a closed bottom edge. In that case, we remove the recorded triple accordingly.

We perform the updates for each type of transition mentioned above for each row and this can be done in $O(n)$ time. Therefore, the total runtime is $O(mn)$ and space usage is $O(n)$ because we only keep track of at most $O(n)$ triples at a time. Because matches are symmetrical, we can rotate the matrix if needed to ensure that $n \leq m$. For space, we ignore log factors that could result from how numeric values are stored as well as additive log terms from the special case when 2^n is $o(m)$. ◀

The preceding algorithm has several advantages over alternative approaches that we have explored. In particular, it scans over all of the elements of the matrix only one time, and while it is scanning across the matrix, it only uses $O(n)$ working memory. The 2nd and 4th authors implemented the algorithm from Theorem 3 in JavaScript to identify and color empty rectangles in images, as seen in Figure 1, demonstrating the algorithm's usefulness for computer art and graphics applications [11, 3].



(a) Original image without coloring. **(b)** Image after rectangles are identified and colored.

Figure 1 The algorithm was applied to color an image in the style of a Piet Mondrian artwork [12].

4 Maximum and Minimum Squares With Bounded Sums

Rather than searching for a particular match, we could query how many matches there are in total. Unfortunately, for many patterns, the only approach we currently know for counting matches is to find all of the matches one by one. However, we suggest that, for some patterns, one may be able to count all of the matches without explicitly enumerating through each one. In addition to counting, there could be more complex queries that one might ask about the set of matches where finding and listing all of the matches might not be necessary to resolve the query.

In the following, we consider a coding challenge problem from [6] that is defined by a more complex query that does not require finding and listing all matches. In particular, the query we investigate asks for the maximum side length such that every square subblock of this side length within the matrix has bounded sum. We denote this problem by `Max-Bounded-Sums` and it is more formally stated as follows. Given an m by n matrix M and a number k such that M contains non-negative integer entries. What is the maximum value l such that every l by l square subblock within M has a sum of at most k?

The previous solution from [6] runs in $O(mn \log(m+n))$ time. We propose a $\log(m+n)$ time factor improvement over the prior approach. Let $s(i,j,k,l)$ be the partial sum of the subblock whose top left corner is (i,j) and bottom right corner is (k,l). We have

$$s(i,j,k,l) = s(i,j,m,n) - s(k+1,j,m,n) - s(i,l+1,m,n) + s(k+1,l+1,m,n) \quad (1)$$

Consider fixing $k = m$ and $l = n$. The restricted function $s(\cdot,\cdot,m,n)$ can be precomputed in $O(mn)$ time. Hence, a value of $s(i,j,k,l)$ can be computed in constant time by looking up four precomputed sums. This is a straightforward special case of 2D Range Minimum Query (RMQ) [23]. We proceed to the following theorem.

▶ **Theorem 4.** *`Max-Bounded-Sums` is solvable in $O(mn)$ time.*

Proof. To solve this problem, we scan across each entry of the matrix checking if the square subblock with the current entry as the upper left corner has a sum less than or equal to k. If it doesn't, then we decrement the side length and continue scanning. The small, but clever, insight here is that we do not need to go backwards and check that the prior entries, along with the smaller side length, result in a sum less than or equal to k. This is guaranteed because the sum is bounded above by the sum from the square subblocks with any larger side length since the entries are all non-negative.

The preprocessing step takes $O(mn)$ time. Then, computing the sums for at most $O(mn)$ square subblocks takes only constant time per sum. Therefore, the total runtime is $O(mn)$. ◀

In addition, the same technique as in the preceding proof can be applied if we take products of entries instead of sums. However, in this case, the multiplications and divisions could end up being costly operations because the size of the stored values could grow exponentially as we multiply entries. Furthermore, the bounded sums problem has a dual problem denoted by `Min-Bounded-Sums` where you need to find the minimum side length l such that every l by l square within M has a sum of at least k. Our approach can also be applied to solve the dual problem in $O(mn)$ time.

▶ **Corollary 5.** *`Min-Bounded-Sums` is solvable in $O(mn)$ time.*

It is important to notice that the requirement that the matrix entries are non-negative is important. If we allow for the matrix entries to be negative, then our algorithm as well as the approach from [6] would no longer work. This is because we would need to go back and check that the smaller square subblocks satisfy that the sum is at most k which would lead us to consider more potential matches and increase the runtime. As a result, we have the following open problem.

▶ **Question 6.** *Can we solve `Max-Bounded-Sums` over integer value matrices in $O(mn)$ time?*

5 Conclusion

In this work, we investigated the runtime complexity of coding challenge problems that relate to finding maximum and minimum size rectangular and square subblocks within matrices. We introduced new algorithms and refinements to existing algorithms to improve upon the previously known runtime bounds. In particular, we demonstrated in Theorem 1 and Corollary 2 that finding a maximum or minimum size subblock with only positive border entries is solvable in $\tilde{O}(mn)$ time. Furthermore, in Theorem 3, we demonstrated that we can find all subblocks with only positive border entries and 0's as interior entries in $O(mn)$ time

and $O(n)$ space. Finally, in Theorem 4 and Corollary 5, we demonstrated that we can find the maximum (minimum) side length such that all squares with this side length have sum upper (lower) bounded by a given value in $O(mn)$ time.

As future directions, we first mention a related problem of computing the rectangle covering number for a Boolean matrix. In particular, the objective is to compute the minimum number of subblocks containing only 1's that are needed to cover all 1 entries of a Boolean matrix [17]. Furthermore, we consider the following open problem.

▶ **Question 7.** *Can the rectangle covering number be computed efficiently?*

This question is of particular interest to us because it is a seemingly different yet related query on the set of subblocks that match the simple pattern restricting all entries to be 1's.

Second, pattern matching within matrices appears relevant to the study of two-dimensional data transformation tasks. Data transformation tasks often consist of finding matches to a pattern and then replacing those matches based on a specification. A recent example of this comes from the ARC Prize Competition where participants compete against the ARC-AGI Benchmark [10]. By investigating certain examples from their benchmark, it is our suggestion that some of these tasks are closely related to ensuring that subblocks within matrices match simple border and interior patterns.

Finally, another future direction of this work is to expand our techniques to the study of pattern matching in higher dimensions to find maximum and minimum size boxes or hyperrectangles.

References

1 221. Maximal Square. LeetCode. URL: https://leetcode.com/problems/maximal-square/.

2 85. Maximal Rectangle. LeetCode. URL: https://leetcode.com/problems/maximal-rectangle/.

3 AlgoArt.org. Accessed: 9-26-2024. URL: https://algoart.org/gallery.html?algid=alg11.

4 Finding maximum rectangular frame in array of zeros and ones. Computer Science Stack Exchange. URL: https://cs.stackexchange.com/q/77983.

5 The 12th Open International Student Olympiad in Programming named after S.A. Lebedev and V.M. Glushkov (KPI-OPEN 2017). Accessed: 5-6-2021. URL: https://web.archive.org/web/20210506134014/http://kpi-open.kpi.ua/problemset/kpi_open_2017_2/eng.pdf.

6 Maximum size of square such that all submatrices of that size have sum less than k. GeeksforGeeks, 2022. Accessed: 2-11-2022. URL: https://www.geeksforgeeks.org/maximum-size-of-square-such-that-all-submatrices-of-that-size-have-sum-less-than-k.

7 Maximum area rectangle of 1s in a binary matrix. GeeksforGeeks, 2025. Accessed: 1-7-2026. URL: https://www.geeksforgeeks.org/dsa/maximum-size-rectangle-binary-sub-matrix-1s/.

8 Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980. doi:10.1145/1008861.1008865.

9 Heinz Breu. An efficient algorithm for finding all maximal square blocks in a matrix. In *Algorithms and Data Structures: Workshop WADS'89 Ottawa, Canada, August 17–19, 1989 Proceedings 1*, pages 460–471. Springer, 1989. doi:10.1007/3-540-51542-9_38.

10 Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc Prize 2024: Technical Report. *arXiv preprint arXiv:2412.04604*, 2024. doi:10.48550/arXiv.2412.04604.

11 Xingyu Dong and Michael Wehar. Enumerating rectangles with empty interior. GitHub Repository. URL: https://github.com/MichaelWehar/Enumerating-Rectangles-With-Empty-Interior.

12 Loe Feijs. Analyzing the structure of mondrian's 1920-1940 compositions, 2020. arXiv:2011.00843.

13 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

14 Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., 1990.

15 František Mráz, Daniel Průša, and Michael Wehar. Two-dimensional pattern matching against local and regular-like picture languages. *Theoretical Computer Science*, 870:137–152, 2021. doi:10.1016/J.TCS.2020.12.026.

16 Nathan Pinsker. USACO 2016 January Contest, Platinum: Problem 1. Fort Moo. Accessed: 1-7-2026. URL: https://usaco.org/index.php?page=viewproblem2&cpid=600.

17 Mozhgan Pourmoradnasseri and Dirk Oliver Theis. The rectangle covering number of random boolean matrices. *Electron. J. Comb.*, 24(2):2, 2017. doi:10.37236/5576.

18 Daniel Průša and Michael Wehar. Complexity of searching for 2 by 2 submatrices in boolean matrices. In *International Conference on Developments in Language Theory*, pages 266–279. Springer, 2020.

19 Kevin Sheng and Ryan Chou. USACO Platinum 2016 January - Fort Moo (Solution). Accessed: 1-7-2026. URL: https://usaco.guide/problems/usaco-600-fort-moo/solution.

20 Gift Siromoney, Rani Siromoney, and Kamala Krithivasan. Abstract families of matrices and picture languages. *Computer Graphics and Image Processing*, 1(3):284–307, 1972. doi:10.1016/S0146-664X(72)80019-4.

21 Xing Sun and Andrew Nobel. On the size and recovery of submatrices of ones in a random binary matrix. *Journal of Machine Learning Research-JMLR*, 9(Nov):2431–2453, 2008.

22 Marc J Van Kreveld and Mark T De Berg. Finding squares and rectangles in sets of points. *BIT Numerical Mathematics*, 31(2):202–219, 1991. doi:10.1007/BF01931281.

23 Hao Yuan and Mikhail J Atallah. Data structures for range minimum queries in multidimensional arrays. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 150–160. SIAM, 2010. doi:10.1137/1.9781611973075.14.