Hexagonal Run-Length Zero Capacity Region—Part II: Automated Proofs

Spencer Congero, *Student Member, IEEE*, and Kenneth Zeger

***Abstract*— The zero capacity region for hexagonal (d, k) run-length constraints is known for many, but not all, d and k. The pairs (d, k) for which it has been unproven whether the capacity is zero or positive consist of: (i) $k = d + 2$ when $d \geq 2$; (ii) $k = d + 3$ when $d \geq 1$; (iii) $k = d + 4$ when either $d = 4$ or d is odd and $d \geq 3$; and (iv) $k = d + 5$ when $d = 4$. Here, we prove the capacity is zero in case (i) when $2 \leq d \leq 9$, in case (ii) when $3 \leq d \leq 11$, and in case (iii) when $d \in \{4, 5, 7, 9\}$. We also prove the capacity is positive in case (ii) when $d \in \{1, 2\}$, in case (iii) when $d = 3$, and in case (iv). The zero capacities for $k = d + 4$ are the first and only known cases equal to zero when $k - d > 3$. All of our results are obtained by developing three algorithms that automatically and rigorously assist in proving either the zero or positive capacity results by efficiently searching large numbers of configurations. The proofs involve either upper bounding the number of paths through certain large directed graphs, finding forbidden strings, or building distinct tileable square labelings. Some of the proofs examine over 20 billion cases using a supercomputer. In Part I of this two-part series, it is proven that the capacity is zero for all of case (i), and for case (ii) whenever $d \geq 7$. Thus, the only remaining unknown cases are now when $k = d + 4$, for any odd $d \geq 11$.**

***Index Terms*— Channel capacity, run length coding, hexagonal constraint.**

I. Introduction

THIS paper is Part II of a two-part series. Some of the background, motivation, and basic definitions will be repeated here so that the presentation can be followed in a self-contained manner. The reader is referred to Part I [7] for other details.

A one-dimensional run-length constraint imposes both lower and upper bounds on the number of zeros that occur between consecutive ones in a binary string. Specifically, if d and k are nonnegative integers, or ∞, then a binary string is said to satisfy a (d, k) constraint if every consecutive pair of ones in the string has at least d zeros between them and the string never has more than k zeros in a row. We will assume throughout that $k < \infty$. It is known that if $k > d$, then the number of (one-dimensional) N-bit binary strings that satisfy the (d, k) constraint grows exponentially in N (e.g., [9]) and that the logarithm (base two) of that number, divided by N, approaches a positive limit as N grows to infinity. This limit is known as the "capacity" of the constraint.

The concepts of (d, k) constraints and capacities have been generalized to two dimensions, where the one-dimensional (d, k) constraint is imposed both vertically and horizontally. Sometimes these two-dimensional constraints are referred to as "rectangular constraints". To determine the capacity of a rectangular constraint, one counts the number of binary labelings of an $N \times N$ square that satisfy the constraint, takes its logarithm, and then divides by the area N^2 of the square. It is known that this quantity approaches a limit $C_{\text{rect}}(d, k)$ (called the "capacity" again) as N grows to infinity (e.g., [12]).

The *zero capacity region* for a particular type of constraint is the set of all pairs (d, k) for which the (d, k) capacity equals zero. If a particular constraint has zero capacity, then the number of valid labelings of a region does not grow exponentially fast in terms of the volume (e.g., length for 1 dimension, area for 2 dimensions, etc.) of the region.

Various estimates of the two-dimensional rectangular capacity were obtained for the particular case $C_{\text{rect}}(1, \infty) \approx 0.587891162$ by Calkin and Wilf [5], Weeks and Blahut [23], Baxter [3], Pavlov [18], and in [16]. This rectangular $(1, \infty)$ constraint is sometimes referred to as the "hard square model" by physicists [2], and its capacity is known to equal the rectangular capacity $C_{\text{rect}}(0, 1)$.

For two-dimensional rectangular (d, k) constraints, the zero capacity region was completely characterized in [12], where it was shown that the capacity satisfies $C_{\text{rect}}(d, k) > 0$ if and only if $k \geq d+2$, when $d \geq 1$ (i.e., $C_{\text{rect}}(d, k) = 0$ when $k = d+1$). It is also known that $C_{\text{rect}}(0, k) > 0$ and $C_{\text{rect}}(k, \infty) > 0$ for all $k \geq 1$. Bounds on the two-dimensional rectangular (d, k) capacity were given in [12], by Sharov and Roth in [20], and were later improved and generalized to higher dimensions by Schwartz and Vardy in [19].

A two-dimensional "hexagonal" constraint imposes one-dimensional (d, k) constraints along the 3 primary directions on a hexagonal lattice. An equivalent way to view the hexagonal constraint on a rectangular lattice is to impose the (d, k) constraint both horizontally and vertically, and also along one of the two diagonal directions (we will use the northeast-southwest direction, but refer to it as the "northeast diagonal") [2, p. 409].

The hexagonal (d, k) capacity, denoted by $C_{\text{hex}}(d, k)$, is the limit as $N \to \infty$ of the logarithm base two of the number of $N \times N$ squares satisfying the hexagonal (d, k) constraint, divided by the area N^2 of the square.

Manuscript received September 21, 2020; revised August 22, 2021; accepted October 10, 2021. Date of publication October 14, 2021; date of current version December 23, 2021.

The authors are with the Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, CA 92093 USA (e-mail: scongero@ucsd.edu; ken@zeger.us).

Communicated by M. Schwartz, Associate Editor for Coding Techniques.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TIT.2021.3120088.

Digital Object Identifier 10.1109/TIT.2021.3120088

The hexagonal (d, k) capacity $C_{\text{hex}}(d, k)$ is known to be positive for certain pairs (d, k). In fact, if $C_{\text{hex}}(d, k) > 0$, then it immediately follows that $C_{\text{hex}}(d', k') > 0$ whenever either $d' < d$ or $k' > k$ (or both), since the constraints weaken in either instance. Positive lower bounds on the hexagonal (d, k) capacity were previously proven for $d = 0$, and for all values of $d \geq 5$ for sufficiently large k (for example, $k = d + 5$ suffices), and now also for $1 \leq d \leq 4$ with our results in this paper. In what follows, we will summarize, for each $d > 0$, the smallest known k such that $C_{\text{hex}}(d, k) > 0$.

The only exactly known non-zero capacity of a hexagonal (d, k) constraint is for the case $(1, \infty)$, which is known in the physics literature as the "hard hexagon model". As with the rectangular constraint, it is easy to show that the hexagonal $(0, 1)$ and $(1, \infty)$ capacities are the same, by reversing the roles of 0s and 1s. The problem of counting the number of patterns in a bounded area that satisfy the hexagonal $(1, \infty)$ constraint was considered in the context of Ising models in physics, by Onsager [17], and Wannier [22]. An equivalent problem is to find the number of configurations of non-attacking kings on a chessboard with regular hexagonal cells.

Metcalf and Yang [15] conjectured that the capacity of the hexagonal $(1, \infty)$ constraint was $\log_2 e^{1/3} \approx 0.48090$, but this was disproven by Baxter and Tsang [4], who obtained a slightly more accurate estimate. Baxter [1], [2], and Joyce [10], [11] performed numerous intricate calculations, which when combined determine the exact hexagonal $(1, \infty)$ capacity.

Using the technique of finding two distinct tileable squares, Censor and Etzion [6] proved that $C_{\text{hex}}(d, d + 4) > 0$ for all even $d \geq 6$. An immediate consequence is that $C_{\text{hex}}(d, d + 5) > 0$ for all odd $d \geq 5$, since the hexagonal $(d, d + 5)$ constraint is weaker than the $(d + 1, d + 5)$ constraint. In this paper, we present a tiling algorithm that automatically generates distinct tileable square labelings that demonstrate positive hexagonal (d, k) capacities for certain pairs (d, k). In particular, we prove that the capacities $C_{\text{hex}}(1, 4)$, $C_{\text{hex}}(2, 5)$, $C_{\text{hex}}(3, 7)$, and $C_{\text{hex}}(4, 9)$ are all positive.

Also, we note that the positive hexagonal (d, k) capacities obtained in [6] were for the case of $k = d + 4$ when d is even and $d \geq 6$, but the proof technique does not apply to odd $d \geq 5$. In this paper, in contrast to even $d \geq 6$, we show that some of the open cases with $k = d + 4$ when d is odd have zero capacity.

Whether or not $C_{\text{hex}}(d, k)$ is positive or zero has been unproven[1] for the following cases:

(i) $k = d + 2$ when $d \geq 2$
(ii) $k = d + 3$ when $d \geq 1$
(iii) $k = d + 4$ when either $d = 4$, or d is odd and $d \geq 3$
(iv) $k = d + 5$ when $d = 4$.

Among these open cases, we prove in Part I that the hexagonal (d, k) capacity equals zero in all of case (i), and in case (ii) whenever $d \geq 7$. Here, in Part II, we prove that the capacity is zero in case (i) when $2 \leq d \leq 9$ and case (ii) when $3 \leq d \leq 11$ (in Corollary II.11 and Corollary III.4), and in case (iii) when $d \in \{4, 5, 7, 9\}$ (in Corollary II.11), and that the capacity is positive in case (ii) when $d \in \{1, 2\}$, in case (iii) when $d = 3$, and in case (iv) (in Theorem IV.1).

[1]Some of these cases were stated in [13] and [14] and are included here for archival purposes.

1	2
3	4

Fig. 1. Tiling configuration for demonstrating positive capacities with the Rectangle Tiling Algorithm. If each of two different binary labelings A and B of an $M \times N$ rectangle can arbitrarily occupy each of the four positions shown without violating the hexagonal (d, k) constraint, then the constraint has positive capacity.

Table I summarizes the present knowledge of the zero capacity region when d is less than 19 and k is less than 25, including the results we present in Parts I and II of these papers. The results from Part I are shown surrounded by squares and the results from Part II are shown surrounded by circles. We note that four of the results turn out to be produced by both the methods in Part I and Part II, and we denote them in the table being surrounded by both a circle and a square. Proofs of the results in Part I and Part II have not previously appeared in the literature. We note that although we provide here and in Part I the first published proofs of the cases where $k = d + 2$, those satisfying $d \geq 7$ are not listed as new results in the table, since they directly follow from our stronger (but more complex) $k = d + 3$ proof in Part I.

We also note that no positive capacities in the table were previously proven on any of the four rows corresponding to $d = 1, 2, 3, 4$, but our results in Theorem IV.1 imply that the entire row to the right of each of our newly added "+" entries is filled with positive capacities.

One way to demonstrate that a particular hexagonal (d, k) capacity is positive is to exhibit at least two rectangular labelings of the same dimensions that can validly tile the plane in any configuration. For example, for some fixed d and k, suppose A and B are two different $M \times N$ rectangles filled with 0s and 1s that each satisfy the hexagonal (d, k) constraint, and such that $d < k < M \leq N$. If each of the 16 possible assignments of rectangles A and B to the four possible positions shown in Figure 1 causes the resulting $2N \times 2M$ rectangle to satisfy the (d, k) hexagonal constraint, then arbitrary tilings of the plane by rectangles A and B also satisfy the same constraint. In such a case, since the rectangles A and B each have area MN and the four positions can be chosen freely as either A or B, the capacity is at least $1/(MN)$, which is positive.

We create a Rectangle Tiling Algorithm (discussed in Section IV) using this technique, and show in Theorem IV.1 that $C_{\text{hex}}(1, 4)$, $C_{\text{hex}}(2, 5)$, $C_{\text{hex}}(3, 7)$, and $C_{\text{hex}}(4, 9)$ are all positive. For each of these cases, a square is determined that can take on two distinct labelings which can arbitrarily tile the plane without violating the constraint. These squares are depicted in Figures 16–19, in which a variable x can be set as either 0 or 1 to obtain two distinct tileable squares (note that $\overline{x} = 1 - x$). Similarly, for $d = 0$, while the fact that $C_{\text{hex}}(0, 1) > 0$ follows from elaborate derivations of Baxter [1] and Joyce [10], [11], a much simpler proof is given in Theorem IV.1 using two distinct tileable squares.

TABLE I

SUMMARY OF THE KNOWN ZERO HEXAGONAL (d,k) CAPACITY REGION FOR SMALL d AND k. ZERO AND POSITIVE CAPACITIES ARE DENOTED BY "0" AND "+", RESPECTIVELY. THE CIRCLED SYMBOLS DENOTE OUR CONTRIBUTIONS IN THE PRESENT PAPER (PART II) AND THE ZEROS IN SQUARES ARE FROM OUR PART I PAPER [7], AND THOSE WITH BOTH SQUARES AND CIRCLES OCCURRED IN BOTH PARTS I AND II. THE QUESTION MARKS ARE REMAINING UNSOLVED CASES

$d\backslash k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
1		0	0	0	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
2			0	0	⊡	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
3				0	0	⊡	⊕	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
4					0	0	⊡	⊕	⊕	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
5						0	0	⊡	⊕	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
6							0	0	⊡	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
7								0	0	0	⊡	⊕	+	+	+	+	+	+	+	+	+	+	+	+	+
8									0	0	0	⊡	+	+	+	+	+	+	+	+	+	+	+	+	+
9										0	0	0	⊡	⊕	+	+	+	+	+	+	+	+	+	+	+
10											0	0	0	⊡	+	+	+	+	+	+	+	+	+	+	+
11												0	0	0	⊡	?	+	+	+	+	+	+	+	+	+
12													0	0	0	□	+	+	+	+	+	+	+	+	+
13														0	0	0	□	?	+	+	+	+	+	+	+
14															0	0	0	□	+	+	+	+	+	+	+
15																0	0	0	□	?	+	+	+	+	+
16																	0	0	0	□	+	+	+	+	+
17																		0	0	0	□	?	+	+	+
18																			0	0	0	□	+	+	+

Prior to this present paper, for the case of $k = d+4$, some hexagonal (d,k) capacities were known to be positive but none were known to equal zero. Specifically, it was previously known [1], [10], [11] that $C_{\text{hex}}(0,1) > 0$, and it is shown here (Theorem IV.1) that $C_{\text{hex}}(1,4) > 0$, $C_{\text{hex}}(2,5) > 0$, and $C_{\text{hex}}(3,7) > 0$, from which it follows that $C_{\text{hex}}(d,d+4) > 0$ whenever $0 \le d \le 3$. Additionally, it was proven by Censor and Etzion [6] that $C_{\text{hex}}(d,d+4) > 0$ for all even $d \ge 6$. This knowledge left as open problems whether $C_{\text{hex}}(d,d+4)$ is positive or zero in the $d = 4$ or odd $d \ge 5$ cases.

In our present paper, we demonstrate the first known cases for $k = d+4$ where the hexagonal capacity is zero, specifically whenever $d \in \{4,5,7,9\}$. The fact that the $k = d+4$ cases alternate between zero and positive hexagonal capacities from $d = 5$ to $d = 10$ contrasts the case of $k = d+3$ where we showed (in Part I) that $C_{\text{hex}}(d,d+3) = 0$ for all $d \ge 3$. Also, in the rectangular constraint case, there is no such alternation between zero and positive capacities [12].

Here, in Part II, one of our approaches to proving particular hexagonal (d,k) capacities are zero is to show that the number of valid labelings of an $N \times N$ square grows like $2^{O(N)}$ as $N \to \infty$, whereas one would need $2^{\Omega(N^2)}$ to assure a positive capacity. To accomplish this, we create the Constant Position Algorithm (defined formally in Section II-D), which assists in the proof by showing that a valid hexagonal (d,k) labeling of any $(k+1)$ consecutive rows in an $N \times N$ square allows only a small number of choices for validly labeling the row immediately above the $(k+1)$ consecutive rows. The exact number of such valid labelings of that new row is shown to be bounded, as $N \to \infty$. This in turn implies that there are only $2^{O(N)}$ valid labelings of the square as $N \to \infty$, which yields zero capacity. These facts are established by constructing two directed graphs representing allowable sequences of labelings of a $(k+1)\times(k+1)$ square that slides horizontally or vertically, one row or column at a time, respectively.

In order to upper bound the number of paths through these two directed graphs, we show that it suffices to determine which pairs of paths through the horizontal graph have vertically compatible vertices. To aid this automated search, we show the useful fact that there is no loss of generality in restricting paths to strongly connected components of the graphs (Lemma II.8). The computation yields an upper bound on the number of ways that a valid labeling of a $(k+1) \times N$ horizontal strip can be extended upwards by one row, and, consequently, gives an upper bound on the number of valid labelings of an $N \times N$ square.

It is shown that if a certain constant position property, defined in Section II-D, exists for a particular hexagonal (d,k) constraint, then the upper bound is tight enough to make a positive capacity impossible for that constraint (Theorem II.9).

We use our Constant Position Algorithm to efficiently verify if the constant position property holds for the (d, k) cases of concern. The algorithm reduces the complexity of such a search from what we assess to be "virtually impossible" with today's technology, down to "manageable", using a multi-node supercomputer with extensive memory capabilities. When the algorithm was implemented and executed, over 20 billion separate cases were examined, and as a result we directly observed that the constant position property indeed holds for 11 new cases. Namely, assisted by the Constant Position Algorithm, we prove in Corollary II.11 that $C_{\text{hex}}(d, k) = 0$ whenever

$$(d,k) \in \{(3,6),(4,7),(4,8),(5,8),(5,9),(6,9),\\ (7,10),(7,11),(8,11),(9,12),(9,13)\}.$$

In fact, we also observed that the constant position property holds more generally for all cases of zero hexagonal (d, k) capacity when $d \leq 9$.

In addition to the Constant Position Algorithm and the Rectangle Tiling Algorithm, we also introduce a third automated proof technique, which we call the Forbidden String Algorithm. The Forbidden String Algorithm, discussed in Section III, proves some hexagonal (d, k) capacities are zero by showing that certain binary strings can never occur in large validly labeled rectangles. It uses the fact that $C_{\text{hex}}(d, k) = C_{\text{hex}}(d + 1, k)$ if 10^d1 is forbidden, and $C_{\text{hex}}(d, k) = C_{\text{hex}}(d, k - 1)$ if 10^k1 is forbidden.

The emphasis in this paper will be on the Constant Position Algorithm, as this algorithm takes the most effort to describe, and gives the best results. After that, we will discuss the Forbidden String Algorithm and the Rectangle Tiling Algorithm.

Some preliminary definitions are now provided. A *rectangle* is an $M \times N$ two-dimensional array, where M is the number of rows and N is the number of columns. If $M < N$ (respectively, $M > N$), then the rectangle is called a *horizontal strip* (respectively, a *vertical strip*). A *labeling* of a set is an assignment $l(x) \in \{0, 1\}$ to each element x of the set. A labeling l of a rectangle is said to *satisfy the hexagonal* (d, k) *constraint* (or is *valid*) if along every row, column, and northeast diagonal, the number of 0s between any two 1s is at least d, and no run of 0s is longer than k. A *position* in an $M \times N$ rectangle is a relative location (x, y), indexed by integer-valued Cartesian coordinates. The ith row (respectively, column) of such a rectangle consists of positions with second (respectively, first) coordinate i. For example, the bottom-left element of any rectangle is at position $(1, 1)$ and the top-right (i.e. in row M and column N) element is at position (N, M).

If l is a labeling, then $l(x, y)$ will denote the value of the labeling l at position (x, y).

Let $L(M, N)$ denote the set of valid labelings of an $M \times N$ rectangle. The *capacity* of the hexagonal (d, k) constraint is defined as

$$C_{\text{hex}}(d,k) = \lim_{M,N\to\infty} \frac{\log_2 |L(M,N)|}{MN}$$

and is known to exist for all d and k by subadditivity (e.g., [12]). Note that if $C_{\text{hex}}(d, k) > 0$, then the number of valid labelings is lower bounded as $|L(M, N)| = 2^{\Omega(MN)}$. For the remainder of the paper, it is assumed that d and k are fixed and all logarithms are base 2.

II. Constant Position Algorithm for Proving Zero Hexagonal (d, k) Capacity

The Constant Position Algorithm is an automated computer search that examines as many as billions of cases in order to assist the rigorous proof of certain zero hexagonal (d, k) capacity cases. Our proof relies on a peculiar property, which we call the constant position property and describe in Section II-D, that turns out to be a sufficient condition for the hexagonal (d, k) capacity to be zero (see Theorem II.9).

The algorithm tries to determine if a given hexagonal (d, k) constraint has zero capacity. The main idea is to check how many valid labelings there are of a horizontal $(k + 1) \times N$ strip, given the (valid) labeling of the horizontal $(k + 1) \times N$ strip immediately below it (i.e., shifted one row down). If the number of such labelings of the upper horizontal strip is small enough, no matter which labeling is used for the lower horizontal strip, then the number of valid labelings of an $N \times N$ square can be upper bounded tightly enough to avoid $2^{\Omega(N^2)}$ growth, thus proving the (d, k) constraint has zero capacity.

In order to efficiently determine if the labelings of such shifted horizontal strips are severely constrained, it is convenient to create a directed graph, whose nodes are valid labelings of a $(k + 1) \times (k + 1)$ square, and whose directed edges convey whether one square labeling can overlay the other, but shifted one column to the right. We also create an analogous graph for vertical sliding compatibility. We examine labelings corresponding to walks through the constructed (horizontal) graph that do not change from one graph component to another. We prove that this is, in fact, possible to do without sacrificing generality (see Lemma II.8). As a result, the Constant Position Algorithm below focuses on horizontal $(k + 1) \times N$ labelings that each correspond to a walk in a single arbitrary component of the (horizontal) graph.

We note that as k grows, the number of valid labelings of $(k+1) \times (k+1)$ squares grows rapidly, which makes storage and processing computationally difficult. Thus, the computational complexity of performing the Constant Position Algorithm increases, and at some point becomes infeasible, even with massive computing power (typically when $k \geq 11$). However, for every computationally feasible case, our results establish that $C_{\text{hex}}(d, k) = 0$ if and only if the constant position property holds, We suspect the "only if" direction of this assertion may be true for even more complex cases, but we leave it as an open question in Conjecture II.12.

A. Definitions

Let Λ be the set of valid labelings of a $(k + 1) \times (k + 1)$ square. Define two directed graphs

$$G_h = (\Lambda, E_h)$$
$$G_v = (\Lambda, E_v)$$

such that $(x, y) \in E_h$ if and only if the x-labeling of the rightmost k columns of the $(k+1) \times (k+1)$ square is identical

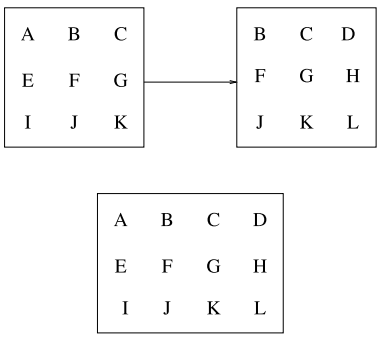


Fig. 2. Illustration of an edge in G_h. Two 3×3 labeled squares (the vertices) are overlaid to form a 3×4 horizontal strip. The letters represent binary values according to some valid labeling.

to the y-labeling of the leftmost k columns of the square, and such that $(x, y) \in E_v$ if and only if the x-labeling of the topmost k rows of the square is identical to the y-labeling of the bottommost k rows of the square. Any such graphs will be called *label graphs*.

Figure 2 illustrates labelings of two 3×3 squares and how they can form the vertices of a single edge in G_h. Thus if there is an edge from x to y in G_h, then the valid labeling x can be extended rightward by one column to a valid labeling of a $(k + 1) \times (k + 2)$ rectangle, using the y-labeling of the rightmost column of the $(k+1)\times(k+1)$ rectangle. Similarly, if there is an edge from x to y in G_v, then the valid x-labeling can be extended upward one row using the y-labeling of the topmost row of the $(k + 1) \times (k + 1)$ rectangle. In particular, if (λ_1, λ_2) is an edge in G_v, then labeling $\lambda_2 \in \Lambda$ is said to be *vertically compatible above* labeling $\lambda_1 \in \Lambda$.

The label graph G_h is defined so that a labeling of a $(k+1)\times N$ horizontal strip is valid if and only if the sequence of labelings of $(k+1)\times(k+1)$ squares, obtained by sliding one column at a time from left to right in the strip, is a (directed) walk in the graph G_h. Similarly, the label graph G_v is defined so that a labeling of a $M \times (k + 1)$ vertical strip is valid if and only if the sequence of labelings of squares, obtained by sliding one row at a time from bottom to top in the strip, is a (directed) walk in the graph G_v. In these cases, the walk in G_h is said to *correspond to* the labeling of the horizontal strip, and the walk in G_v is said to *correspond to* the labeling of the vertical strip.

Define $L_{G_1,G_2}(M, N)$ to be the set of labelings l of an $M\times N$ rectangle such that the restriction of l to any $(k+1)\times N$ horizontal strip corresponds to a walk through G_1 and the restriction of l to any $M \times (k + 1)$ vertical strip corresponds to a walk through G_2. The two label graphs G_1 and G_2 are said to *generate* $L_{G_1,G_2}(M, N)$.

One basic fact that we will repeatedly rely on is that any labeling of an $M \times N$ square is valid if and only if the labeling is valid on every $(k + 1) \times (k + 1)$ subsquare of the $M \times N$ square. This is due to the fact that any violation of the k constraint (i.e., the existence of a string 0^{k+1}) along a horizontal, vertical, or diagonal would have to occur in a $(k + 1) \times (k + 1)$ subsquare (as would any violation of the d contraint). A consequence (shown in Lemma II.5) is that $L_{G_h,G_v}(M, N) = L(M, N)$.

A directed graph is called *strongly connected* if there exist directed paths from every vertex to every other vertex in the graph. A strongly connected subgraph K of G is *maximal* if no other strongly connected subgraph of G contains K. Maximal strongly connected subgraphs of G will be referred to as *components*. The components of G partition the vertices of G, but not necessarily the edges. A key property that we will make use of is that any walk through a directed graph that leaves a particular component can never return to that component.

A component is called *cyclic* if it contains a directed cycle. In G_h, any component K is cyclic if and only if a bi-infinite walk through K corresponds to a labeling of a $(k + 1) \times \infty$ horizontal strip. In fact, the only non-cyclic components are single vertices without self-loops, and thus the lone vertex in a non-cyclic component can appear in the labeling of a $(k+1)\times N$ horizontal strip at most once. Analogous statements apply to cyclic and non-cyclic components in G_v.

If $d > 0$ and $k < \infty$, then the graphs G_h and G_v cannot contain any self-loops, since the vertex in any self-loop would necessarily contain either an all-0 or all-1 labeling of a row or column of a $(k+1)\times(k+1)$ square. More generally, all cycles in G_h and G_v must have at least $d + 1$ edges, and thus all cyclic components of G_h and G_v must contain at least $d + 1$ vertices.

The following example illustrates the types of components that can occur in G_h or G_v.

Example II.1: The directed graph in Figure 3 has components whose vertex sets are $\{1, 2, 3\}$, $\{4\}$, $\{5, 6\}$, and $\{7, 8, 9\}$, and whose edge sets consist of all edges that connect vertices within each vertex set. Of these components, all are cyclic except the component whose vertex set is $\{4\}$.

In the following example, we use the hexagonal $(3, 4)$ constraint to illustrate the correspondence between walks through the graph G_h and sequences of labelings of $(k+1)\times(k + 1)$ squares in $(k + 1) \times M$ horizontal strips for an actual valid example labeling. The capacity of the hexagonal $(3, 4)$ constraint is known to be zero (it is implied by the known zero capacity of the rectangular $(3, 4)$ constraint), and it results in a relatively small label graph G_h.

Example II.2: Figure 4 illustrates the directed graph G_h for the hexagonal $(3, 4)$ constraint. The graph G_h consists of 7 components: three disjoint 5-cycles, and four isolated vertices. Each vertex is a labeling of a 5×5 square. Figure 5 shows an example of a valid labeling of a 15×20 rectangle. It can be seen that the labeling of any horizontal strip of width 5 corresponds to a walk through one of the 3 cycles in the graph.

Let R be the set of all $(k + 1) \times (k + 1)$ squares in an $M \times N$ rectangle. Each square in R lies entirely in one particular $(k + 1) \times N$ horizontal strip within the $M \times N$ rectangle. Each such horizontal strip contains $(N-k)$ squares of size $(k+1)\times(k+1)$, and for any labeling of the $M \times N$ rectangle, the labeling of each such $(k + 1) \times (k + 1)$ square within the strip belongs to exactly one component of G_h. In fact, when scanning a strip from left to right, if the labeling of a $(k + 1) \times (k + 1)$ square lies in a particular component, but the labeling of the next square to its right (i.e., sliding one column rightward) in the strip does not lie in the same

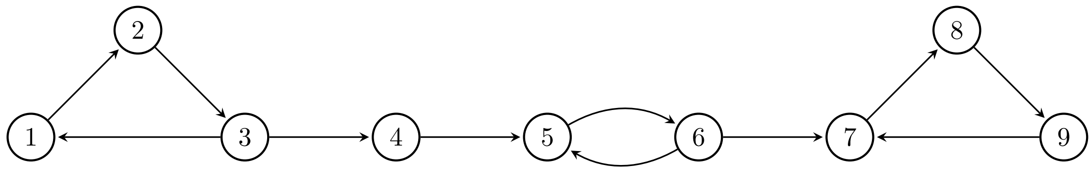


Fig. 3. Graph used in example II.1.

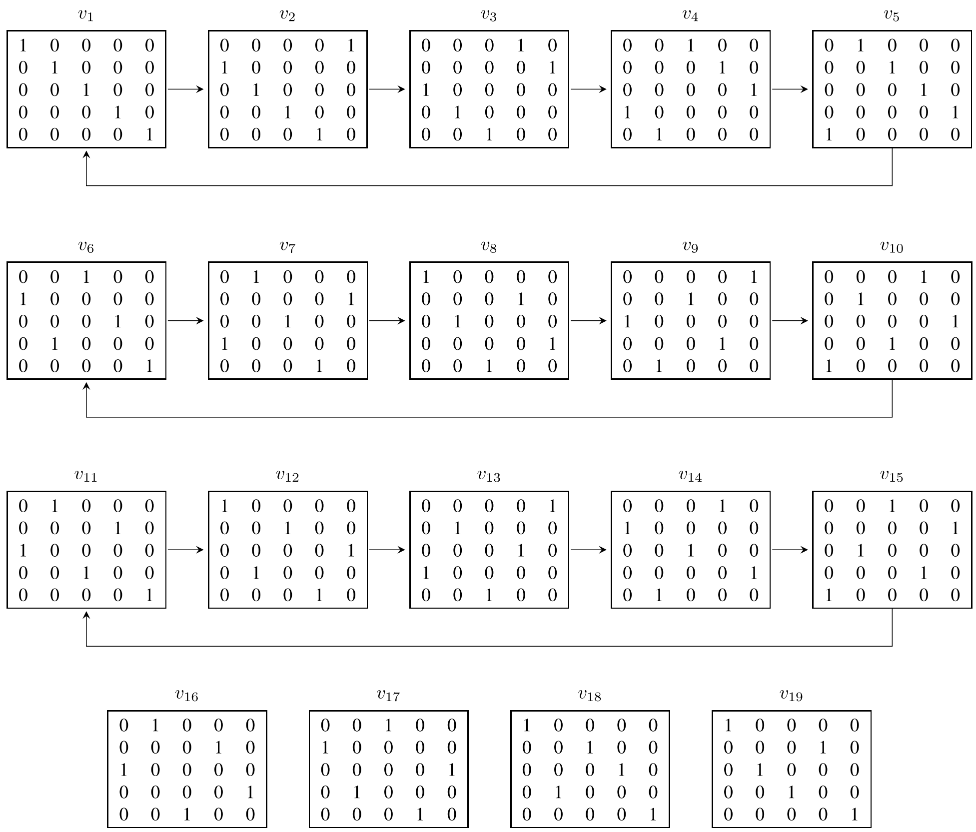


Fig. 4. The graph G_h for the hexagonal $(3, 4)$ constraint. The graph G_h consists of 3 disjoint 5-cycles and 4 isolated vertices, where each vertex is a valid labeling of a 5×5 square.

0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1

Fig. 5. A labeling of a 15×20 rectangle satisfying the hexagonal $(3, 4)$ constraint. The labeling of each 5×20 horizontal strip in the rectangle corresponds to a walk through the second component of G_h listed in Figure 4. For example, the labeling of the horizontal strip consisting of the top 5 rows of the rectangle corresponds to the sequence of 16 vertices that starts at v_6 and continues through the cycle in the second component of G_h, i.e., $v_6, v_7, v_8, v_9, v_{10}, v_6, \ldots$.

component, then the former component can never be revisited within that strip.

Given a fixed valid labeling l of an $M \times N$ rectangle, we identify, for each horizontal strip and for each component of G_h, the first (i.e., leftmost) occurrence in the strip of a $(k+1) \times (k+1)$ square labeling from the component. We use the notation $T_h(l)$ to denote the set of all such leftmost squares in all strips of the $M \times N$ rectangle. Specifically, for any

$l \in L_{G_h,G_v}(M,N)$, define

$$T_h(l) = \{r \in R : \text{the labeling of each } (k+1)\times(k+1) \text{ square to the left of } r \text{ in the same strip belongs to a different component of } G_h \text{ as the labeling of } r\}.$$

Note that every $(k+1)\times(k+1)$ square whose labeling under l is in a non-cyclic component of G_h necessarily lies in $T_h(l)$.

For any fixed labeling $l \in L_{G_h,G_v}(M,N)$, each $(k+1)\times N$ horizontal strip in the $M \times N$ rectangle corresponds to a walk through the graph G_h. Each node in such a walk belongs to exactly one component of G_h. If (a,b) is a directed edge in such a walk and a and b belong to different components of G_h, then the rectangle being labeled by b is in $T_h(l)$. Thus, $T_h(l)$ is the set of all locations of component transitions in G_h for the horizontal strips of the $M \times N$ rectangle.

We define the equivalence relation $\sim$ between labelings $l_1, l_2 \in L_{G_h,G_v}(M,N)$ so that $l_1 \sim l_2$ iff $T_h(l_1) = T_h(l_2)$, and denote the associated equivalence class of any $l \in L_{G_h,G_v}(M,N)$ by $[l]$.

The Constant Position Algorithm is described in Section II-D in terms of 6 Steps, and in this section we offer a preview in order to convey the gist of how it works. The following definition will be used throughout the remainder of the paper.

Definition II.3: For a given hexagonal (d,k) constraint, let $H_1,\ldots,H_\alpha$ and $V_1,\ldots,V_\beta$, respectively, denote the components of the directed graphs G_h and G_v. Also, define the disjoint graph unions

$$H = \bigcup_{i=1}^{\alpha} H_i \quad \text{and} \quad V = \bigcup_{i=1}^{\beta} V_i.$$

The graphs H and V are obtained from the graphs G_h and G_v, respectively, by removing all edges between different components. For some pairs (d,k), it may happen that none of the components of G_h (respectively, G_v) are connected by edges to any other components (e.g., see Example II.2), in which case $H = G_h$ (respectively, $V = G_v$).

B. Preview of Step 5 of the Algorithm

An explicit description of the Constant Position Algorithm is given in Section II-D. Here, we motivate Step 5, a key part of the algorithm.

Consider a valid labeling of a $(k+2)\times(k+1)$ vertical rectangle within an $N \times N$ square. Denote the labelings of the lower and upper $(k+1)\times(k+1)$ squares within this vertical rectangle by λ and λ', respectively, and let λ and λ' be nodes common to both H and V. Note that λ' is vertically compatible above λ, i.e., the ordered pair (λ,λ') is a directed edge in V. Since the lower and upper squares overlap in k rows, the number of possibilities for the pair (λ,λ') is limited.

Denote by ρ the top row of the $(k+2)\times(k+1)$ rectangle. For any given labeling λ, all possible labelings λ' that are vertically compatible above λ may agree with each other in certain locations of ρ. By observing numerous such labelings for the (d,k) pairs of interest, we discovered that for every fixed pair of components that λ and λ' could belong to in H, there always appears to be at least one position in ρ that gets labeled the same by every λ' that is vertically compatible above a given λ.

In other words, whereas the noted position in ρ depends only on the fixed pair of components of λ and λ', the value of the labeling at that position depends additionally on the specific choice of λ within its component. However, the value is constant for each λ' in its component that is vertically compatible above λ. The existence of such a position seems to arise due to the smallness of the difference $(k-d)$ in our cases of interest, and provides a crucial step in our proof that certain hexagonal capacities are zero.

C. Preview of Lemma II.8

Lemma II.8 provides an important reduction in the amount of computational complexity needed to determine whether the constant position property holds for the particular hexagonal (d,k) constraint being examined. Specifically, it allows one to restrict attention to searching the connected components of two directed graphs rather than the entire graphs. We now provide a preview and summary of the proof of this lemma.

Let $l \in L_{G_h,G_v}(N,N)$. Then the labeling of each $(k+1)\times N$ horizontal strip induced by l corresponds to a walk through G_h containing exactly $(N-k)$ nodes, and there are at most α vertices whose previous vertex in the walk is from a different component (since there are α components in G_h). These at most α vertices in a component are the first occurrences in the walk of a vertex from the component they lie in, and are called "transition vertices". Similarly, there are exactly $(N-k)$ horizontal strips of size $(k+1)\times N$ in the $N \times N$ square, so there are at most $\alpha(N-k) \le \alpha N$ transition vertices among all of the $(N-k)$ horizontal strips. We call the squares labeled by the transition vertices "transition squares".

In Lemma II.7, we show that there are not too many possible arrangements of these transition squares in the $N \times N$ square. More specifically, in each $(k+1)\times N$ horizontal strip, any of the at most α transition squares can be appear in at most $(N-k)$ locations or not at all. Thus there are at most $(N-k+1)^\alpha$ possible arrangements of the transition squares in such a strip. Since there are $(N-k)$ horizontal strips, it follows that there are at most

$$(N-k+1)^{\alpha(N-k)} \le 2^{\alpha N \log N} = 2^{o(N^2)}$$

possible arrangements of the transition squares in the entire $N \times N$ square.

Now suppose the number of valid labelings of an $N \times N$ square is large enough (asymptotically) to yield a positive capacity, i.e., $|L_{G_h,G_v}(N,N)| = 2^{\Omega(N^2)}$. Then, by the previous argument, there exists a particular arrangement of these transition squares for which there are many labelings in $L_{G_h,G_v}(N,N)$ (specifically $2^{\Omega(N^2)}$) that induce this exact arrangement of transition squares in the $N \times N$ square. We show in Lemma II.8 that for such a fixed particular arrangement, there exists a smaller subsquare of the $N \times N$ square that:

(i) is disjoint from the transition squares, and
(ii) has many valid labelings.

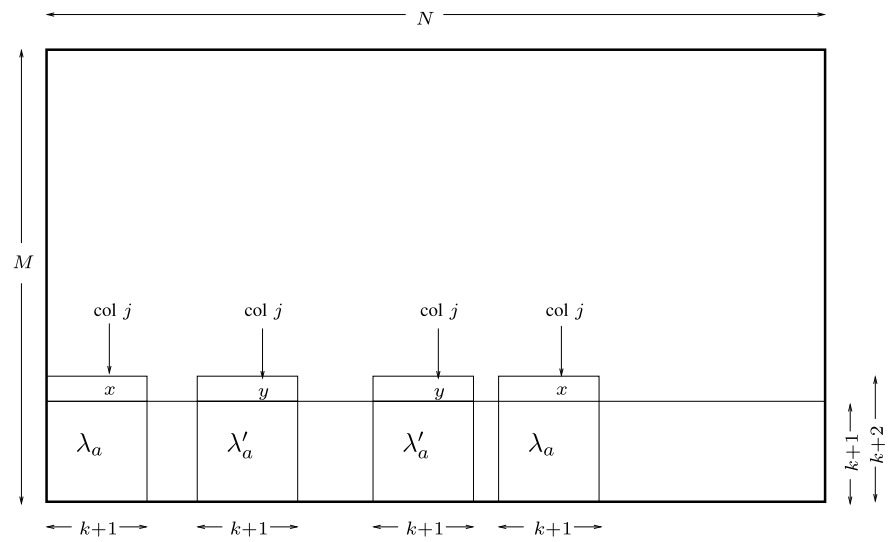


Fig. 6. Illustration of the constant position property.

Specifically, this subsquare has side length of order $\sqrt{N}$, and has $2^{\Omega(N)}$ valid labelings.

To find such a subsquare, we partition the $N \times N$ square into a grid of about N/σ^2 subsquares with side lengths approximately $\sigma\sqrt{N}$, and we can (and do) choose σ so that the percentage of these subsquares that intersect transition squares stays arbitrarily small as N grows.

The subsquares that intersect transition squares have a total area equal to the number of subsquares (approximately $(k+1)^2\alpha N$) times the subsquare area (approximately $\sigma^2 N$). Thus, the number of valid labelings of all the subsquares that intersect transition squares is at most $2^{(k+1)^2\alpha\sigma^2 N^2}$, whose growth rate as $N \to \infty$ can be made arbitrarily less than that of the capacity growth rate, $2^{C_{\text{hex}}(d,k)N^2}$, by choosing σ small enough. This fact implies that the growth rate of the number of valid labelings of all the subsquares that do *not* intersect transition squares can be made asymptotically arbitrarily close to $2^{C_{\text{hex}}(d,k)N^2}$. Since there are only N/σ^2 subsquares in total, we can show that at least one subsquare that does not intersect any transition square must have about $2^{C_{\text{hex}}(d,k)N}$ valid labelings.

Such a subsquare satisfying (i) and (ii) above can be found for any side length $\sqrt{N}$ by considering larger and larger $N \times N$ squares, and so $|L_{H,G_v}(\sqrt{N},\sqrt{N})|$ is asymptotically $2^{C_{\text{hex}}(d,k)N}$, and thus $|L_{H,G_v}(N,N)|$ is asymptotically $2^{C_{\text{hex}}(d,k)N^2}$. Applying the same argument to transitions between components of G_v that occur in labelings of $L_{H,G_v}(N,N)$ shows that $|L_{H,V}(N,N)|$ is also asymptotically $2^{C_{\text{hex}}(d,k)N^2}$. In other words, in Lemma II.8 we show the capacity of the hexagonal (d,k) constraint is unchanged even if we constrain all walks through G_h and G_v, corresponding to labelings of $(k+1) \times N$ horizontal and $N \times (k+1)$ vertical strips, to stay within a single component each. This theorem justifies restricting attention to the components of G_h and G_v in Steps 3, 4, and 5 of the Constant Position Algorithm.

D. Algorithm Description

We say that an ordered pair of components (H_a, H_b) of G_h is *vertically semi-compatible* if at least one vertex in H_b is vertically compatible above at least one vertex in H_a.

Definition II.4: A hexagonal (d,k) constraint has the *constant position property* if for every pair (H_a, H_b) of vertically semi-compatible components from G_h, there exists $j \in \{1,\ldots,k+1\}$, such that for each labeling λ_a in H_a, the value at position $(j,k+1)$ of every labeling λ_b in H_b vertically compatible above λ_a is constant (note: the value can vary with λ_a).

The constant position property is the key observation that allows us to successfully find new hexagonal (d,k) capacities that equal zero. This property is illustrated in Figure 6.

It is assumed that the bottom-most $(k+1) \times N$ horizontal strip of the $M \times N$ rectangle contains $(k+1) \times (k+1)$ square labelings from the component H_a in G_h, and that the overlapping $(k+1) \times N$ horizontal strip one row higher contains $(k+1)\times(k+1)$ square labelings from the component H_b in G_h. The four $(k+1) \times (k+1)$ squares shown in the bottom strip of Figure 6 are assumed to have labelings λ_a and λ'_a, as indicated. Any location denoted by x or y is at position $(j,k+1)$ of the $(k+1)\times(k+1)$ squares that are shifted one row up from the squares labeled by λ_a or λ'_a. If the constraint has the constant position property, then the values at each such x are the same no matter which $(k+1) \times (k+1)$ labeling from H_b is chosen, and similarly for the values of each such y, although the value of y may differ from the value of x.

The Constant Position Algorithm is not guaranteed to find all such zero capacity constraints, but has proven effective for all cases within its computational complexity capabilities (see Theorem II.13).

Constant Position Algorithm

- **Step 1:** Create the set Λ of valid labelings of a $(k+1)\times(k+1)$ square.

- **Step 2:** Create label graphs $G_h = (\Lambda, E_h)$ and $G_v = (\Lambda, E_v)$.
- **Step 3:** Determine the components of G_h and G_v, i.e., $H_1, \ldots, H_\alpha$ and $V_1, \ldots, V_\beta$.
- **Step 4:** Identify all pairs (H_a, H_b) of vertically semi-compatible components.
- **Step 5:** Determine if the hexagonal (d, k) constraint has the constant position property.
- **Step 6:** If the determination in Step 5 is true, then output "success"; otherwise, output "failure".

E. Lemmas for Zero Capacity Proof

In this section, four lemmas are given that lead to the main technical result, Theorem II.9 in Section II-F, which asserts that the hexagonal (d, k) capacity is zero whenever the constant position property occurs.

Lemma II.5: $L_{G_h,G_v}(M, N) = L(M, N)$.

Proof: Suppose $l \in L(M, N)$, i.e., l is a valid labeling of an $M \times N$ rectangle. For any $(k+1) \times N$ horizontal strip, the sequence of labelings of $(k + 1) \times (k + 1)$ squares, sliding left to right one column at a time within the strip, forms a walk through G_h, since these $(k + 1) \times (k + 1)$ square labelings are necessarily valid and each successive pair of labelings corresponds to an edge in G_h by the construction of G_h. Similarly, labelings of vertical strips correspond to walks in G_v.

Conversely, if $l \notin L(M, N)$, then either the d constraint or the k constraint is violated somewhere in the $M \times N$ rectangle, which implies that the labeling of some $(k + 1) \times (k + 1)$ subsquare is not valid. Therefore the labelings of the horizontal and vertical strips containing this subsquare do not correspond to walks through G_h and G_v, respectively. Thus $l \notin L_{G_h,G_v}(M, N)$. ∎

The label graphs G_h and G_v contain components, some of which may be reachable from other components. Any edge that connects a vertex from one component to another component cannot itself lie in any component. Thus, a label graph consists of a disjoint union of components together with connecting edges not belonging to any component. If we remove all such connecting edges of the label graphs G_h and G_v, then we are left with the disjoint unions of components of G_h and G_v. For any fixed $M \times N$ rectangle, those disjoint unions of components generate a subset of the labelings of the rectangle that are valid under the hexagonal (d, k) constraint (i.e., $L_{H,V}(M, N) \subseteq L(M, N)$). Thus, using the number of such valid labelings generated by the reduced label graphs to estimate the hexagonal (d, k) capacity will yield a number less than or equal to $C_{\text{hex}}(d, k)$ (i.e., $|L_{H,V}(M, N)| \leq |L(M, N)|$). Lemma II.8 below however, shows that, in fact, the capacity $C_{\text{hex}}(d, k)$ is still obtained with equality by only counting the number of valid labelings in this reduced case (i.e., $|L_{H,V}(M, N)|$ and $|L(M, N)|$ have the same growth rate as $M, N \to \infty$).

First we provide two technical lemmas to aid in the proof of Lemma II.8. In particular, the following lemma gives: (i) a linear (in the number of rows of an $M \times N$ rectangle) upper bound on the number of component transition locations in $T_h(l)$; and (ii) an upper bound on the number of different transition sets $T_h(l)$ that can occur across all $l \in L_{G_h,G_v}(M, N)$.

Lemma II.6: Suppose label graphs G_h and G_v generate a set of labelings $L_{G_h,G_v}(M, N)$ of an $M \times N$ rectangle. Then the following hold:

(i) $|T_h(l)| \leq \alpha M$ for any $l \in L_{G_h,G_v}(M, N)$,
(ii) $|L_{G_h,G_v}(M, N)/\sim| \leq 2^{\alpha \log(N+1)M}$.

Proof: Let $l \in L_{G_h,G_v}(M, N)$ be a labeling of an $M \times N$ rectangle generated by G_h and G_v. Then, in particular, the labeling of each $(k + 1) \times N$ horizontal strip of l corresponds to a walk through G_h. From the definition of $T_h(l)$, a $(k + 1) \times (k + 1)$ square r can be included in $T_h(l)$ only if the labeling of each $(k + 1) \times (k + 1)$ square to the left of r in the same strip belongs to a component of G_h other than the component containing the labeling of r. Thus there can be at most α (i.e., the number of components in G_h) $(k + 1) \times (k + 1)$ squares of any horizontal strip included in $T_h(l)$, and therefore $|T_h(l)| \leq \alpha M$.

Now for the second part. In a given $(k+1) \times N$ strip there are at most $N+1$ options for the position of a $(k+1) \times (k+1)$ square $r \in T_h(l)$, where the one extra option corresponds to labelings from a component not appearing in the horizontal strip at all. Thus there are at most $(N + 1)^\alpha$ ways to arrange the at most α possible such squares from $T_h(l)$ that can appear in a given horizontal strip. Since the number of $(k + 1) \times N$ horizontal strips is at most M, there are no more than $(N + 1)^{\alpha M}$ ways to position all of the squares in $T_h(l)$. Therefore, $|L_{G_h,G_v}(M, N)/\sim| \leq 2^{\alpha \log(N+1)M}$. ∎

Recall that two labelings of an $M \times N$ rectangle are equivalent under the relation $\sim$ if the labelings transition within horizontal strips from one component of G_h to another at the same locations. The following lemma helps us prove Lemma II.8 by allowing us to restrict attention to one equivalence class of such labelings.

Lemma II.7: If $C_{\text{hex}}(d, k) > 0$, then for any $\epsilon > 0$, there exist constants M_1 and N_1 such that for all $M > M_1$ and $N > N_1$, there exists an equivalence class $[l] \in L_{G_h,G_v}(M, N)/\sim$ whose size is at least $2^{(C_{\text{hex}}(d,k)-\epsilon)MN}$.

Proof: Let $\epsilon > 0$. Since

$$C_{\text{hex}}(d, k) = \lim_{M,N\to\infty} \frac{\log |L_{G_h,G_v}(M, N)|}{MN} > 0$$

we can find constants M_0 and N_0 such that $|L_{G_h,G_v}(M, N)| \geq 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2})MN}$ for all $M > M_0$ and $N > N_0$. By Lemma II.6, $|L_{G_h,G_v}(M, N)/\sim| \leq 2^{\alpha \log(N+1)M}$, where α is the number of components in G_h, and thus for any $M > M_0$ and $N > N_0$,

$$\begin{aligned} 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2})MN} &\leq |L_{G_h,G_v}(M, N)| \\ &= \sum_{[l]\in L_{G_h,G_v}(M,N)/\sim} |[l]| \\ &\leq 2^{\alpha \log(N+1)M} \max_{[l]\in L_{G_h,G_v}(M,N)/\sim} |[l]|. \end{aligned}$$

Therefore,

$$\begin{aligned} \max_{[l]\in L_{G_h,G_v}(M,N)/\sim} |[l]| &\geq 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2})MN-\alpha \log(N+1)M} \\ &= 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2}-\alpha\frac{\log(N+1)}{N})MN}. \end{aligned}$$

Taking N large enough gives $\alpha \cdot \frac{\log(N+1)}{N} \leq \frac{\epsilon}{2}$, proving the lemma. ∎

The following lemma shows that, when enumerating the valid labelings of an $M \times N$ rectangle in order to calculate the hexagonal (d,k) capacity, it suffices to count only those labelings for which the labeling of each $(k+1)\times N$ horizontal strip corresponds to a path in a single component of G_h, and the labeling of each $M \times (k+1)$ vertical strip corresponds to a path in a single component of G_v.

Lemma II.8:

$$C_{\text{hex}}(d,k) = \lim_{M,N\to\infty} \frac{\log |L_{H,V}(M,N)|}{MN}.$$

Proof: First suppose $C_{\text{hex}}(d,k) = 0$. Then since H and V are subgraphs of G_h and G_v, respectively, for all M and N we have $L_{H,V}(M,N) \subseteq L_{G_h,G_v}(M,N)$, and thus $|L_{H,V}(M,N)| \leq |L_{G_h,G_v}(M,N)|$. In this case,

$$\begin{aligned} 0 &\leq \lim_{M,N\to\infty} \frac{\log |L_{H,V}(M,N)|}{MN} \\ &\leq \lim_{M,N\to\infty} \frac{\log |L_{G_h,G_v}(M,N)|}{MN} \\ &= 0. \end{aligned}$$

Thus we will assume $C_{\text{hex}}(d,k) > 0$. Note that the limits on the right and left sides in the statement of the theorem exist by [12]. Let $\epsilon \in (0,2)$. Then by Lemma II.7, there exist M_1, N_1 such that for all $M > M_1$ and all $N > N_1$, there exists an equivalence class $[l] \in L_{G_h,G_v}(M,N)/\sim$ such that $|[l]| \geq 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{3})MN}$.

In what follows, we may assume $M = N > \max\{M_1, N_1\}$. We will use floor functions to ensure the dimensions of the required subsets of an $N \times N$ square are integers. Let σ be a positive real number such that

$$\sigma \leq \sqrt{\frac{\epsilon}{2(k+1)^2\alpha}} \tag{1}$$

(where α is the number of components in G_h) and consider an $f(N) \times f(N)$ square ψ, where $f(N) = \lfloor \sigma\sqrt{N} \rfloor$ (assume N is large enough so that $f(N) \geq 1$). Thus,

$$f(N)^2 \leq \sigma^2 N. \tag{2}$$

Define $s(N) = f(N)\lfloor \sqrt{N}/\sigma \rfloor$ to be the side length of a large square subset, which occupies nearly all of the $N \times N$ square. Such an $s(N)\times s(N)$ square can be tiled in the obvious way by using $\lfloor \sqrt{N}/\sigma \rfloor^2$ disjoint copies of ψ. Let Ψ denote the set of these disjoint subsquares of the $s(N)\times s(N)$ square. Thus,

$$|\Psi| = \left\lfloor \frac{1}{\sigma}\sqrt{N} \right\rfloor^2 \leq N/\sigma^2. \tag{3}$$

Note that the number of positions of the $N \times N$ square that are not covered by any $\psi \in \Psi$ is $N^2 - s(N)^2$.

The number of component transitions that occur among all the horizontal strips in the $N \times N$ square is the size of the transition set $T_h(l)$, whereas the number of $f(N) \times f(N)$ subsquares is the size of Ψ. We will demonstrate that there are more of these subsquares than there are component transitions, and in fact, there is always at least one subsquare which does not overlap any $(k+1) \times (k+1)$ square associated with a component transition in G_h. Any such subsquare in Ψ will be called *safe*, and otherwise, *unsafe*.

Since α is the number of components of G_h, by Lemma II.6, we have $|T_h(l)| \leq \alpha N$, for all $l \in L_{G_h,G_v}(M,N)$. That means there are at most $(k+1)^2\alpha N$ of the N^2 positions in the $N\times N$ square that are contained in some $(k+1) \times (k+1)$ square of $T_h(l)$. Each such position can be located in at most one of the (disjoint) squares in Ψ. Thus, the number of safe squares in Ψ is at least

$$\begin{aligned} &|\Psi| - (k+1)^2\alpha N \qquad (4)\\ &= \left\lfloor \frac{1}{\sigma}\sqrt{N} \right\rfloor^2 - (k+1)^2\alpha N \\ &> \left(\frac{1}{\sigma}\sqrt{N} - 1 \right)^2 - (k+1)^2\alpha N \\ &\geq \left(\sqrt{2(k+1)^2\alpha N/\epsilon} - 1 \right)^2 - (k+1)^2\alpha N \quad \text{[from (1)]} \\ &> 0 \qquad (5) \end{aligned}$$

for sufficiently large N (since $\epsilon < 2$). Therefore, there exists at least one safe square in Ψ.

In the remainder of the proof, we will show that at least one safe square has many valid labelings. Note that there are at most

$$2^{((k+1)^2\alpha N) f(N)^2} \leq 2^{(k+1)^2\alpha\sigma^2 N^2} \qquad \text{[from (2)]} \tag{6}$$

ways to label the unsafe squares in Ψ.

For any subsquare $\psi \in \Psi$, let $[l](\psi)$ denote the set of labelings of the subsquare ψ induced by the $N\times N$ square labelings from the equivalence class $[l]$. The quantity $|[l]|$ is the number of valid labelings (of the $N \times N$ square) which are equivalent under $\sim$ to l. This number of valid labelings can be upper bounded by multiplying: (i) the number of labelings of the slightly smaller $s(N) \times s(N)$ square induced by the labelings in equivalence class $[l]$; and (ii) the number of (possibly non-valid) labelings of the set of $N^2 - s(N)^2$ positions in the $N \times N$ square that lie outside of the $s(N) \times s(N)$ square. Furthermore, for these two quantities in the product, (i) can be upper bounded by multiplying the numbers of labelings of each ψ in Ψ, induced by the labelings in $[l]$; and (ii) can be upper bounded by raising 2 to the number of positions in the $N \times N$ square that lie outside the $s(N)\times s(N)$ square. Thus,

$$\begin{aligned} &2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{3})N^2} \\ &\leq |[l]| \qquad \text{[from Lemma II.7]} \\ &\leq \left(\prod_{\psi\in\Psi} |[l](\psi)| \right) \cdot 2^{N^2 - s(N)^2} \\ &= \left(\prod_{\text{unsafe } \psi\in\Psi} |[l](\psi)| \right) \left(\prod_{\text{safe } \psi\in\Psi} |[l](\psi)| \right) \cdot 2^{N^2 - s(N)^2} \\ &\leq 2^{(k+1)^2\alpha\sigma^2 \ N^2} \cdot \left(\prod_{\text{safe } \psi\in\Psi} |[l](\psi)| \right) \cdot 2^{N^2 - s(N)^2} \\ &\qquad \text{[from (6)]} \end{aligned}$$

$$\leq 2^{(k+1)^2\alpha\sigma^2\ N^2}\cdot\left(\prod_{\text{safe }\psi\in\Psi}\max_{\text{safe }\psi\in\Psi}|[l](\psi)|\right)\cdot 2^{N^2-s(N)^2}$$
$$\leq 2^{(k+1)^2\alpha\sigma^2\ N^2}\cdot\left(\max_{\text{safe }\psi\in\Psi}|[l](\psi)|\right)^{|\Psi|}\cdot 2^{N^2-s(N)^2}$$
$$\leq 2^{(k+1)^2\alpha\sigma^2\ N^2}\cdot\left(\max_{\text{safe }\psi\in\Psi}|[l](\psi)|\right)^{N/\sigma^2}\cdot 2^{N^2-s(N)^2}$$
$$[\text{from (3)}].$$

(Note that each "max" in the lines above is over a nonempty set by (5).) Solving for the max gives

$$\max_{\text{safe }\psi\in\Psi}|[l](\psi)|$$
$$\geq\left(2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{3})N^2-(k+1)^2\alpha\sigma^2\ N^2-N^2+s(N)^2}\right)^{\sigma^2/N}$$
$$=2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{3}-(k+1)^2\alpha\sigma^2-1+(s(N)^2/N^2))\sigma^2\ N}.$$

In other words, since any safe ψ is an $f(N)\times f(N)$ square in which each strip of height $(k+1)$ contains labelings from a particular component of G_h, one gets

$$|L_{H,G_v}(f(N),f(N))| \tag{7}$$
$$\geq 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{3}-(k+1)^2\alpha\sigma^2-1+(s(N)^2/N^2))\sigma^2N}$$
$$\geq 2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2}-(k+1)^2\alpha\sigma^2)\sigma^2N} \tag{8}$$

for N large enough, since $s(N)^2/N^2$ approaches one from below as $N\to\infty$. Therefore,

$$\lim_{N\to\infty}\frac{\log|L_{H,G_v}(N,N)|}{N^2}$$
$$=\lim_{N\to\infty}\frac{\log|L_{H,G_v}(f(N),f(N))|}{f(N)^2}$$
$$\geq\lim_{N\to\infty}\frac{\log(2^{(C_{\text{hex}}(d,k)-\frac{\epsilon}{2}-(k+1)^2\alpha\sigma^2)\sigma^2\ N})}{\sigma^2\ N}\quad[\text{from (8)}]$$
$$=C_{\text{hex}}(d,k)-\frac{\epsilon}{2}-(k+1)^2\alpha\sigma^2$$
$$\geq C_{\text{hex}}(d,k)-\epsilon\quad[\text{from (1)}].$$

Note that the first two limits above exist by subadditivity (e.g. [12]). Thus, we have

$$C_{\text{hex}}(d,k)$$
$$\leq\lim_{N\to\infty}\frac{\log|L_{H,G_v}(N,N)|}{N^2}\quad(\text{since }\epsilon\text{ was arbitrary})$$
$$\leq C_{\text{hex}}(d,k)\quad(\text{since }L_{H,G_v}(N,N)\subseteq L_{G_h,G_v}(N,N)).$$

If we now start with H and G_v (instead of G_h and G_v) and repeat the argument in the vertical direction, then we will obtain

$$\lim_{N\to\infty}\frac{\log|L_{H,V}(N,N)|}{N^2}=\lim_{N\to\infty}\frac{\log|L_{H,G_v}(N,N)|}{N^2}$$
$$=C_{\text{hex}}(d,k),$$

which completes the proof of the lemma. ■

F. Zero Capacity Theorem

The following theorem establishes that $C_{\text{hex}}(d,k)=0$ whenever the Constant Position Algorithm outputs "success".

Theorem II.9: If the hexagonal (d,k) constraint has the constant position property, then the hexagonal (d,k) capacity is zero.

Proof: Lemma II.5 showed that there is a bijection between valid labelings of strips and certain walks through G_h and G_v, and so instead of counting valid labelings, it suffices to count corresponding walks.

Let s_i denote the $(k+1)\times N$ horizontal strip (in an $M\times N$ rectangle), whose bottom row is the ith row, counting from the bottom, of the $M\times N$ rectangle, i.e. $1\leq i\leq M-k$.

Let W_a be a valid labeling of the bottommost $(k+1)\times N$ horizontal strip, s_1. Let W_b be a valid labeling of s_2, such that W_a and W_b agree on $s_1\cap s_2$.

Suppose the walk through G_h corresponding to W_a lies in a component H_a, and the walk corresponding to W_b lies in a component H_b (these components must be cyclic since $N>k+1$). Since W_a and W_b agree on their common $k\times N$ horizontal strip (i.e., on rows 2 through $k+1$ from the bottom of the $M\times N$ rectangle), the labeling λ_a of any $(k+1)\times(k+1)$ square labeled by W_a, and the labeling λ_b of the $(k+1)\times(k+1)$ square shifted one row upward labeled by W_b, satisfy $(\lambda_a,\lambda_b)\in E_v$. Thus (H_a,H_b) is a pair of vertically semi-compatible components identified in Step 4.

Now, since the hexagonal (d,k) constraint has the constant position property by assumption, let $j\in\{1,\ldots,k+1\}$ be an index corresponding to (H_a,H_b) specified in the constant position property.

By the constant position property, for each particular labeling W_a of s_1, the labeling by W_b of column j of the top row of the leftmost $(k+1)\times(k+1)$ square in s_2 is completely determined by the labeling by W_a of the leftmost $(k+1)\times(k+1)$ square in s_1. (Recall that the walks corresponding to W_a and W_b do not transition between different components of G_h.) The same fact remains true if the two $(k+1)\times(k+1)$ squares slide together one column at a time from left to right. Thus, the sequence of values of the labeling by W_b in the top row of s_2, from horizontal positions j to $N-(k+1-j)$ in the $M\times N$ rectangle, are all completely determined by the labeling of s_1. These positions consist of at least the middle $N-2k$ positions in the top row of s_2, for any j. In sum, the labeling of s_1 immediately determines the labeling of k out of $k+1$ rows of s_2, and now we have shown that it also determines all but at most $2k$ of the positions in the top row of s_2. As a result, the number of different possible valid labelings of s_2, corresponding to walks from a given cyclic component of G_h, for a given labeling of s_1 corresponding to walks from a (possibly different) given cyclic component of G_h, is at most 2^{2k}. Varying the labelings W_b of s_2 over the α cyclic components of G_h increases this upper bound to at most $\alpha 2^{2k}$.

Continuing the argument from the previous paragraph inductively, moving one row upward in the $M\times N$ rectangle each time, shows that, for a given labeling of s_1, there are at most $(\alpha 2^{2k})^{M-k}$ ways to label the $M\times N$ rectangle excluding s_1.

Then, using the loose upper bound that there are at most $2^{(k+1)N}$ possible valid labelings of s_1 corresponding to a walks through any particular cyclic component of G_h, there are at most

$$2^{(k+1)N}(\alpha 2^{2k})^{M-k} \leq 2^{(k+1)N+(2k+\log\alpha)M} \tag{9}$$

labelings in $L_{H,V}(M,N)$. Therefore,

$$\begin{aligned}
0 &\leq C_{\text{hex}}(d,k) \\
&= \lim_{M,N\to\infty} \frac{\log|L_{H,V}(M,N)|}{MN} \\
&\qquad\qquad [\text{from Lemma II.8}] \\
&\leq \lim_{M,N\to\infty} \frac{\log\left(2^{(k+1)N+(2k+\log\alpha)M}\right)}{MN} \\
&\qquad\qquad [\text{from (9)}] \\
&= \lim_{M,N\to\infty} \frac{(k+1)N+(2k+\log\alpha)M}{MN} \\
&= 0.
\end{aligned}$$

■

By successful execution of the Constant Position Algorithm presented in Section II-D, we have obtained the following result.

Result II.10: The Constant Position Algorithm verified that the following hexagonal (d,k) constraints satisfy the constant position property:

- $k = d+2$ and $2 \leq d \leq 9$
- $k = d+3$ and $3 \leq d \leq 9$
- $k = d+4$ and $d \in \{4,5,7,9\}$.

The following corollary follows from Result II.10 and Theorem II.9. We note that, while the proof of this corollary is rigorous, verification of the constant position property aspect of it appears to be virtually impossible to do by hand, and relies on computer-assisted verification of an enormous number of cases.

Corollary II.11: The hexagonal (d,k) capacity is zero whenever

- $k = d+2$ and $2 \leq d \leq 9$
- $k = d+3$ and $3 \leq d \leq 9$
- $k = d+4$ and $d \in \{4,5,7,9\}$.

We note that some of the zero capacities in Corollary II.11 immediately imply others,[2] since $C_{\text{hex}}(d+1,k) \leq C_{\text{hex}}(d,k)$ and $C_{\text{hex}}(d,k-1) \leq C_{\text{hex}}(d,k)$, but we include them since they were previously unproven. The following conjecture states a converse to Theorem II.9.

Conjecture II.12: If the hexagonal (d,k) capacity is zero, then the hexagonal (d,k) constraint has the constant position property.

While we presently do not know if this conjecture is true in general, we have verified it computationally for a substantial number of cases for small d and k, namely for all $d \leq 9$.

Theorem II.13: Conjecture II.12 is true for all $d \leq 9$.

[2]Specifically, $C_{\text{hex}}(4,8) = 0$ implies $C_{\text{hex}}(4,7) = 0$ and $C_{\text{hex}}(5,8) = 0$; $C_{\text{hex}}(5,9) = 0$ implies $C_{\text{hex}}(5,8) = 0$ and $C_{\text{hex}}(6,9) = 0$; $C_{\text{hex}}(7,11) = 0$ implies $C_{\text{hex}}(7,10) = 0$ and $C_{\text{hex}}(8,11) = 0$; and $C_{\text{hex}}(9,13) = 0$ implies $C_{\text{hex}}(9,12) = 0$.

G. Algorithm Implementation Details

We describe below specific implementation details of each step of the Constant Position Algorithm, and indicate the greatest computational burdens and ways to reduce complexity.

1) Step 1: Creating the Valid Labelings of a $(k+1)\times(k+1)$ Square: An iterative method is used that recursively creates the valid labelings of a $(k+1)\times(k+1)$ square from valid labelings of smaller rectangles. Pointers are used to represent adjacent subsquares in labelings, which leads to a massive saving in computational complexity in Step 2.

Let B_m be the collection of valid labelings of a $(k+1)\times m$ rectangle. We first create $B_1, B_2, \ldots, B_{d+1}$ (in that order) and then use B_{d+1} to create B_{k+1}, thus avoiding explicit generation of $B_{d+2}, \ldots, B_k$. Generating these sets wastes memory, since the d constraint has no effect on the validity of labelings once $m > d+1$, and the k constraint does not start having an effect until $m = k+1$. In fact, the sizes of $B_{d+2}, \ldots, B_k$ can be quite large, but B_{k+1} is generally much smaller, since the k constraint plays a role. Thus, directly generating B_{k+1} from B_{d+1} saves memory at the expense of extra computation.

In order to create B_1, all valid labelings of a $(k+1)\times 1$ rectangle (i.e., a column) are generated directly, and each is given a unique ID. To create B_2, it is determined which labelings of B_1 may be placed horizontally next to each other to create valid labelings of a $(k+1)\times 2$ rectangle.

In order to create $B_3, B_4, \ldots,$ a particular data structure built from pointers (which will be called IDs) is used to represent a labeling. Suppose $2 \leq m \leq d+1$, and let λ be a labeling in B_m. Also, let $\lambda_a, \lambda_b \in B_{m-1}$ be the labelings of the left-most and right-most $(k+1)\times(m-1)$ subrectangles, respectively, of the $(k+1)\times m$ rectangle labeled by λ. The labeling λ is represented using a data structure that contains: (1) an ID (unique among labelings in B_m); (2) an array containing the IDs of the labelings of each column; (3) an array containing the IDs of λ_a and λ_b.

Creation of B_m, with $3 \leq m \leq d+1$, is now described. For every two labelings $\lambda, \lambda' \in B_{m-1}$ such that $\lambda_b = \lambda'_a$, a labeling λ^* of a $(k+1)\times m$ rectangle is induced, where the labeling of the left-most $(k+1)\times(m-1)$ rectangle is λ, and the labeling of the right-most column is the labeling of the right-most column of λ'. The new labeling λ^* is valid if and only if the restriction of λ^* to each row and diagonal does not violate the d constraint, since the restriction of λ^* to any column is valid by construction, and the rows and diagonals are not long enough for the labeling to violate the k constraint. If λ^* passes these validity checks, it is added to B_m.

Finally, generating B_{k+1} from B_{d+1} follows essentially the same procedure, except now consideration is made of all $(k-d+1)$-tuples of labelings in B_{d+1} that can be overlaid on each other successively (as in the previous paragraph) to create a labeling of a $(k+1)\times(k+1)$ rectangle. This labeling is valid if and only if the restriction to any row and diagonal does not violate the k constraint, since the restriction to any column is valid by construction, and the restriction to any row or diagonal already satisfies the d constraint, since any $d+1$

consecutive positions are labeled by some labeling in B_{d+1}. If the labeling passes these checks of validity, it is added to B_{k+1}.

Figure 9 shows the complexity reduction using this method compared to two other less efficient methods.

2) Step 2: Creating the label graphs G_h and G_v: A method is given to create the edge sets E_h and E_v. Each construction is described separately, as they are slightly different.

To determine the out-edges in E_h from a labeling (i.e., a vertex of G_h) λ of a $(k+1)\times(k+1)$ square, the data structure from Step 1 that represents λ is useful. Recall that Step 1 assigns an ID to each labeling of a $(k+1)\times(d+1)$ rectangle that appears in λ, and let λ_I denote the $(k-d+1)$-length array of the IDs of these sub-rectangles. Then to check if there is an edge in G_h from λ to a labeling λ' of a $(k+1)\times(k+1)$ square, it suffices to check if $\lambda_I(j)=\lambda'_I(j-1)$ for $j=2,\ldots,k-d+1$. This reduces the problem of determining if $(\lambda,\lambda')\in E_h$ to comparing $(k-d)$ pairs of integers (instead of the more complex comparison of all $k(k+1)$ values of the bit positions in the overlapping rectangles).

The runtime of this step can be reduced by decreasing the number of pairs of valid square labelings that are examined. In practice, for the constraints of interest, each valid square labeling has relatively few out-edges in E_h (typically, almost all have out-degree equal to one, and the rest generally have out-degree less than 5). As a pre-processing step, each labeling λ' is sorted by the element $\lambda'_I(1)$, which is the ID of the leftmost $(k+1)\times(d+1)$ sub-rectangle of λ'. Then for each square labeling λ, only square labelings λ' satisfying $\lambda'_I(1)=\lambda_I(2)$ are examined to find the out-edges from λ in E_h.

However, the same method cannot be used in determining the edges of G_v, since no record of any $(d+1)\times(k+1)$ sub-rectangles was made in Step 1. Nevertheless, a similar pre-processing step is performed by assigning an integer ID to each valid $(k+1)\times 1$ column, and then sorting the square labelings based on the ID of each square's leftmost column. Then a search is made for out-edges in E_v only between square labelings λ and λ' for which the labeling of the top k positions of the leftmost column of λ agree with the labeling of the bottom k positions of the leftmost column of λ'. This method indeed helps in practice, even though it does not prune the potential set of edges in E_v as much as the previous method pruned the potential set of edges in E_h.

3) Step 3: Finding the Components of the label graphs G_h and G_v: We prune the label graphs G_h and G_v by iteratively removing the square labelings from Λ that are sources or sinks in either G_h or G_v. When such a labeling is removed, the corresponding vertices in both G_h and G_v are eliminated, since if a square labeling is a source or sink in either graph, then it cannot appear in a labeling of the infinite plane, and so it can be removed from both graphs (see Lemma II.8).

Since removing sources or sinks from one graph may create more sources or sinks in the other graph, the implementation of the Constant Position Algorithm ping-pongs between removing sources and sinks from both G_h and G_v until the process halts.

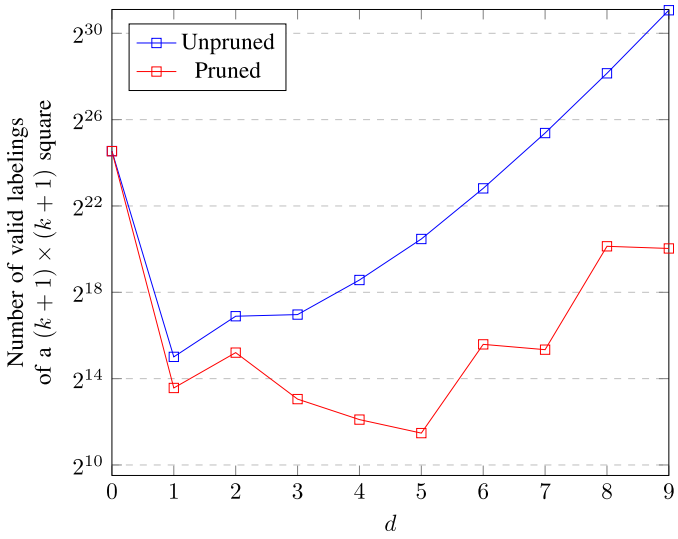


Fig. 7. Plot showing the number of valid labelings of a $(k+1)\times(k+1)$ square versus d, and the number of pruned valid labelings of a $(k+1)\times(k+1)$ square versus d, where $k=d+4$.

We note that Tarjan's algorithm for finding the components of G_h (as described next) would also identify the sources and sinks in G_h, but our tests indicate that pruning sources and sinks drastically reduces the number of valid labelings of $(k+1)\times(k+1)$ squares (see Figure 7), and so it was performed here to reduce memory consumption.

The components of the pruned G_h are determined using Tarjan's algorithm [21]. The runtime of Tarjan's algorithm is linear in the number of vertices and edges, i.e., $O(|\Lambda|+|E_h|)$. The pruning step described previously is in part to prevent stack overflow, as the implementation of Tarjan's algorithm is recursive, and some (d,k) constraints allow on the order of billions of valid labelings of a $(k+1)\times(k+1)$ square, which could potentially result in as many levels of recursion.

Note that no explicit determination is made of the components of G_v, since such information is not needed in the rest of the algorithm.

4) Step 4: Finding the Pairs (H_a,H_b) of vertically semi-compatible Components of G_h: To find pairs (H_a,H_b) that are vertically semi-compatible, we examine each component H_a of the pruned G_h, and every labeling λ_a in H_a, and check if λ_a has an out-edge in the pruned G_v to a labeling in H_b. Note that this condition does not guarantee that a walk through H_a and a walk through H_b could correspond to labelings of (vertically) successive $(k+1)\times N$ horizontal strips in an $M\times N$ rectangle. However, this weaker condition is much simpler to verify, and performs well despite creating more cases to check, since in practice the out-degree of most vertices is small.

5) Step 5: Showing the Hexagonal (d,k) Constraint Has the constant position property: For the particular hexagonal (d,k) constraints considered in this paper, we observed that each component of the pruned G_h is vertically semi-compatible with a very small number of other components, often just one or two (see Table II). This fact, combined with the small out-degrees of vertices in the pruned G_v, allows relatively quick verification of the constant position property.

TABLE II

COMPUTATIONAL COMPLEXITY PARAMETERS FOR ALL HEXAGONAL (d,k) CONSTRAINTS WITH $d \leq 9$ WHERE $C_{\text{HEX}}(d,k) = 0$. THE QUANTITIES REFER TO THE SIZE OF VARIOUS PARAMETERS OF THE DIRECTED GRAPH G_h, EITHER BEFORE OR AFTER PRUNING. OUR NEWLY DISCOVERED CASES FOR $k = d+3$ AND $k = d+4$ ARE INDICATED BY THE ASTERISKS ON THE FAR LEFT. EVERY ROW IN THIS TABLE CORRESPONDS TO A HEXAGONAL (d,k) CONSTRAINT THAT WAS CONFIRMED VIA COMPUTER SEARCH TO SATISFY THE CONSTANT POSITION PROPERTY

(d,k)	Number of vertices $\lvert\Lambda\rvert$ before pruning	Number of edges $\lvert E_h\rvert$ before pruning	Number of vertices after pruning	Number of components α	Average number of vertices per component after pruning	Non-cyclic components	Number of vertically semi-compatible pairs (H_a, H_b)
$(1,2)$	3	3	3	1	3.0	0	1
$(1,3)$	20	20	3	1	3.0	0	1
$(2,3)$	12	11	3	1	3.0	0	1
$(2,4)$	135	170	65	1	65.0	0	1
$(3,4)$	19	15	15	3	5.0	0	3
$(3,5)$	186	150	15	3	5.0	0	3
* $(3,6)$	3,539	4,082	202	19	10.6	0	19
$(4,5)$	82	51	15	3	5.0	0	3
$(4,6)$	817	678	202	19	10.6	0	19
* $(4,7)$	13,400	15,020	234	19	12.3	0	19
* $(4,8)$	388,397	572,552	4,391	252	17.4	0	264
$(5,6)$	329	174	133	19	7.0	0	19
$(5,7)$	3,158	2,040	133	19	7.0	0	19
* $(5,8)$	53,701	49,201	2,384	245	9.7	0	249
* $(5,9)$	1,449,120	1,883,744	2,846	235	12.1	0	237
$(6,7)$	1,756	783	133	19	7.0	0	19
$(6,8)$	17,281	10,480	2,220	241	9.2	0	241
* $(6,9)$	283,200	243,603	2,248	241	9.3	0	241
$(7,8)$	10,133	4,534	2,025	225	9.0	0	225
$(7,9)$	102,614	56,923	2,025	225	9.0	0	225
* $(7,10)$	1,696,233	1,324,066	39,946	3,663	10.9	0	3,663
* $(7,11)$	43,511,098	47,066,939	41,478	4,431	9.4	736	4,478
$(8,9)$	65,676	26,159	2,025	225	9.0	0	225
$(8,10)$	689,199	354,239	39,946	3,663	10.9	0	3,663
* $(8,11)$	11,667,348	8,594,525	39,974	3,663	10.9	0	3,663
$(9,10)$	462,531	181,758	37,851	3,441	11	0	3,441
$(9,11)$	5,142,892	2,548,549	37,851	3,441	11	0	3,441
* $(9,12)$	88,149,741	60,893,380	1,068,296	82,697	13.0	0	82,697
* $(9,13)$	2,244,615,058	2,132,131,265	1,069,432	83,601	12.8	892	83,652

H. Computational Complexity of the Algorithm

The Constant Position Algorithm was run for a number of previously open cases, and succeeded in showing that $C_{\text{hex}}(d,k) = 0$ in the following five new (d,k) constraints: $(6,9)$, $(4,8)$, $(5,9)$, $(7,11)$, and $(9,13)$. These cases are listed in order of increasing computational complexity. Even though the $(6,9)$ case follows immediately from the $(5,9)$ case, since $C_{\text{hex}}(6,9) \leq C_{\text{hex}}(5,9) = 0$, we include it and other already-known cases in the information below as these cases provide interesting data about complexity.

The run time of the algorithm is shown in Figure 8 for constraints of the form $(d, d+4)$, for $0 \leq d \leq 9$ (the case $(6,9)$ is not shown, but runs about as fast as the $(4,8)$ case). Even though our results were limited in this plot to the cases $d = 4, 5, 7, 9$, the algorithm was run for all $d \leq 9$ in order to better understand the complexity.

There is roughly exponential growth for $1 \leq d \leq 9$, but there is a sharp decline from $d = 0$ to $d = 1$, with the runtime of the $d = 0$ case on the order of the runtime for the $d = 9$ case. There are two reasons for this phenomenon.

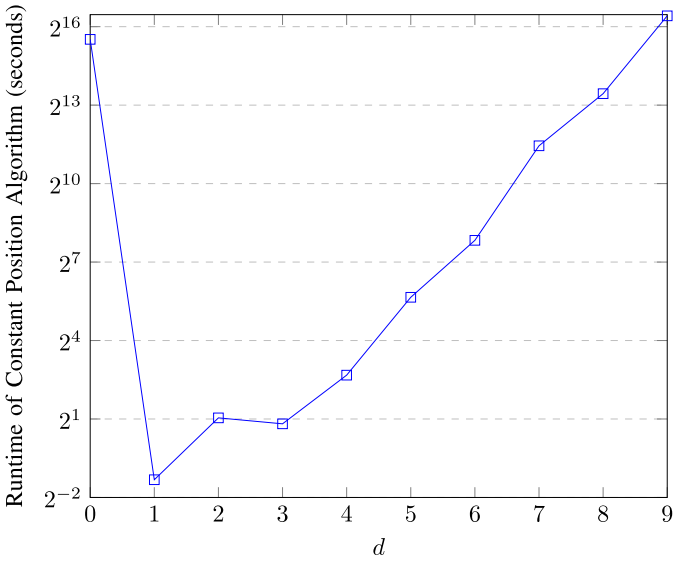


Fig. 8. Runtime, in seconds, on a supercomputer of the constant position algorithm for hexagonal (d, k) constraints, as a function of d, where $k = d + 4$. Note that one day is about $2^{16.4}$ seconds.

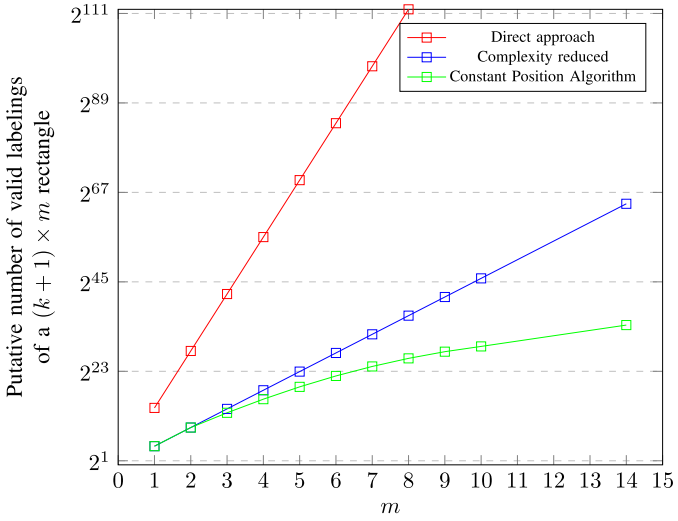


Fig. 9. Plot showing the number of putative valid labelings of a $(k+1) \times m$ rectangle versus m for the case $(d, k) = (9, 13)$ based on three different methods: a direct method, a complexity reduced method, and our adaptive pruning method.

First, as seen in Figure 7, the number of valid $(k+1) \times (k+1)$ squares in the $d = 0$ case, even after pruning, is many orders of magnitude larger than the number of valid $(k+1) \times (k+1)$ squares in the $d = 1$ case. Second, since there is essentially no d constraint in the $d = 0$ case, the graphs G_h and G_v in the $d = 0$ case are highly connected, which slows the Constant Position Algorithm in multiple places. The high connectivity of the graphs G_h and G_v in the $d = 0$ case is evidenced by the fact that even the $(0, 1)$ constraint has positive capacity [1], [10], [11].

The highest complexity case that could be run in reasonable time was when $(d, k) = (9, 13)$, in which case the runtime was about 2^{16} seconds, or about one full day. The next open case of interest would be when $(d, k) = (11, 15)$, which is projected to take at least about 8 days of runtime to complete. Since our supercomputer time allocation was limited to two full days at a time, the $d = 11$ case was not attempted. Furthermore, the amount of memory usage grew exponentially in d, which posed further difficulties.

Table II shows, for each hexagonal (d, k) constraint having zero capacity and with $d \leq 9$, the specific sizes of vertex and edge sets in G_h, the size of G_h after pruning, the number of (both total and non-cyclic) components of G_h, the average number of vertices per component, and the number of pairs of vertically semi-compatible components.

One contribution of the Constant Position Algorithm is its dramatic reduction in computational complexity compared to more straightforward approaches. This complexity improvement enables the algorithm to discover zero hexagonal capacity constraints that would otherwise be computationally prohibitive. In particular, the algorithm adaptively prunes the set of putative valid labelings for each width m rectangle, and recursively builds the set of valid labelings of width $m+1$ from those obtained for width m. This reduction in the number of labelings to inspect makes the overall task more manageable.

A plot of the number of putative valid $(k + 1) \times m$ labelings, as a function of the rectangle width m, is illustrated in Figure 9 for the case $(d, k) = (9, 13)$. The green curve corresponds to our complexity reduced version of the Constant Position Algorithm. The "direct approach" (shown in red) corresponds to a naive implementation without complexity reduction, i.e., checking all $2^{(k+1)m}$ possible labelings of a $(k+1) \times m$ rectangle for validity, one-by-one. The "complexity reduced" method (shown in blue) corresponds to first creating the valid labelings of a column of height $(k + 1)$, and then checking which possible arrangements of these columns create valid labelings of a $(k + 1) \times m$ rectangle. It can be seen that at the highest level of complexity when $m = 14$, the number of labelings that we must check for validity in our algorithm is about 2^{34}, which is considerably lower than the roughly 2^{64} required for the complexity-reduced blue curve.

A plot of the number of valid $(k + 1) \times (k + 1)$ squares versus d, where $k = d+4$, is shown in Figure 7. The blue curve shows the number of valid labelings of a $(k + 1) \times (k + 1)$ square that are found in Step 1 of the algorithm, while the red curve shows the number of these squares retained after pruning all sources and sinks in the label graphs G_h and G_v in Step 2. As the plot indicates, for larger values of d, the vast majority of valid labelings of a $(k + 1) \times (k + 1)$ square do not appear in a component of G_h and a component of G_v, Eliminating these squares from consideration during Step 2 of the algorithm drastically improves the efficiency of the subsequent steps.

1) Computing Resources: Due to the large amount of memory required to run the Constant Position Algorithm for larger values of d and $(k - d)$, the algorithm was implemented on a large-memory node of the Comet supercomputer at the San Diego Supercomputer Center. These nodes have a clock speed of 2.2 GHz. The OpenMP API was used for implementing the parallelism, primarily in Steps 1-3 of the algorithm. We used 64 CPUs in parallel and a total of 800 gigabytes of shared

memory. The computationally most complex case we ran was $(d, k) = (9, 13)$, which took about one full day to run and had about 2 billion valid 14×14 square labelings.

The next unsolved case for which the hexagonal capacity is not known to be zero or positive is $(11, 15)$, which could not be performed with our current resources. Using statistical sampling, we estimate it would have about 80 billion valid 16×16 square labelings, which suggests that the time and space complexities would increase by about 40-fold compared to the $(9, 13)$ case. Such resources are presently not easily available.

III. Forbidden String Algorithm for Proving Zero Hexagonal (d, k) Capacity

In this section we present a second algorithm for automatically and rigorously proving that certain hexagonal (d, k) capacities are zero. In this entire section, we will frequently use the notation $A(i, j)$ to denote the labeling of a position in an array by a binary value, or else an "Unused" indicator. The position (i, j) uses ordinary Cartesian integer coordinates.[3]

A binary string is *forbidden* for a hexagonal (d, k) constraint if there exists a positive integer N such that for any rectangle with side lengths at least N, no valid labelings of the rectangle contain the string in any horizontal, vertical, or northeast diagonal. For some pairs (d, k) there exist one or more forbidden strings of the form 10^z1, where $d \leq z \leq k$, whereas for other pairs no such strings are forbidden.

If it is known that $C_{\text{hex}}(d, k-1) = 0$ and 10^k1 is a forbidden string for the hexagonal (d, k) constraint, then we deduce $C_{\text{hex}}(d, k) = C_{\text{hex}}(d, k-1) = 0$. Similarly, if $C_{\text{hex}}(d+1, k) = 0$ and 10^d1 is a forbidden string for the hexagonal (d, k) constraint, then $C_{\text{hex}}(d, k) = C_{\text{hex}}(d+1, k) = 0$. The basic idea behind our Forbidden String Algorithm is to establish, by a computer-generated proof, that either 10^d1 or 10^k1 is forbidden for the hexagonal (d, k) constraint, and then rely on a previously known fact that either $C_{\text{hex}}(d, k-1) = 0$ or $C_{\text{hex}}(d+1, k) = 0$.

A. Non-Forbidden Strings

Sometimes, however, it is provably impossible for certain strings 10^z1 to be forbidden. Here, we demonstrate a number of such cases, and thereby concentrate the use of the Forbidden String Algorithm on other cases.

In what follows, we will assume only integer valued coordinates of points in the plane. Suppose all of the points on the northwest line $y = -x + a$ and the northeast line $y = x + c$ are labeled 1 and all other points on the integer lattice are labeled 0. These two lines intersect if and only if a and c are either both even or both odd. As a result, any point on the line $y = -x+a$ lies on the same northeast line as one point on the line $y = -x+b$ if and only iff a and b have the same parity, or equivalently, if and only if $b - a$ is even. The horizontal (and vertical) number of points between the two northwest lines $y = -x + a$ and $y = -x + b$ is $b - a - 1$.

[3]In contrast, in Section IV, we use matrix (row,column) coordinates for our arrays.

Suppose (u, v) lies on the line $y = -x+a$ and (s, t) lies on the line $y = -x+b$, and both points lie on the same northeast line $y = x + c$. Then $u = (a - c)/2$, $v = (a + c)/2$, $s = (b-c)/2$, and $t = (b+c)/2$, so $(s, t) = (u, v)+(1, 1)(b-a)/2$. Thus, the diagonal number of points between these two lines is $\frac{b-a}{2} - 1 = \frac{z-1}{2}$, where $z = b-a-1$. In other words, if two northwest lines, separated horizontally by an odd number z of points, are labeled by 1s and every other point is labeled 0, then the string 10^z1 appears horizontally and vertically, and the string $10^{(z-1)/2}1$ appears diagonally.

For any sequence of nonnegative integers $a_1, \ldots, a_n$, define $s_j = j + a_1 + \cdots + a_j$ for all $j \geq 1$ and let $s_0 = 0$. Define a labeling by

$$L_{a_1,\ldots,a_n}(x, y) = \begin{cases} 1 & \text{if } x + y = s_j \mod s_n \text{ where } 0 \leq j \leq n-1 \\ 0 & \text{else.} \end{cases}$$

Then $L_{a_1,\ldots,a_n}$ consists of periodically repeating infinite northwest diagonals of 1s. Each infinite horizontal row is labeled by infinitely repeating the pattern $10^{a_1}10^{a_2}1 \ldots 10^{a_n}$ to the left and right. The 1 on the far left side of this pattern is at the origin, and the entire pattern shifts one position to the left each time one moves upward by one row. This lattice contains runs of a_1, a_2, …,a_n zeros horizontally and vertically between consecutive 1s.

If a_i is odd, then the string 0^{a_i} appears horizontally and vertically and the string $0^{(a_i-1)/2}$ appears diagonally, between the consecutive northwest diagonal lines passing through the points $(s_{i-1}, 0)$ and $(s_i, 0)$ in the labeling $L_{a_1,\ldots,a_n}$.

On the other hand, if a_i and a_{i+1} are both even, then the number of points horizontally between the northwest diagonal lines passing through $(s_{i-1}, 0)$ and $(s_{i+1}, 0)$ is odd, so the string $0^{(a_i+a_{i+1})/2}$ appears diagonally between these two diagonal lines (i.e. it skips the diagonal line passing through the point $(s_i, 0)$). The string 0^{a_i} appears horizontally and vertically between the diagonal lines passing through $(s_{i-1}, 0)$ and $(s_i, 0)$, and the string $0^{a_{i+1}}$ appears horizontally and vertically between the diagonal lines passing through $(s_i, 0)$ and $(s_{i+1}, 0)$.

The following theorem eliminates certain possible strings as being forbidden.

Theorem III.1: For a given hexagonal (d, k) constraint, the string 10^z1 is not forbidden whenever $\hat{d} \leq z \leq \hat{k}$, where

$$\hat{d} = \begin{cases} d & \text{if } d \text{ even} \\ d+1 & \text{if } d \text{ odd} \end{cases} \qquad \hat{k} = \begin{cases} k & \text{if } d \text{ even} \\ k-1 & \text{if } d \text{ odd.} \end{cases}$$

Proof: Take $a_1 = \hat{d}$, $a_2 = \hat{d} + 2$, $a_3 = \hat{d} + 4$, $\ldots a_n = \hat{k}$ in $L_{a_1,\ldots,a_n}$, and note that each a_i is even.

When $1 \leq i \leq n-1$, we have $a_{i+1} = a_i + 2$, so the string $0^{(a_i+a_{i+1})/2} = 0^{a_i+1}$ appears diagonally and the strings 0^{a_i} and $0^{a_{i+1}}$ appear horizontally and vertically. And when $i = n$, the string $0^{(a_1+a_n)/2}$ appears diagonally and the strings 0^{a_1} and 0^{a_n} appear horizontally and vertically.

In summary, the pattern 10^z1 appears along every horizontal row and vertical column whenever z is even and $\hat{d} \leq z \leq \hat{k}$. Also, since $(a_i + a_{i+1})/2 = ((\hat{d} + 2(i-1)) + (\hat{d} + 2(i-1) + 1)/2 = \hat{d} + 2i - 1$, the pattern 10^z1

1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	1	0	0	0	0	0	0

Fig. 10. A tileable valid labeling of a 15×15 square for the hexagonal $(5, 8)$ constraint. The string $10^5 1$ appears in the top row (in red) and thus is not forbidden.

appears along every northeast diagonal whenever z is odd and $\hat{d} \leq z \leq \hat{k}$.

Thus, the labeling satisfies the hexagonal (d, k) constraint and $10^z 1$ is not forbidden, since it appears in the valid labeling $L_{\hat{d},\hat{d}+2,\ldots,\hat{k}}$. ■

The Forbidden String Algorithm relies on deducing that $C_{\text{hex}}(d, k) = C_{\text{hex}}(d + 1, k)$ by showing $10^d 1$ is forbidden, or $C_{\text{hex}}(d, k) = C_{\text{hex}}(d, k - 1)$ by showing $10^k 1$ is forbidden. Theorem III.1 shows that, in particular, if both d and k are even, then this approach will not work, since $10^d 1$ and $10^k 1$ are not forbidden strings of the hexagonal (d, k) constraint.

In some cases, certain strings of the form $10^z 1$ cannot be forbidden for a given hexagonal (d, k) constraint, but they do not fall within the scope of Theorem III.1. In the theorem below, for each hexagonal (d, k) constraint given, we exhibit a single square labeling that can validly tile the plane, and we deduce therefore that any strings within it are not forbidden.

Theorem III.2: The string $10^z 1$ is not forbidden for the hexagonal (d, k) constraint in the following cases: (i) $d = 5$, $k = 8$, $z = d$; (ii) $d = 7$, $k = 11$, $z \in \{d, k\}$.

Proof: The following square labelings in Figure 10 and Figure 11 satisfy the mentioned constraint, can tile the plane, and contain the strings asserted to not be forbidden. Figure 12 shows portions of the hexagonal lattice tiled by these labelings. ■

B. Algorithm Description

Forbidden String Algorithm

- **Step 1:** Set $A(i, j) =$ Unused, for all i, j.
- **Step 2:** Assume $A(0, 0) = 1$ and $A(z + 1, 0) = 1$.
- **Step 3:** Force 0s for the d constraint.
- **Step 4:** If there does not exist 0^{k+1} horizontally, vertically, or diagonally, then go to Step 5.

  Else repeatedly pop the stack until the top of the stack is an assumed 1.

  If the top of the stack is the original assumption $A(0, 0) = 1$, then a proof is found, so exit.

  Else convert the top of the stack to a forced 0 and repeat Step 4.
- **Step 5:** Select an unused position (x, y) and assume $A(x, y) = 1$.
- **Step 6:** Go to Step 3.

1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0

Fig. 11. A tileable valid labeling of a 24×24 square for the hexagonal $(7, 11)$ constraint. The strings $10^7 1$ and $10^{11} 1$ appear (in red) in rows 1 and 2, respectively, and thus are not forbidden.

C. Algorithm Details

The nonnegative integer z is an input parameter chosen to determine the initial assumed string $10^z 1$ in Step 2. In every case tested, we chose $z = d$ or $z = k$.

Each point in the array A has integer coordinates and is marked as either "Unused", or else labeled "0" or "1". The goal is to try to show that the original two assumptions in Step 2 lead to a violation of the hexagonal (d, k) constraint, thus implying that $10^z 1$ is a forbidden string.

A stack is used to keep track of the current state of the algorithm, namely which array positions (i, j) are labeled 0 or 1 and whether they achieved those values by being assumed or by being forced. Each "assume" or "force" instruction in the algorithm is also pushed onto the top of the stack.

In Step 3, the algorithm forces unused points to be 0 if they lie within d positions of a point labeled 1 horizontally, vertically, or diagonally. In other words,

- If $A(x + i, y) =$ Unused and $1 \leq |i| \leq d$, then force $A(x + i, y) = 0$.

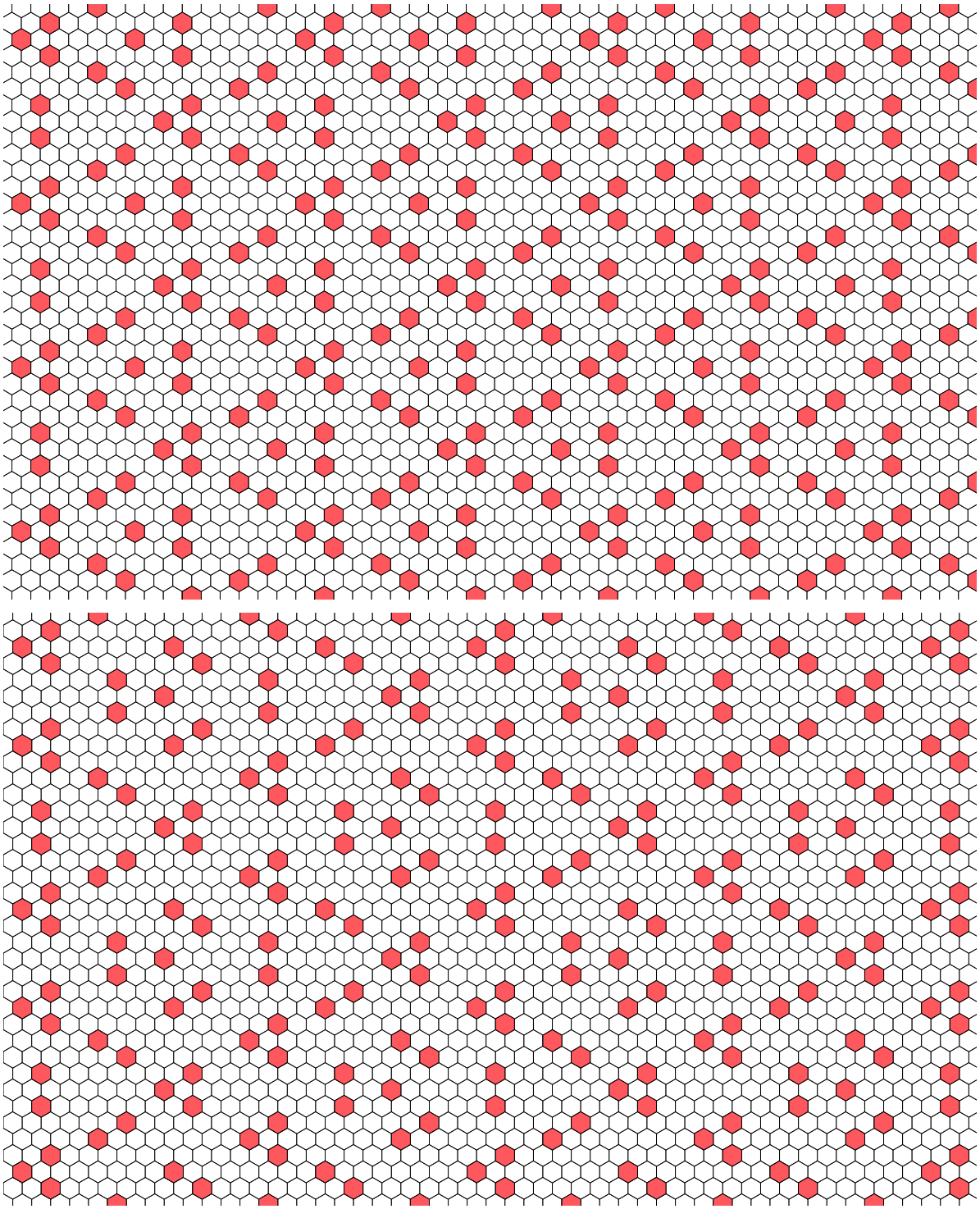


Fig. 12. The upper labeling satisfies the hexagonal $(5, 8)$ constraint, and the lower labeling satisfies the hexagonal $(7, 11)$ constraint. Red hexagons indicate 1s, and white hexagons indicate 0s. The upper labeling shows $10^5 1$ is not a forbidden string under the hexagonal $(5, 8)$ constraint, and the lower labeling shows $10^7 1$ and $10^{11} 1$ are not forbidden strings under the hexagonal $(7, 11)$ constraint.

- If $A(x, y + i) =$ Unused and $1 \leq |i| \leq d$, then force $A(x, y + i) = 0$.
- If $A(x + i, y + i) =$ Unused and $1 \leq |i| \leq d$, then force $A(x + i, y + i) = 0$.

Since the algorithm always forces 0s after a 1 is added, the array A is guaranteed to obey the d constraint. If $k < 2d$, then additional 0s can be forced when a 1 is inserted, in order to prevent a violation of the k constraint. Specifically, in addition to points at distances $1, \ldots, d$ in all 3 directions, 0s can be forced at distances $k + 2, \ldots, 2d + 1$ in all 3 directions, since otherwise, if any of these points were labeled 1, a string of 0^{k+1} would result.

Step 4 checks for violations of the k constraint, and, if found, then removes previously forced bits until the top of the stack is an assumed 1, at which point the assumed 1 is converted to a forced 0. We then repeat Step 4 until the k constraint is enforced. To reduce search complexity, each stack push records the smallest bounding rectangle containing all 0s and 1s in the array up to that point. Only that bounding rectangle, rather than the entire array A, needs to be searched for strings 0^{k+1}.

In Step 5, the algorithm finds some unused position and labels it as an assumed 1. This can be a deterministic process or randomized. We chose to search the array loosely in an order defined by a spiral space filling curve starting at the origin and eventually covering the entire array, but with an additional randomized component.

In practice an $N \times N$ char array is used, with the coordinates indexed from 0 to $N-1$, and N chosen sufficiently large (e.g., $N = 100$ generally suffices). The location $(N/2, N/2)$ is chosen as the origin. If the algorithm ever attempts to assume or force a labeling of a position (x, y) that lies outside the range of the array A, then the algorithm halts and declares a failure to find a proof of a forbidden string. This can occur when there exists a valid labeling of all locations in the array, under the original assumptions.

A recursive implementation of the Forbidden String Algorithm is given in Appendix A, which may be useful for alternate implementations or a formal correctness proof.

D. Example for $d = 1$ and $k = 3$

We illustrate in Figure 13 the Forbidden String Algorithm with the hexagonal $(1, 3)$ constraint, by showing that $10^3 1$ is a forbidden string, and thus $C_{\text{hex}}(1, 3) = C_{\text{hex}}(1, 2) = 0$. The automated proof is shown on the left-hand side and the labeled bits at various stages on the right-hand side. Each time an "unused" (i.e., not labled) bit is labeled either 0 or 1, that bit labeling is pushed onto a stack and is represented by an extra indentation of the proof line in the figure. Conversely, when a bit is popped off the stack it is marked as "unused" and a reduction of indentation occurs.

For example, the original two assumed 1s are pushed on the stack at lines 1 and 8 of the proof, at locations $(0, 0)$ and $(4, 0)$, respectively. The forced 0s that resulted from these initial 1s are pushed after each assumption, namely in lines 2–7 and lines 9–14 of the proof. The square diagram between lines 11-14 shows the labeled bits based on the original two assumed 1s.

An additional 1 is assumed in line 15 position $(3, 2)$ and its forced 0s follow it in the proof. This assumed 1, however, leads to a horizontal string 0^5 in row 1, as shown in line 22 of the proof. This string violates the fact that $k = 3$, and the violation is illustrated in the square diagram ending at line 22 with the string 0^5 shown in red.

Forced bits are then continually popped off the top of the stack from line 23 through line 28 in the proof, until an assumed 1 is reached. Since this assumed 1 (at location $(3, 2)$) had just led to a contradiction, it is flipped to a forced 0 in line 29, and the resulting bit labelings are shown in the square diagram ending at line 29.

A new 1 is assumed in line 30 at location $(2, 2)$, and its forced 0s follow it in lines 31–35. As a result, a violation of the k constraint occurs due to the string 0^4 occuring diagonally, and this violation is illustrated in the square diagram ending at line 36. Thus the stack pops off forced bits in lines 37–41 until the assumed 1 at $(2, 2)$ is reached. That assumed 1 is converted to a forced 0 in line 42, as illustrated in the diagram ending at line 42.

At this point a 1 is forced at $(3, 1)$ to avoid 0^4 from occuring vertically in column 3. But this 1 forces 0s at $(2, 1)$ and $(4, 2)$, as shown in lines 43–45, thus causing the string 0^4 to occur diagonally, violating the k constraint. This violation is listed in line 46 and shown in the diagram ending at line 46.

```
 1: A( 0, 0) = 1 Assumed
 2:   A( 0, 1) = 0 Forced
 3:     A( 0, -1) = 0 Forced
 4:       A(-1, 0) = 0 Forced
 5:         A(-1, -1) = 0 Forced
 6:           A( 1, 1) = 0 Forced
 7:             A( 1, 0) = 0 Forced
 8:               A( 4, 0) = 1 Assumed
 9:                 A( 4, -1) = 0 Forced
10:                   A( 4, 1) = 0 Forced
11:                     A( 3, 0) = 0 Forced
12:                       A( 3, -1) = 0 Forced
13:                         A( 5, 0) = 0 Forced
14:                           A( 5, 1) = 0 Forced
15:                             A( 3, 2) = 1 Assumed
16:                               A( 3, 1) = 0 Forced
17:                                 A( 2, 1) = 0 Forced
18:                                   A( 2, 2) = 0 Forced
19:                                     A( 3, 3) = 0 Forced
20:                                       A( 4, 2) = 0 Forced
21:                                         A( 4, 3) = 0 Forced
22:                                         k-violation in row 1
23:                                       A( 4, 3) = Unused
24:                                     A( 4, 2) = Unused
25:                                   A( 3, 3) = Unused
26:                                 A( 2, 2) = Unused
27:                               A( 2, 1) = Unused
28:                             A( 3, 1) = Unused
29:                             A( 3, 2) = 0 Flipped bit
30:                               A( 2, 2) = 1 Assumed
31:                                 A( 2, 1) = 0 Forced
32:                                   A( 1, 1) = 0 Forced
33:                                     A( 1, 2) = 0 Forced
34:                                       A( 2, 3) = 0 Forced
35:                                         A( 3, 3) = 0 Forced
36:                                         k-violation in diagonal 1
37:                                       A( 3, 3) = Unused
38:                                     A( 2, 3) = Unused
39:                                   A( 1, 2) = Unused
40:                                 A( 1, 1) = Unused
41:                               A( 2, 1) = Unused
42:                               A( 2, 2) = 0 Flipped bit
43:                                 A( 3, 1) = 1 Forced
44:                                   A( 2, 1) = 0 Forced
45:                                     A( 4, 2) = 0 Forced
46:                                     k-violation in diagonal 1
47:                                   A( 4, 2) = Unused
48:                                 A( 2, 1) = Unused
49:                               A( 3, 1) = Unused
50:                             A( 2, 2) = Unused
51:                           A( 3, 2) = Unused
52:                         A( 5, 1) = Unused
53:                       A( 5, 0) = Unused
54:                     A( 3, -1) = Unused
55:                   A( 3, 0) = Unused
56:                 A( 4, 1) = Unused
57:               A( 4, -1) = Unused
58:               A( 4, 0) = 0 Flipped bit
59:               Contradiction
QED
```

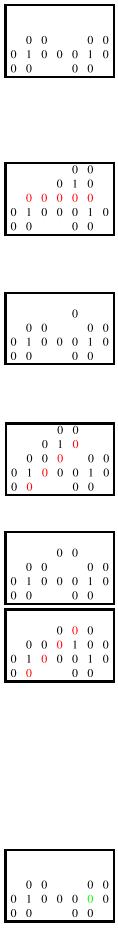


Fig. 13. Proof automatically generated by the forbidden string algorithm. The proof shows that $10^3 1$ is a forbidden string for the hexagonal $(1, 3)$ constraint, and thus $C_{\text{hex}}(1, 3) = C_{\text{hex}}(1, 2) = 0$. The level of indentation indicates the number of assigned binary labels in the plane. Snapshots of the assumed and forced bit values are shown at various points in the proof.

Finally, the stack is popped again until the original assumed 1 at $(4, 0)$ is reached, in lines 47–57, and this assumed 1 is converted to a 0 in line 58, as shown in the diagram ending at line 58. This line, however, finishes the proof, since one concludes that the original two assumed 1s lead to a contradiction.

E. Algorithmic Zero Capacity Results

In searching for forbidden strings of the form 10^d1 or 10^k1, we need not consider cases when d is even or when k is even (by Theorem III.1). The following theorem is thus limited to strings 10^d1 with odd d, or strings 10^k1 with odd k. Each of the strings in the theorem was automatically proven to be forbidden by the Forbidden String Algorithm.

Theorem III.3: The following are forbidden strings for the indicated hexagonal (d, k) constraints:

d	k	forbidden string
3	6	10^d1
4	7	10^k1
6	9	10^k1
7	10	10^d1
8	11	10^k1
9	12	10^d1
10	13	10^k1
11	14	10^d1

Corollary III.4: The hexagonal (d, k) capacity is zero when $k = d + 3$ and $d \in \{3, 4, 6, 7, 8, 9, 10, 11\}$.

Proof: It was stated in [13] and proven in our Part I that $C_{\text{hex}}(d, d + 2) = 0$ for all $d > 0$. Thus, Theorem III.3 implies that $C_{\text{hex}}(d, d + 3) = C_{\text{hex}}(d + 1, d + 3) = 0$ when $d \in \{3, 7, 9, 11\}$ and $C_{\text{hex}}(d, d + 3) = C_{\text{hex}}(d, d + 2) = 0$ when $d \in \{4, 6, 8, 10\}$. ∎

We note that in Corollary III.4, the proofs of $C_{\text{hex}}(3, 6) = 0$ and $C_{\text{hex}}(4, 7) = 0$ each relied on the fact that $C_{\text{hex}}(4, 6) = 0$ (which was proven in our Part I), since 10^31 and 10^71 were forbidden strings, respectively. This gives an alternate derivation of $C_{\text{hex}}(3, 5) = C_{\text{hex}}(5, 7) = 0$, from that given in Part I.

The Constant Position Algorithm found each of the first six cases of zero capacity that were found by the Forbidden String Algorithm in Corollary III.4, but it also found more cases that failed for the Forbidden String Algorithm. Thus, the Constant Position Algorithm gives an improvement over the Forbidden String Algorithm in many cases. On the other hand, the Forbidden String Algorithm is simpler to describe, tends to run much faster, and verifies the interesting property of forbidden strings. Also, the Constant Position Algorithm was unable to prove $C_{\text{hex}}(11, 14) = 0$ with the available supercomputer resources, due to an overflow of memory, whereas the Forbidden String Algorithm was able to prove it.

The Forbidden String Algorithm was implemented in C++ on an Apple MacBook Air laptop computer and took at most a couple of seconds to run, up to the third largest case (i.e., $d = 9$, $k = 12$), and took about 30 minutes for the case $(d, k) = (10, 13)$. The largest case (i.e., $d = 11$, $k = 14$) required more memory and took 42 minutes to run on a supercomputer, pushing assumptions on its stack almost ten billion times before finding the proof. Table III shows some statistics for the Forbidden String Algorithm.

TABLE III
COMPLEXITY STATISTICS FOR THE FORBIDDEN STRING ALGORITHM

d	k	stack pushes	assumed 1s	maximum stack depth
3	6	1,039	111	8
4	7	33,329	1,943	22
6	9	739,886	28,509	16
7	10	1,258,418	39,333	14
8	11	45,060,612	1,194,333	19
9	12	97,422,226	2,229,709	16
10	13	3,856,454,149	78,232,745	22
11	14	9,868,475,943	177,564,097	18

The "stack pushes" column shows the total number of times a push was made on the stack. This quantity reflects the number of times a particular position in the array A was set to either 0 or 1. The "assumed 1s" column shows the total number of times a position in the array A was set to 1 by assumption (i.e., the 1 was not forced). The "maximum stack depth" column shows the largest number of assumed 1s that were ever on the stack at a single time.

IV. RECTANGLE TILING ALGORITHM FOR PROVING POSITIVE HEXAGONAL (d, k) CAPACITY

The following algorithm attempts to automatically discover distinct labelings of $M \times N$ rectangular arrays A and B which are valid no matter how they (jointly) tile the plane, if such rectangle labelings exist. For any i and j, let $A(i, j)$ and $B(i, j)$ denote the value of the binary labeling of position (i, j) in these two arrays, respectively. [4]

The main idea is to randomly label some of the unlabled positions in the rectangles with a 1s, and then fill in all the resulting forced 0s, always checking that the d and k constraints are met. If a constraint is violated, then such assumed labeled positions can be reversed with backtracking.

A. Algorithm Description

Rectangle Tiling Algorithm

- **Step 1:** Set $A(i, j) = B(i, j) =$ Unused, for all i, j.
- **Step 2:** Choose $A(1, 1) = 1$ and $B(1, 1) = 0$.
- **Step 3:** Force 0s for the d constraint in A and B for all tiling configurations.
- **Step 4:** If there does not exist 0^{k+1} horizontally, vertically, or diagonally, for any tiling configurations, then go to Step 5.

  Else repeatedly pop the stack until the top of the stack is a chosen 1.

  If the top of the stack is the original choice $A(1, 1) = 1$, then failure, so exit.

  Else convert the top of the stack to a forced 0 and repeat Step 4.

[4]Matrix notation is used here, where i is the row (increasing downward from 1 to M) and j is the column (increasing rightward from 1 to N).

- **Step 5:** If there are no unused positions in A and B, then the algorithm succeeded, so exit.
- **Step 6:** Select an unused position (x, y) in A or B and set it to 1.
- **Step 7:** Go to Step 3.

B. Algorithm Details

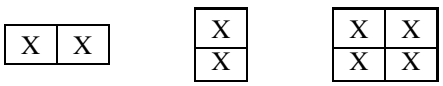
The two arrays A and B are implemented as two-dimensional char arrays in C++, each of size $M \times N$, where M and N are fixed positive integers. Step 1 initializes all MN positions in each of A and B. Step 2 chooses one position where A and B are forced to differ, in order to ensure the two rectangles are different. The upper-left corner of the arrays is one possible choice, but other locations are also feasible, and possibly advantageous. Step 3 enforces the d constraint by forcing nearby 0s. Step 4 enforces the k constraint. If it is violated, then chosen 1s are continually removed from the stack (and converted to forced 0s), until there is no longer a violation. Step 6 tries to find a new position to label 1. Various strategies can be used for this choice, or just a simple random selection.

While the Rectangle Tiling Algorithm shares some features with the Forbidden String Algorithm, they have fundamentally different objectives. The former tries to find two different valid rectangles, whereas the latter tries to find a contradiction to the assumption of a string of the form 10^z1 existing in some valid labeling.

There are nuances to how the two algorithms enforce constraints as well. In the Forbidden String Algorithm, enforcing the d constraint is rather straightforward, but in the Rectangle Tiling Algorithm, many different possible configurations of the rectangles A and B must be considered, since any chosen 1 in one of the two arrays can cause forced 0s in itself or the other array in multiple ways, depending on how they tile the plane. Similarly, checking for violations of the k constraint in the Forbidden String Algorithm is straightforward, although somewhat laborious, whereas in the Rectangle Tiling Algorithm, many different tiling configurations must be examined.

The Rectangle Tiling Algorithm is not guaranteed to terminate. This can happen if there do not exist two rectangular labelings that tile the plane according to the desired constraint, or possibly due to failure to find such rectangular labelings even if they exist.

If the Rectangle Tiling Algorithm does terminate, one can check the two rectangles produced to verify that they do not violate the constraints for arbitrary tilings of the plane. For example, if $M = N > k$, then it suffices to check the following configurations for violations of constraints:



where the first configuration is checked for horizontal violations, the second configuration for vertical violations, and the third configuration for diagonal violations.

In each configuration, all possible assignments of the squares A and B to the locations denoted by "X" are constructed (i.e., 4 arrangements for each of the first two configurations and $2^4 = 16$ for the third configuration).

The two algorithms also present different space complexity concerns. While both algorithms use a stack to perform backtracking so that bad choices of labelings can be corrected, the stack can grow without bound in the Forbidden String Algorithm (or at least up to the area of the bounding array used, which can be very large), whereas in the Rectangle Tiling Algorithm, the stack can never be larger than $2MN$, and if it fills to that value, then the algorithm has successfully completed its task.

A recursive implementation of the Rectangle Tiling Algorithm is given in Appendix B, which may be useful for alternate implementations or a formal correctness proof.

C. Enforcing the d and k Constraints

In Step 3 of the Rectangle Tiling Algorithm, the d constraint is enforced by labeling positions in the arrays A and B zero if they are within distance d of a chosen 1, for any configuration of A and B in a tiling. In Step 4 of the Rectangle Tiling Algorithm, the k constraint is enforced by hunting for violating strings of the form 0^{k+1} in arbitrary configurations of A and B in a tiling. Enforcing the k constraint is considerably more time-consuming that the d constraint, but both processes share the common feature of examining arbitrary tilings by labelings of A and B.

In order to efficiently be able to enforce these two constraints, we use the graphical representation of arbitrary tilings described above. Each of the three graphs G_{hori}, G_{vert}, G_{diag} is used to check all directed paths of length d from the position of the chosen 1, in the horizontal, vertical, and diagonal directions. For any nodes on such paths that are marked as "Unused", a 0 is forced for the corresponding position and array. Then the whole process is repeated for the reverse-direction-edge versions of the three graphs.

For example, if A and B are 5×5 squares and $(d, k) = (1, 5)$, Figure 14 shows in the far left all the forced 0s in both A and B due to the assumptions that $A(1, 1) = 1$ and $B(1, 1) = 0$.

Let us denote certain data structures by $A_{i,j}$ and $B_{i,j}$, corresponding to each position (i, j) of $M \times N$ arrays A and B, respectively. These data structures contain the binary value labeling the position, or else an indication that it is "unused". We define the following three directed graphs to enable a reduced complexity search for potential violations of the k constraint when the arrays A and B are partially labeled:

$$G_{\text{hori}} = (V, E_{\text{hori}})$$
$$G_{\text{vert}} = (V, E_{\text{vert}})$$
$$G_{\text{diag}} = (V, E_{\text{diag}}).$$

The vertex set V and directed edge sets E_{hori}, E_{vert}, E_{diag} are given by:

$$V = \bigcup_{i=1}^{M} \bigcup_{j=1}^{N} \{A_{i,j}, B_{i,j}\}$$

$$E_{\text{hori}} = \bigcup_{i=1}^{M} \Bigg(\bigcup_{j=1}^{N-1} \{(A_{i,j}, A_{i,j+1}), (B_{i,j}, B_{i,j+1})\} \cup \{(A_{i,N}, A_{i,1}), (A_{i,N}, B_{i,1}), (B_{i,N}, B_{i,1}), (B_{i,N}, A_{i,1})\} \Bigg)$$

$$E_{\text{vert}} = \bigcup_{j=1}^{N} \Bigg(\bigcup_{i=1}^{M-1} \{(A_{i,j}, A_{i+1,j}), (B_{i,j}, B_{i+1,j})\} \cup \{(A_{M,j}, A_{1,j}), (A_{M,j}, B_{1,j}), (B_{M,j}, B_{1,j}), (B_{M,j}, A_{1,j})\} \Bigg)$$

$$\begin{aligned} E_{\text{diag}} = & \bigcup_{i=2}^{M} \bigcup_{j=1}^{N-1} \{(A_{i,j}, A_{i-1,j+1}), (B_{i,j}, B_{i-1,j+1})\} \\ & \cup \bigcup_{i=2}^{M} \{(A_{i,N}, A_{i-1,1}), (A_{i,N}, B_{i-1,1}), (B_{i,N}, B_{i-1,1}), (B_{i,N}, A_{i-1,1})\} \\ & \cup \bigcup_{j=1}^{N-1} \{(A_{1,j}, A_{M,j+1}), (A_{1,j}, B_{M,j+1}), (B_{1,j}, B_{M,j+1}), (B_{1,j}, A_{M,j+1})\} \\ & \cup \{(A_{1,N}, A_{M,1}), (A_{1,N}, B_{M,1}), (B_{1,N}, A_{M,1}), (B_{1,N}, B_{M,1})\}. \end{aligned}$$

For the horizontal graph G_{hori}, all of the nodes corresponding to positions of A and B, except for positions in the far right column, have exactly one out-edge, namely to the next position to the right in the same array. Each of the positions in the far right column has two out-edges, pointing to the positions in A and B of the same row, but in the far left column.

For the vertical graph G_{vert}, all of the nodes corresponding to positions of A and B, except for positions in the bottom row, have exactly one out-edge, namely to the next position down in the same array. Each of the positions in the bottom row has two out-edges, pointing to the positions in A and B of the same column, but in the top row.

For the diagonal graph G_{diag}, all of the nodes corresponding to positions of A and B, except for positions in the top row and right column, have exactly one out-edge, namely to the next position to the right and up in the same array. The top row and right column positions have two out-edges each, directed to the similarly diagonally-adjacent positions in A and B.

These three graphs facilitate searching for the string 0^{k+1} horizontally, vertically, or northeast diagonally. Any horizontal (respectively, vertical or diagonal) run of 0s of length z corresponds to a path[5] of length z through the graph G_{hori} (respectively, G_{vert} or G_{diag}).

One way to search for any possible occurrence of 0^{k+1} is to assume such a run starts at a particular location (x, y) in either A or B, and then search for a path of length $k+1$, labeled by 0s, in either G_{hori}, G_{vert}, or G_{diag}. This, however, repeats work unnecessarily, since adjacent starting positions may share substantial portions of runs.

A more efficient technique, which we use, is to first find, starting at each position (x, y) of A, the longest path in G_{hori} labeled entirely by 0s and the longest path in the reversed-edge-direction version of G_{hori} labeled entirely by 0s. This will yield the longest horizontal run of 0s in A that passes through the position (x, y). This process is then repeated for each position of B. With such a technique, duplicated work is avoided and the longest runs of 0s passing through each position of A and B are determined. If any of these runs have length greater than k, then the process can be terminated immediately and a horizontal violation of the k constraint can be declared. If no such violation is discovered, this process is repeated for the graph G_{vert}, and then again for G_{diag}. If no run of k zeros is found in any of the three graphs, then the current labeling is declared to not (yet) violate the k constraint.

D. Example for $d = 1$ and $k = 5$

We illustrate in Figure 14 the Rectangle Tiling Algorithm with the hexagonal $(1, 5)$ constraint, by constructing two 5×5 distinct labelings that satisfy the constraint no matter how they tile the plane. Thus $C_{\text{hex}}(1, 5) > 1/25$. The stack is shown at 6 different stages, after choosing 1s and after popping due to violations of the k constraint. Below each stack is the current labeling of two squares A and B, with unlabeled portions left blank.

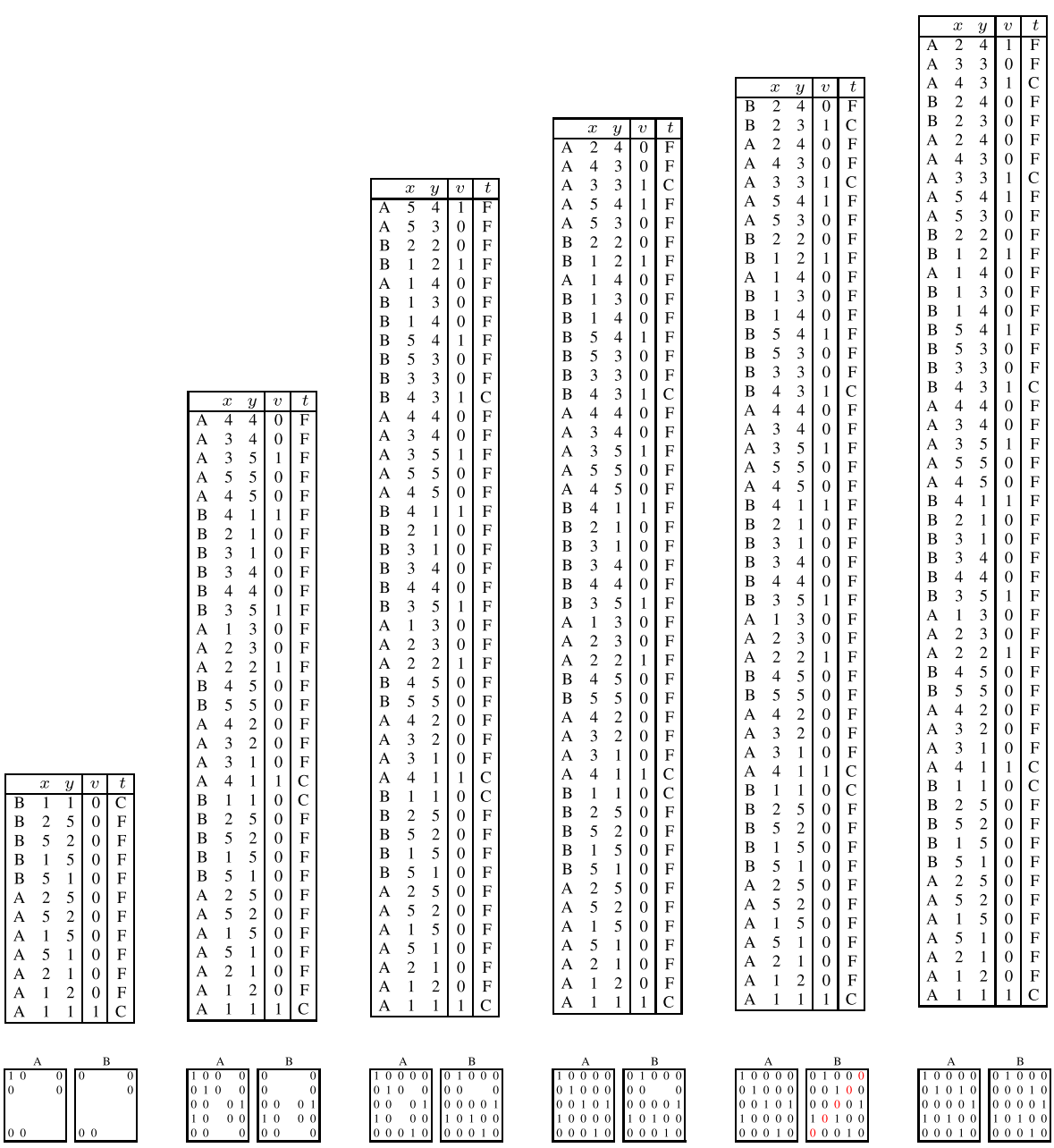
In the first stack snapshot shown in Figure 14, the original two chosen values are shown (a 1 and a 0), and the resulting forced 0s from the chosen 1. The stack grows upward, and each line stores the coordinates of the labeled point, the value of the label, which of the two arrays (A or B) it lies in, and whether the value was chosen or forced. It can be seen that in the fifth snapshot of the stack, a violation of the k constraint is discovered (in red) as the string 0^5 on the main diagonal of array B. The reason this is a violation is that no row, column, or main northeast diagonal can be all 0s, for otherwise, a tiling in the plane of two such squares in a diagonal configuration will result in a diagonal string 0^{10}. Also, note that in the fourth stack snapshot, it is possible to discover that $B(2, 4) = 1$ is forced, in order to prevent the k-violation described in the fifth stack snapshot. However, not all such forced values are found in practice, and implementing such procedures leads to a tradeoff between the time spent hunting for such forced values and the benefit of finding them sooner, rather than later.

E. Algorithmic Positive Capacity Results

We include here a proof of five positive hexagonal (d, k) capacities. The first one, $C_{\text{hex}}(0, 1) > 0$, is rather trivial but is included for completeness since the only other known proof, due to Baxter and Joyce in more general form as discussed earlier, is extremely complicated. The other four cases have been stated previously but never proven in the literature.

In each case we demonstrate positive capacity by exhibiting a pair of distinct square labelings which can arbitrarily tile the plane without violating the corresponding constraint. Thus

[5]We assume the vertices in any path are distinct.



	x	y	v	t
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

	x	y	v	t
A	4	4	0	F
A	3	4	0	F
A	3	5	1	F
A	5	5	0	F
A	4	5	0	F
B	4	1	1	F
B	2	1	0	F
B	3	1	0	F
B	3	4	0	F
B	4	4	0	F
B	3	5	1	F
A	1	3	0	F
A	2	3	0	F
A	2	2	1	F
B	4	5	0	F
B	5	5	0	F
A	4	2	0	F
A	3	2	0	F
A	3	1	0	F
A	4	1	1	C
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

	x	y	v	t
A	5	4	1	F
A	5	3	0	F
B	2	2	0	F
B	1	2	1	F
A	1	4	0	F
B	1	3	0	F
B	1	4	0	F
B	5	4	1	F
B	5	3	0	F
B	3	3	0	F
B	4	3	1	C
A	4	4	0	F
A	3	4	0	F
A	3	5	1	F
A	5	5	0	F
A	4	5	0	F
B	4	1	1	F
B	2	1	0	F
B	3	1	0	F
B	3	4	0	F
B	4	4	0	F
B	3	5	1	F
A	1	3	0	F
A	2	3	0	F
A	2	2	1	F
B	4	5	0	F
B	5	5	0	F
A	4	2	0	F
A	3	2	0	F
A	3	1	0	F
A	4	1	1	C
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

	x	y	v	t
A	2	4	0	F
A	4	3	0	F
A	3	3	1	C
A	5	4	1	F
A	5	3	0	F
B	2	2	0	F
B	1	2	1	F
A	1	4	0	F
B	1	3	0	F
B	1	4	0	F
B	5	4	1	F
B	5	3	0	F
B	3	3	0	F
B	4	3	1	C
A	4	4	0	F
A	3	4	0	F
A	3	5	1	F
A	5	5	0	F
A	4	5	0	F
B	4	1	1	F
B	2	1	0	F
B	3	1	0	F
B	3	4	0	F
B	4	4	0	F
B	3	5	1	F
A	1	3	0	F
A	2	3	0	F
A	2	2	1	F
B	4	5	0	F
B	5	5	0	F
A	4	2	0	F
A	3	2	0	F
A	3	1	0	F
A	4	1	1	C
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

	x	y	v	t
B	2	4	0	F
B	2	3	1	C
A	2	4	0	F
A	4	3	0	F
A	3	3	1	C
A	5	4	1	F
A	5	3	0	F
B	2	2	0	F
B	1	2	1	F
A	1	4	0	F
B	1	3	0	F
B	1	4	0	F
B	5	4	1	F
B	5	3	0	F
B	3	3	0	F
B	4	3	1	C
A	4	4	0	F
A	3	4	0	F
A	3	5	1	F
A	5	5	0	F
A	4	5	0	F
B	4	1	1	F
B	2	1	0	F
B	3	1	0	F
B	3	4	0	F
B	4	4	0	F
B	3	5	1	F
A	1	3	0	F
A	2	3	0	F
A	2	2	1	F
B	4	5	0	F
B	5	5	0	F
A	4	2	0	F
A	3	2	0	F
A	3	1	0	F
A	4	1	1	C
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

	x	y	v	t
A	2	4	1	F
A	3	3	0	F
A	4	3	1	C
B	2	4	0	F
B	2	3	0	F
A	2	4	0	F
A	4	3	0	F
A	3	3	1	C
A	5	4	1	F
A	5	3	0	F
B	2	2	0	F
B	1	2	1	F
A	1	4	0	F
B	1	3	0	F
B	1	4	0	F
B	5	4	1	F
B	5	3	0	F
B	3	3	0	F
B	4	3	1	C
A	4	4	0	F
A	3	4	0	F
A	3	5	1	F
A	5	5	0	F
A	4	5	0	F
B	4	1	1	F
B	2	1	0	F
B	3	1	0	F
B	3	4	0	F
B	4	4	0	F
B	3	5	1	F
A	1	3	0	F
A	2	3	0	F
A	2	2	1	F
B	4	5	0	F
B	5	5	0	F
A	4	2	0	F
A	3	2	0	F
A	3	1	0	F
A	4	1	1	C
B	1	1	0	C
B	2	5	0	F
B	5	2	0	F
B	1	5	0	F
B	5	1	0	F
A	2	5	0	F
A	5	2	0	F
A	1	5	0	F
A	5	1	0	F
A	2	1	0	F
A	1	2	0	F
A	1	1	1	C

Fig. 14. Snapshots of the stack during the Rectangle Tiling Algorithm. The stack grows upward, and each line in the stack shows which square (A or B) is modified, the location (x, y) of modification, the binary value v, and the type of change t ("C" for Chosen, or "F" for Forced). The coordinates are given in matrix notation, namely (row, column), where the upper-left corner is position $(1, 1)$.

each square labeling carries with it one bit of information content.

To find these tileable square labelings, we wrote a backtracking algorithm that starts with two blank squares and attempts to fill in 1s in randomly chosen unoccupied positions, along with any 0s that are then forced due to the d constraint. If a contradiction results from trying to place a 0 where a 1 already exists, or vice versa, then the previous action is popped off of a stack, and new moves are attempted. Successive pushing and popping of the stack eventually led to the results shown. Computer run times were typically on the order of several minutes.

We note that it is an open question whether pairs of labelings exist that tile the plane validly whenever a capacity is positive. Alternatively, it is conceivable that only aperiodic tilings of the plane can demonstrate positive capacity. In fact, Durand, Gamard, Grandjean [8] demonstrated the existence of a certain Wang tile set that can only tile the plane aperiodically, and yet achieves a positive capacity, although their results do not apply directly to the hexagonal (d, k) constraint situation.

Theorem IV.1: If $(d, k) \in \{(0, 1), (1, 4), (2, 5), (3, 7), (4, 9)\}$, then the hexagonal (d, k) capacity is positive.

Proof: For each constraint, a square is given that can take on two distinct labelings by choosing the value of the indicated

1	x
1	1

Fig. 15. Two distinct labelings of a 2×2 square for the hexagonal $(0, 1)$ constraint where $x \in \{0, 1\}$. Thus, $C_{\text{hex}}(0, 1) \geq 1/4$.

0	1	0	0	1	0	0	x
0	0	1	0	0	1	0	$\overline{x}$
1	0	0	1	0	0	x	0
0	1	0	0	1	0	$\overline{x}$	0
0	0	1	0	0	x	0	1
1	0	0	1	0	0	0	0
0	1	0	0	0	1	0	0
0	0	0	1	0	0	1	0

Fig. 16. Two distinct labelings of an 8×8 square for the hexagonal $(1, 4)$ constraint where $x \in \{0, 1\}$. Thus, $C_{\text{hex}}(1, 4) \geq 1/64$.

0	0	1	0	0	x
1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1
1	0	0	1	0	0
0	1	0	0	1	0

Fig. 17. Two distinct labelings of a 6×6 square for the hexagonal $(2, 5)$ constraint where $x \in \{0, 1\}$. Thus, $C_{\text{hex}}(2, 5) \geq 1/36$.

0	0	0	1	0	0	0	x
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	1
0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0

Fig. 18. Two distinct labelings of an 8×8 square for the hexagonal $(3, 7)$ constraint where $x \in \{0, 1\}$. Thus, $C_{\text{hex}}(3, 7) \geq 1/64$.

x to be either 0 or 1, where $\overline{x} = 1 - x$ (see Figures 15 – 19). One can verify by inspection that the resulting two labelings for each constraint can be arbitrarily assigned to squares in a tiling of the plane by the squares, without violating the relevant hexagonal (d, k) constraint. Thus the area occupied by one such square can have any one of at least two possible labelings, so the capacity is lower bounded by the reciprocal of the area of the square, which in particular is positive. ■

APPENDIX A
RECURSIVE IMPLEMENTATION OF THE FORBIDDEN STRING ALGORITHM

```
int A[N][N];
int Origin = N/2;

for(i,j=1 to N) A[i][j] = Unused;

Stack<Cell> S;

struct Cell {
  int X, Y, value;
  string type;
}

bool Assign( X, Y ) {
  A[X][Y] = 1;

  Cell c;
  c.X = X; c.Y = Y;
  c.type = "assumed";
  c.value = 1;
  S.push( c );

  ForceZeros();

  if( No_k_Violation() )
    if( NoUnusedLocations() )
      return false; // Ran out of memory.

  else {
    do{ A[S.top.X][S.top.Y]=Unused; S.pop()}
    while( !(S.top.type = "assumed"
             && S.top.value == 1 ));

    if(S.top.X == Origin
             && S.top.Y == Origin)
      return true; // Found proof 10^z1 forbidden.
    else {
      A[S.top.X][S.top.Y] = 0;
      S.top.value = 0;
      S.top.type = "forced";
    }
  }

  (X,Y) = NewUnusedLocation();
  return Assign( X, Y ); // Assume another 1.
}

A( Origin + z + 1, Origin ) = 1;

if( Assign( Origin, Origin ) ) print "Success";
else print "Failure";
```

0	0	0	0	1	0	0	0	0	x
0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	0

Fig. 19. Two distinct labelings of a 10×10 squares for the hexagonal $(4, 9)$ constraint where $x \in \{0, 1\}$. Thus, $C_{\text{hex}}(4, 9) \geq 1/100$.

APPENDIX B
RECURSIVE IMPLEMENTATION OF THE RECTANGLE TILING ALGORITHM

```
int A[N][M], B[N][M];
for(i=1 to N, j=1 to M) A[i][j] = B[i][j] = Unused;

Stack<Cell> S;

struct Cell {
  int X, Y, value;
  string type;
  int** Rect;
}

bool Tile( WhichRectangle, X, Y ) {
  WhichRectangle[X][Y] = 1;

  Cell c;
  c.X = X; c.Y = Y;
```

```
  c.Rect = WhichRectangle;
  c.type = "chosen";
  c.value = 1;
  S.push( c );

  ForceZeros();

  if( No_k_Violation() )
    if( RectanglesFull() )
      return true; // Found rectangles.

  else {
    do{S.top.Rect[S.top.X][S.top.Y]=Unused; S.pop()}
    while( !(S.top.type == "chosen"
          && S.top.value == 1 ));

    if(S.top.X == 1 && S.top.Y == 1
                          && S.top.Rect == A )
      return false;
        // Original assumptions led to contradiction.
    else {
      S.top.Rect[S.top.X][S.top.Y]=0;
      S.top.value = 0;
      S.top.type = "forced";
    }
  }

  Rect = NewUnfilledRectangle;
  (X,Y) = NewUnusedLocation( Rect );
  // Try another~1~in one rectangle\ldots
  return Tile( Rect, X, Y );
}

B(1,1) = 0;

if( Tile( A, 1, 1) ) then print "Success";
else print "Failure";
```

ACKNOWLEDGMENT

The authors would like thank David Berard for valuable computer simulations and discussions.

REFERENCES

[1] R. J. Baxter, "Hard hexagons: Exact solution," *J. Phys. A, Math. Gen.*, vol. 13, pp. 1023–1030, Mar. 1980.
[2] R. J. Baxter, *Exactly Solved Models in Statistical Mechanics*. New York, NY, USA: Academic, 1982.
[3] R. J. Baxter, "Planar lattice gases with nearest-neighbour exclusion," *Ann. Combinatorics*, vol. 191, pp. 191–203, Jun. 1999.
[4] R. J. Baxter and S. K. Tsang, "Entropy of hard hexagons," *J. Phys. A, Math. Gen.*, vol. 13, no. 3, pp. 1023–1030, Mar. 1980.
[5] N. J. Calkin and H. S. Wilf, "The number of independent sets in a grid graph," *SIAM J. Discrete Math.*, vol. 11, no. 1, pp. 54–60, Feb. 1998.
[6] K. Censor and T. Etzion, "The positive capacity region of two-dimensional run-length-constrained channels," *IEEE Trans. Inf. Theory*, vol. 52, no. 11, pp. 5128–5140, Nov. 2006.
[7] S. Congero and K. Zeger, "Hexagonal run-length zero capacity region—Part I: Analytical proofs," *IEEE Trans. Inf. Theory*, vol. 68, no. 1, pp. 130–152, Jan. 2021.
[8] B. Durand, G. Gamard, and A. Grandjean, "Aperiodic tilings and entropy," in *Proc. Int. Conf. Develop. Lang. Theory*, in Lecture Notes in Computer Science, vol. 8633. Ekaterinburg, Russia: Springer, 2014, pp. 166–177.
[9] K. S. Immink, *Codes for Mass Data Storage Systems*, 2nd ed. Eindhoven, The Netherlands: Shannon Foundation Publishers, 2004.
[10] G. S. Joyce, "Exact results for the activity and isothermal compressibility of the hard-hexagon model," *J. Phys. A, Math. Gen.*, vol. 21, no. 20, pp. L983–L988, Oct. 1988.
[11] G. S. Joyce, "On the hard hexagon model and the theory of modular functions," *Phil. Trans. Roy. Soc. London A*, vol. 325, pp. 643–702, Aug. 1988.
[12] A. Kato and K. Zeger, "On the capacity of two-dimensional run-length constrained channels," *IEEE Trans. Inf. Theory*, vol. 45, no. 5, pp. 1527–1540, Jul. 1999.
[13] Zs. Kukorelly and K. Zeger, "The capacity of some hexagonal (d,k)-constraints," in *Proc. IEEE Int. Symp. Inf. Theory*, Washington, DC, USA, Jun. 2001, p. 64.
[14] Zs. Kukorelly and K. Zeger, "Automated theorem proving for hexagonal run length constrained capacity computation," in *Proc. IEEE Int. Symp. Inf. Theory*, Seattle, WA, USA, Jul. 2006, pp. 1199–1203.
[15] B. D. Metcalf and C. P. Yang, "Degeneracy of antiferromagnetic ising lattices at critical magnetic field and zero temperature," *Phys. Rev. B, Condens. Matter*, vol. 18, no. 5, pp. 2304–2307, Sep. 1978.
[16] Zs. Nagy and K. Zeger, "Capacity bounds for the three-dimensional run length limited channel," *IEEE Trans. Inf. Theory*, vol. 46, no. 3, pp. 1030–1033, May 2000.
[17] L. Onsager, "Crystal statistics. I. A two-dimensional model with an order-disorder transition," *Phys. Rev.*, vol. 65, pp. 117–149, Feb. 1944.
[18] R. Pavlov, "Approximating the hard square entropy constant with probabilistic methods," *Ann. Probab.*, vol. 40, no. 6, pp. 2362–2399, Nov. 2012.
[19] M. Schwartz and A. Vardy, "New bounds on the capacity of multidimensional run-length constraints," *IEEE Trans. Inf. Theory*, vol. 57, no. 7, pp. 4373–4382, Jul. 2011.
[20] A. Sharov and R. M. Roth, "Two-dimensional constrained coding based on tiling," *IEEE Trans. Inf. Theory*, vol. 56, no. 4, pp. 1800–1807, Apr. 2010.
[21] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.
[22] G. H. Wannier, "Antiferromagnetism. The triangular ising net," *Phys. Rev.*, vol. 79, no. 2, pp. 357–364, Jul. 1950.
[23] W. Weeks and R. E. Blahut, "The capacity and coding gain of certain checkerboard codes," *IEEE Trans. Inf. Theory*, vol. 44, no. 3, pp. 1193–1203, May 1998.

Spencer Congero (Student Member, IEEE) was born in Hartford, CT, USA, in 1994. He received the bachelor's degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 2016. He is currently pursuing the Ph.D. degree with the University of California, San Diego.

Kenneth Zeger was born in Boston, MA, USA, in 1963. He received the S.B. and S.M. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1984, and the M.A. degree in mathematics and the Ph.D. degree in electrical engineering from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 1989 and 1990, respectively. He was an Assistant Professor of electrical engineering at the University of Hawaii from 1990 to 1992. He was with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana–Champaign, as an Assistant Professor from 1992 to 1995, and an Associate Professor from 1995 to 1996. From 1996 to 1998, he has been with the Department of Electrical and Computer Engineering, University of California, San Diego, as an Associate Professor. Since 1998, he has been a Professor with the Department of Electrical and Computer Engineering, University of California, San Diego. He received an NSF Presidential Young Investigator Award in 1991. He served as an Associate Editor At Large for the IEEE TRANSACTIONS ON INFORMATION THEORY from 1995 to 1998 and a member of the Board of Governors of the IEEE Information Theory Society from 1998 to 2000, from 2005 to 2007, and from 2008 to 2010.