Algorithmica (2018) 80:136–154
https://doi.org/10.1007/s00453-016-0247-3

3D Rectangulations and Geometric Matrix Multiplication

Peter Floderus[1] · Jesper Jansson[2] · Christos Levcopoulos[3] · Andrzej Lingas[3] · Dzmitry Sledneu[1]

Received: 14 September 2015 / Accepted: 5 November 2016 / Published online: 16 November 2016

Abstract The problem of partitioning an orthogonal polyhedron P into a minimum number of 3D rectangles is known to be NP-hard. In this paper, we first develop a 4-approximation algorithm for the special case of the problem in which P is a *3D histogram*. It runs in $O(m \log m)$ time, where m is the number of corners in P. We then apply it to exactly compute the arithmetic matrix product of two $n \times n$ matrices A and B with nonnegative integer entries, yielding a method for computing $A \times B$ in $\tilde{O}(n^2 + \min\{r_A r_B,\ n \min\{r_A,\ r_B\}\})$ time, where $\tilde{O}$ suppresses polylogarithmic (in n) factors and where r_A and r_B denote the minimum number of 3D rectangles into which the 3D histograms induced by A and B can be partitioned, respectively.

An extended abstract of this article appeared in *Proceedings of the 25th International Symposium on Algorithms and Computation* (ISAAC 2014), volume 8889 of *Lecture Notes in Computer Science*, pp. 65–78, Springer International Publishing Switzerland, 2014.

✉ Jesper Jansson
jj@kuicr.kyoto-u.ac.jp

Peter Floderus
pflo@maths.lth.se

Christos Levcopoulos
Christos.Levcopoulos@cs.lth.se

Andrzej Lingas
Andrzej.Lingas@cs.lth.se

Dzmitry Sledneu
Dzmitry@maths.lth.se

1 Centre for Mathematical Sciences, Lund University, 22100 Lund, Sweden

2 Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan

3 Department of Computer Science, Lund University, 22100 Lund, Sweden

Keywords Decomposition problem · Minimum rectangulation · Orthogonal polyhedron · Matrix multiplication · Time complexity

1 Introduction

This paper considers two intriguing and at a first glance unrelated problems.

The first problem lies at the heart of three-dimensional computational geometry. It belongs to the class of *polyhedron decomposition* problems, whose applications range from data compression and database systems to pattern recognition, image processing, and computer graphics [7,14]. The problem is to partition an input orthogonal polyhedron into a minimum number of 3D rectangles. Dielissen and Kaldewaij [4] have shown this problem to be NP-hard. (Formally, the NP-hardness proof by [4] is for polyhedra with holes, but the authors remark that the proof should also work for simple polyhedra.) To the best of our knowledge, no non-trivial approximation factors for minimum rectangular partitions of simple orthogonal polyhedra are known, even in restricted non-trivial cases such as that of a 3D histogram, a straightforward generalization of a planar histogram; see Sect. 2 below for the definition. In contrast, the problem of partitioning an orthogonal (planar) polygonal region into a minimum number of 2D rectangles admits a polynomial-time solution [7,11].

The second problem we consider is that of multiplying two $n \times n$ matrices. There exist fast algorithms that do so in substantially subcubic time, e.g., a recent one due to Le Gall runs in $O(n^{2.3728639})$ time [8], but they suffer from very large overheads. On the positive side, input matrices in real world applications often belong to quite restricted matrix classes, so a natural approach is to design faster algorithms for such special cases. Indeed, efficient algorithms for *sparse* matrix multiplication have been known for long time. In the Boolean case, despite considerable efforts by the algorithms community, the fastest known combinatorial algorithms for Boolean $n \times n$ matrix multiplication barely run in subcubic time (in $O(n^3(\log\log n)^2/(\log n)^{9/4})$ time [1], to be precise), but much faster algorithms for Boolean matrix product for restricted classes of Boolean matrices have been developed [2,5,9]. For example, when at least one of the input Boolean matrices admits an exact covering of its ones by a relatively small number of rectangular submatrices, the Boolean matrix product can be computed efficiently [9]. Similarly, if the rows of the first input Boolean matrix or the columns of the second input Boolean matrix can be represented by a relatively cheap minimum cost spanning tree in the Hamming metric (or its generalization to include blocks of zeros or ones) then the Boolean matrix product can be computed efficiently by a randomized combinatorial algorithm [2,5].

1.1 New Results

Our first contribution is an $O(m \log m)$-time, 4-approximation algorithm for computing a minimum 3D rectangular partition of an input 3D histogram with m corners. It works by projecting the input histogram onto the base plane, partitioning the resulting planar straight-line graph into a number of 2D rectangles not exceeding its number of vertices, and transforming the resulting 2D rectangles into 3D rectangles of appropri-

ate height. Importantly, the known algorithms for minimum partition of an orthogonal polygon with holes into 2D rectangles [7,11] do not yield the aforementioned upper bound on the number of rectangles in the more general case of planar straight-line graphs.

Our second contribution is a new technique for multiplying two matrices with nonnegative integer entries. We interpret the matrices as 3D histograms and decompose them into blocks that can be efficiently manipulated in a pairwise manner using the interval tree data structure. Let A and B be two $n \times n$ matrices with nonnegative integer entries, and let r_A and r_B denote the minimum number of 3D rectangles into which the 3D histograms induced by A and B can be partitioned. By applying our 4-approximation algorithm above, we can compute $A \times B$ in $\tilde{O}(n^2 + r_A r_B)$ time, where $\tilde{O}$ suppresses polylogarithmic (in n) factors. Next, by using another idea of slicing the histogram of A (or B) into parts corresponding to rows of A (or columns of B) and measuring the cost of transforming a slice into a consecutive one, we obtain an upper bound of $\tilde{O}(n^2 + n \min\{r_A, r_B\})$. We also give a generalization of the latter upper bound in terms of the minimum cost of a spanning tree for the slices, where the distance between a pair of slices corresponds to the cost of transforming one slice into the other.

We remark that $r_A = O(n^2)$ and $r_B = O(n^2)$ always hold. For inputs where $r_A r_B = \tilde{O}(n^2)$, the worst-case running time of our first algorithm for matrix multiplication is $\tilde{O}(n^2)$, which is almost optimal and much better than that of the currently fastest one for the general case [8]. Furthermore, when at least one of r_A and r_B is $\tilde{O}(n)$, the worst-case running times of our second and third algorithms are almost optimal.

1.2 Organization of the Paper

Section 2 presents our 4-approximation algorithm for a partition of a 3D histogram into a minimum number of 3D rectangles. Section 3 presents our algorithms for the arithmetic matrix product. Section 4 concludes with some final remarks.

2 Rectangular Partitions of 3D Histograms

A 2D histogram is a polygon with an edge e, which we call the *base* of the histogram, having the following property: for every point p in the interior of histogram, there is a (unique) line segment perpendicular to e, connecting p to e and lying totally in the interior of the histogram. In this paper, we consider orthogonal histograms only. For simplicity, we consider the base of a histogram as being horizontal, and all other edges of the histogram lying above the base. In this way, a 2D histogram can also be thought of as the union of rectangles standing on the base of the histogram.

A *3D histogram* is a natural generalization of a 2D histogram. To define a 3D histogram, we need the concept of the "base plane", which for simplicity we define as the horizontal plane containing two of the axes in the Euclidean space. A 3D histogram can then be thought of as the union of (internally disjoint) orthogonal 3D rectangles, standing on the base plane. The *base of the histogram* is the union of the lower faces (also called *bases*) of all these rectangles.

Definition 2.1 A 3D histogram is a union of a finite set C of orthogonal 3D rectangles such that: (i) each element in C has a face on the horizontal base plane; and (ii) all elements in C are located above the base plane.

(In the literature, what we call a 3D histogram is sometimes termed a 2D histogram or a 1D histogram when used to summarize 2D or 1D data, respectively [13].)

By a *rectangular partition* (or *rectangulation*) of a 3D histogram P, we mean an orthogonal partition of P into 3D rectangles. In Sect. 2.2 below, we consider the problem of finding a rectangular partition of a given 3D histogram P into as few 3D rectangles as possible. We present a 4-approximation algorithm for this problem with time complexity $O(m \log m)$, where m denotes the number of vertices in P. The algorithm partitions P into less than m' 3D rectangles, where m' is the number of vertices in the vertical projection of P (i.e., $m' < m$), by applying a subroutine described in Sect. 2.1 that partitions any orthogonal planar straight-line graph (PSLG) with m' vertices into less than m' 2D rectangles.

2.1 Partitioning an Orthogonal PSLG into 2D Rectangles

The problem of partitioning an orthogonal polygon in two dimensions into as few rectangles as possible has been well studied in the literature [7,11], and polynomial-time algorithms for this problem exist (see [7,11] and the references therein). However, for our purposes, it is not necessary to compute an *optimal* solution for the 2-dimensional problem. Instead, we just need a partition of a planar straight-line graph (PSLG) into less than m' rectangles, where m' denotes the number of vertices in the input PSLG. We will show that a simple algorithm, which is faster than the optimal algorithms in [7,11], suffices to obtain this (not always optimal) upper bound.

Since this subsection considers 2D only, we use the term "horizontal" for line segments parallel to the X-axis. By "vertical" lines, we mean lines or line segments parallel to the Y-axis. Each vertex in the planar graphs in our application has degree 2, 3, or 4.

Definition 2.2 A planar straight-line graph (PSLG) $PG = (V, E)$, as used in this paper, is a planar graph where every vertex has an x- and a y-coordinate. Each edge is drawn as a straight line segment, all edges meet at right angles, and each vertex has degree 2, 3, or 4. A *rectangular partition* of PG is a partition $R = (V \cup V_R, E \cup E_R)$ that adds edges and vertices to PG so that R is still a PSLG while every face in R is a rectangle.

Given a PSLG PG, we denote $m' = |V|$. We say that a vertex v of PG is *concave* if it has degree 2, its two adjacent edges are perpendicular to each other, and the corner at v which is of 270 degrees does not lie in the outer, infinite face of PG. Any vertex which is not concave is called *convex*.

We use a sweep line approach to generate a partition into less than m' rectangles. We perform a horizontal sweep with a vertical sweep line [3], using the vertices of PG as event points. Whenever the sweep line reaches a concave vertex v, we insert into the graph PG a vertical line segment s connecting v to the closest edge of PSLG upwards or downwards, thus canceling the concavity at v and transforming v into a

convex vertex of degree 3. Hence, if there was already an edge of PG below v, then the new segment s is inserted above v, otherwise it is inserted below v. To preserve the property that the resulting graph is still a PSLG, the other endpoint of s may have to become a new vertex of the PSLG. (This is a standard procedure for trapezoidation; see, e.g., [3] for more details.) After the sweep is complete, all concave vertices have been eliminated. (It may happen that two concave vertices with the same x-coordinate are connected by a single vertical segment that is disjoint from the rest of the input PSLG. In this case, the plane sweep algorithm will produce this segment. Thus, no two segments produced by the algorithm overlap or touch each other.)

The correctness of the algorithm is easy to see: it eliminates all concave corners of PG by adding vertical line segments. Hence, in the resulting PSLG, each face, except for the outer face, is a rectangle. The running time of this algorithm is dominated by the cost of the plane sweep, which is $O(m' \log m')$ according to well-known methods in computational geometry; see, e.g., [3].

The next lemma relates the resulting number of 2D rectangles to the number of vertices in the input PSLG.

Lemma 2.3 *Any PSLG $PG = (V, E)$ with $|V| = m'$ and minimum vertex degree 2 can be partitioned into b rectangles with $b < m'$ in $O(m' \log m')$ time.*

Proof Let R denote the set of rectangles in the rectangular partition produced by the plane sweep algorithm described above. We use a "charging scheme" to prove the stated inequality. The charging scheme starts by giving each vertex $v \in V$ four tokens; thus, a total of $4m'$ tokens are used. Each vertex v then distributes its tokens in a certain way to the rectangles in R that are adjacent to v. We will show that every rectangle in R receives at least four tokens. Since we started by giving a total of $4m'$ tokens to the vertices, this will prove that there exist at most m' rectangles, and thus $b \leq m'$. Moreover, vertices adjacent to the outer face do not give away more than three tokens. We will thus obtain the strict inequality $b < m'$.

Now we describe the details of the charging scheme. Let v be any vertex of V. The vertex v gives one token to each rectangle r in R which in any way is adjacent to it, with one exception. The exception occurs when v is a concave vertex; then, v is partitioned by a vertical segment e_r added by the algorithm. This segment partitions the three quadrants at the concave corner around the vertex so that one rectangle occupies one

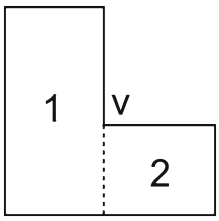

Fig. 1 A concave corner v and a vertical segment (*dashed*) added by the algorithm. One of the rectangles receives two tokens, and the other one receives one token. If there had existed an additional rectangle with v as a corner, it would also have received one token

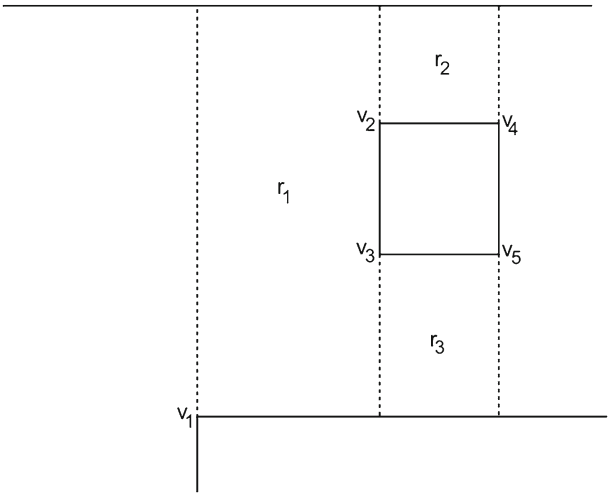

Fig. 2 An example of two rectangles r_2 and r_3 that are only adjacent to two vertices (v_2, v_4 and v_3, v_5, respectively). The *dashed line* segments indicate *vertical line* segments added by the algorithm, while the *full line* segments indicate edges of *PG*. The rectangle r_1 is adjacent to three vertices from *PG*, and will get two tokens from v_1 and one from v_2 and v_3 each. r_2 and r_3 will receive two tokens from each adjacent vertex

quadrant and one occupies the two others. Then v distributes two tokens to the new rectangle occupying only one quadrant, which therefore has a corner at v, and only one token to each one of the other rectangles of R adjacent to v. See Fig. 1; in this example, v distributes only three of its four tokens, because v happens to be adjacent to the outer face.

We now show that each rectangle receives at least four tokens. Let r be any rectangle in R. First note that each vertical segment added by the algorithm has at least one endpoint at a vertex in V. Moreover, for any rectangle r in R, each of the vertical sides of r includes at least one vertex of V. Therefore, each rectangle is adjacent to at least two vertices of V. We distinguish three cases, depending on the number of vertices of V adjacent to r. Observe that the adjacencies are not necessarily at the corners of r.

- Case 1: r is adjacent to at least four vertices of V. Since r will receive at least one token from each of them we are done.
- Case 2: r is adjacent to precisely three vertices of V. Then at one of the vertical sides of r there is only one vertex of V. Moreover, this vertex v must be at a corner of r and fulfills the criteria for giving two tokens to r. The remaining two adjacent vertices of V give at least one token each, so we are done. See Fig. 2.
- Case 3: r is adjacent to precisely two vertices of V. This must mean that both vertical sides of r are segments added by the algorithm, and that one of the endpoints of each of these sides is a vertex of V at a corner of r. This corresponds to the condition for receiving two tokens mentioned earlier. So in total, r receives four tokens from the two corners, and we are done. See Fig. 2. □

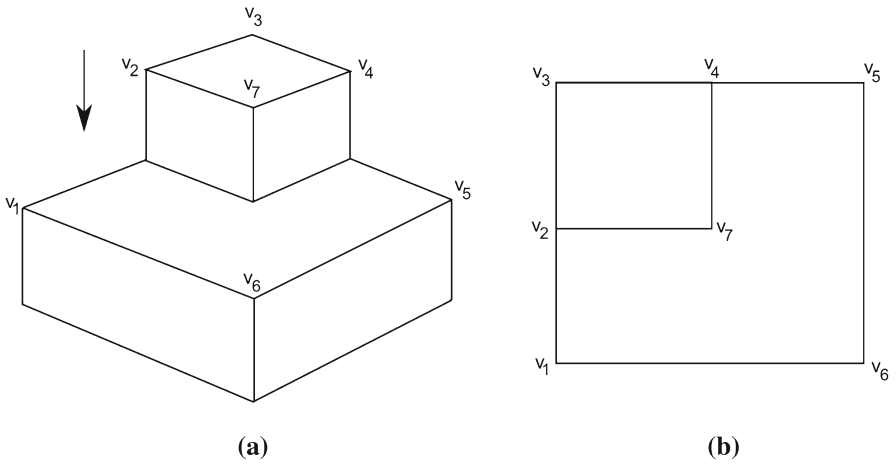

Fig. 3 **a** Displays a 3D histogram and the direction in which we do the projection. **b** Displays the projected figure on the plane with corresponding vertices labeled

2.2 Partitioning a 3D Histogram into 3D Rectangles

We now explain how to construct a projected PSLG *PP* from any input 3D histogram P and how to apply the fast 2D rectangular partition algorithm from Sect. 2.1 to *PP* to obtain a good partition of P into 3D rectangles.

Definition 2.4 The *planar projection PP* is an orthogonal projection of the input 3D histogram P along the "down" direction onto the base plane in Definition 2.1.

See Fig. 3 for an illustration.

We can interpret *PP* as a PSLG where each corner and each subdividing point on a line segment corresponds to a vertex. The edges naturally correlate to the connecting line segments between vertices. Each vertex in *PP* is the vertical projection of at least two vertices of P. Two edges of the 3D histogram may partially overlap in the 2D projection, but the edges in the 2D projection are considered as non-overlapping. Thus, an edge of the 3D histogram may split into several edges in the 2D projection, since vertices should only appear as endpoints of edges.

Every vertex in *PP* must have at least two neighbors. This follows from the fact that each vertex of P (and of any orthogonal polyhedron) has at least two incident horizontal edges, i.e., edges parallel to the base plane. It may happen that some vertex of *PP* is the vertical projection of up to four vertices of P, so those four vertices of P may have a total of eight neighbors in P. But since *PP* is an orthogonal PSLG, no vertex of *PP* has more than four neighbors.

Now we are ready to show the main theorem of this section.

Theorem 2.5 *For any 3D histogram P with m corners, a 4-approximation R of a partition of P into as few 3D rectangles as possible can be computed in $O(m \log m)$ time.*

Proof We let PP be the planar projection of P as in Definition 2.4, assign $PG := PP$, and apply the algorithm from Lemma 2.3 to compute a planar partition R' of PG. The final 3D partition R is obtained from R' by reversing the projection so that each 2D rectangle corresponds to the top of a 3D rectangle in R.

To analyze the approximation factor, denote the number of 3D rectangles in an optimal solution R_{opt} by OPT, the number of 3D rectangles produced by the algorithm described above by b, and the number of vertices in PP by m'. Every vertex of P is adjacent to at least one vertical edge (i.e., edge perpendicular to the base plane) of a 3D rectangle in R_{opt}, which means that every vertex in PP has to be the vertical projection of at least one such vertical edge. Next, every 3D rectangle in R_{opt} has 4 vertical edges, so the total number of vertical edges in R_{opt} (some of which may be projected onto the same vertex in PP) is $4OPT$. Thus, $m' \leq 4OPT$. By Lemma 2.3, we have $b < m'$ and it follows that $b < 4OPT$.

Since the projection can be obtained by contracting each vertex in P and all of its vertical neighbors into one vertex, the projection can be implemented in $O(m)$ time. Thus, the $O(m \log m)$-term from Lemma 2.3 will dominate the time complexity. □

3 Geometric Algorithms for the Arithmetic Matrix Product

In this section, we present our three geometric or in part geometric algorithms for arithmetic matrix product.

3.1 Geometric Data Structures and Notation

Our algorithms for arithmetic matrix multiplication use some data structures for interval and rectangle intersection. An *interval tree* is a search tree that supports intersection queries for a set Q of closed intervals on the real line, i.e., queries asking for reporting the intervals in Q overlapping with the query interval, as follows:

Fact 1 (see Lemmas 2, 3 on pages 193, 195, respectively in [12]). *Suppose that the left endpoints of the intervals in a set Q belong to a subset $\mathcal{U}$ of real numbers of size l and $|Q| = q$. An interval tree T of depth $O(\log l)$ for Q can be constructed in $O(l+q \log lq)$ time using $O(l+q)$ space. The insertion or deletion of an interval with left endpoint in $\mathcal{U}$ into T takes $O(\log l + \log q)$ time. The intersection query asking for listing all stored intervals that overlap with the query interval is supported by T in $O(\log l + r)$ time, where r is the number of reported intervals.*

We also need an efficient data structure based on a segment tree for answering weight queries asking for the total weight of stored weighted intervals containing the query point.

Lemma 3.1 *There is a data structure, a weighted segment tree, that supports the insertions and deletions of weighted intervals with endpoints in $\{0, 1, \ldots, n\}$ in $O(\log n)$ time, and the weight query asking for the total weight of "stored" intervals containing the query point also in $O(\log n)$ time. The data structure can be initialized in $O(n)$ time and it uses $O(n)$ space for a sequence of $n^{O(1)}$ insertions and deletions.*

Proof The data structure is just a modified segment tree T for the elementary segments induced by $\{0, 1, \ldots, n\}$, i.e., $[0, 0]$, $(0, 1)$, $\ldots$, $[n, n]$ (see pp. 212–221 in [12]). The segment tree T is initialized as the standard segment tree for the aforementioned elementary segments. The only difference is that a weight counter set to zero is initialized at each node instead of an initially empty list of intervals. Since T is basically a binary search tree with $O(n)$ nodes, the initialization takes $O(n)$ time.

Now, when an interval is inserted into T then instead of inserting it into the lists of appropriate nodes as in the standard segment tree, its weight is just added to the weight counters at the same nodes. Recall that the appropriate nodes have their interval range included in the inserted interval while their parents do not have this property (see [12]).

Now it is sufficient to insert into T the intervals to be inserted into our data structure with their original weights, and the intervals to be deleted from the data structure with their weights multiplied by -1, in order to answer the weight queries.

The cost of the insertions and the " deletions" is asymptotically the same as that of an interval insertion in the standard segment tree under the assumption that an insertion into a list (in our case updating the weight counter) takes $O(1)$ time (see Theorem 6 on page 214 in [12]). Thus, it is logarithmic in n.

By traversing T with a query point p from the root to a leaf, we just sum the values of weight counters in order to obtain the total weight of intervals containing p, inserted into T. Hence, since the depth of T is $O(\log n)$ (see Theorem 6 on page 214 in [12]), the weight query asking for the total weight of intervals containing p can be answered in $O(\log n)$ time. □

Finally, we shall use the following data structure, easily obtained by computing all prefix sums:

Fact 2 *For a sequence of integers $a_1, a_2, \ldots, a_n$, one can construct a data structure that supports a query asking for reporting the sum $\sum_{k=i}^{j} a_k$ for $1 \leq i \leq j \leq n$ in $O(1)$ time. The construction takes $O(n)$ time.*

In the rest of the paper, A and B denote two $n \times n$ matrices with nonnegative integer entries, and C stands for their matrix product $A \times B$. We also need the following concepts:

- For any $n \times n$ matrix D with nonnegative integer entries, consider the $[0, n] \times [0, n]$ integer grid whose unit cells are in one-to-one correspondence with the entries of D. More precisely, the grid cell between the horizontal lines $i - 1$ and i (counting from the top) and vertical lines $j - 1$ and j (counting from the left) corresponds to $D_{i,j}$ (see Fig. 4a). Then, $his(D)$ stands for the 3D histogram whose base consists of all unit cells of the $[0, n] \times [0, n]$ integer grid corresponding to positive entries of D and whose height over the cell corresponding to $D_{i,j}$ is the value of $D_{i,j}$ (see Fig. 4b).
- For any $n \times n$ matrix D, nonnegative integers $1 \leq i_1 \leq i_2 \leq n$, $1 \leq k_1 \leq k_2 \leq n$, and h_1, h_2, where $h_1 < h_2 \leq D_{i,j}$ for $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$, $rec_D(i_1, i_2, k_1, k_2, h_1, h_2)$ is the 3D rectangle with the corners $(i_1 - 1, k_1 - 1, h_l)$, $(i_1 - 1, k_2, h_l)$, $(i_2, k_1 - 1, h_l)$, (i_2, k_2, h_l), where $l = 1, 2$, lying within $his(D)$.

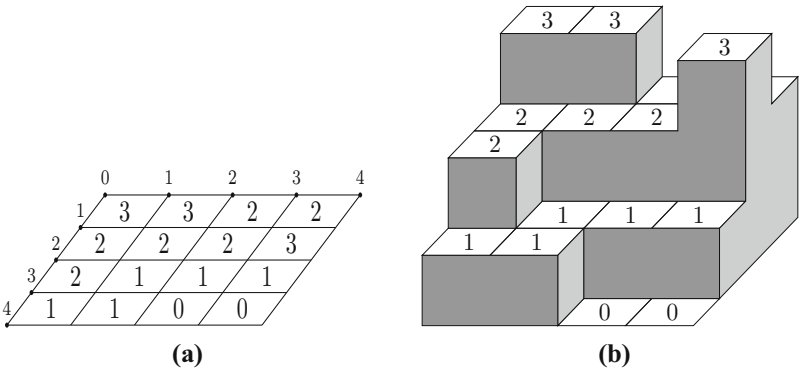

Fig. 4 **a** A matrix D on a grid, and **b** its corresponding histogram $his(D)$

- For any $n \times n$ matrix D, r_D denotes the minimum number of 3D rectangles $rec_D(i_1, i_2, k_1, k_2, h_1, h_2)$ which form a partition of $his(D)$.

Note that $r_D \leq n^2$ for any $n \times n$ matrix D, as $his(D)$ can be trivially partitioned into at most n^2 3D rectangles, each covering one grid cell.

3.2 Algorithm 1

We shall denote the cardinality of a set S (e.g., $S = [k_1, k_2] \cap [k'_1, k'_2]$) by $\#S$.

Our first geometric algorithm for nonnegative integer matrix multiplication relies on the following key lemma:

Lemma 3.2 *Let P_A be a partition of the matrix A into 3D rectangles $rec_A(i_1, i_2, k_1, k_2, h_1, h_2)$, and let P_B be a partition of the matrix B into 3D rectangles $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2)$. For any $1 \leq i \leq n$, $1 \leq j \leq n$, the entry $C_{i,j}$ of the matrix product C of A and B is equal to the sum of $(h_2 - h_1)(h'_2 - h'_1) \times \#[k_1, k_2] \cap [k'_1, k'_2]$ over rectangle pairs $rec_A(i_1, i_2, k_1, k_2, h_1, h_2) \in P_A$, $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2) \in P_B$ satisfying $i \in [i_1, i_2]$ and $j \in [j_1, j_2]$.*

Proof For $1 \leq l_1 < l_2 \leq n$ and $1 \leq m_1 < m_2 \leq n$, let $I(l_1, l_2, m_1, m_2)$ be the $n \times n$ $0-1$ matrix where $I(l_1, l_2, m_1, m_2)_{i,k} = 1$ if and only if $l_1 \leq i \leq l_2$ and $m_1 \leq k \leq m_2$.

Clearly, we have $A = \sum_{rec_A(i_1,i_2,k_1,k_2,h_1,h_2) \in P_A} (h_2 - h_1) I(i_1, i_2, k_1, k_2)$. Similarly, we have $B = \sum_{rec_B(k'_1,k'_2,j_1,j_2,h'_1,h'_2) \in P_B} (h'_2 - h'_1) I(k'_1, k'_2, j_1, j_2)$.

It follows that $C = A \times B$ is the sum over pairs $rec_A(i_1, i_2, k_1, k_2, h_1, h_2) \in P_A$, $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2) \in P_B$ of $(h_2 - h_1)(h'_2 - h'_1) \times I(i_1, i_2, k_1, k_2) \times I(k'_1, k'_2, j_1, j_2)$. It remains to observe that $(I(i_1, i_2, k_1 + 1, k_2) \times I(k'_1, k'_2, j_1 + 1, j_2))_{i,j} = \#[k_1, k_2] \cap [k'_1, k'_2]$ if $i_1 < i \leq i_2$ and $j_1 < j \leq j_2$ and it is equal to zero otherwise. □

See Fig. 5 for a visualization of Lemma 3.2. The following algorithm relies on Lemma 3.2 that basically says that for each pair of 3D rectangles,

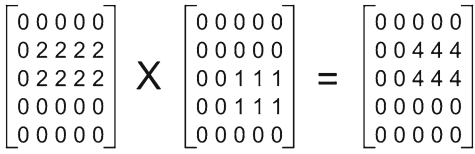

Fig. 5 An example of how Lemma 3.2 works. The two input matrices correspond to $rec_A(2, 3, 2, 5, 2, 4)$ and $rec_B(3, 4, 3, 5, 4, 5)$, respectively

$rec_A(i_1, i_2, k_1, k_2, h_1, h_2) \in P_A$ and $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2) \in P_B$, $C_{i,j}$ should be increased by $(h_2 - h_1) \times (h'_2 - h'_1) \times \#[k_1, k_2] \cap [k'_1, k'_2]$ for $i \in [i_1, i_2]$ and $j \in [j_1, j_2]$. In Step 4, two identical intervals $[i_1, i_2]$ corresponding to the left and right edge of the submatrix of C whose entries should be increased by the aforementioned value are inserted in the lists $Start_{j_1}$ and End_{j_2}, respectively. In both cases, they are weighted by the aforementioned value. In Step 5, in iteration j_1, the weighted interval $[i_1, i_2]$ from $Start_{j_1}$ is inserted into the weighted segment tree U, and in iteration $(j_2 + 1)$, it is removed from U as its copy is in End_{j_2}. In the iterations $j = j_1, \ldots, j_2$ in Step 5, when the interval $[i_1, i_2]$ is kept in the weighted segment tree U and the entries of the submatrix $C_{i,j}$, $i_1 \le i \le i_2$, $j_1 \le j \le j_2$, are evaluated, the weight of the interval contributes to their value.

Algorithm 1 *Input:* Two $n \times n$ matrices A, B with nonnegative integer entries.
Output: The arithmetic matrix product C of A and B.

1. Compute a partition P_A of $his(A)$ into 3D rectangles $rec_A(i_1, i_2, k_1, k_2, h_1, h_2)$ whose number is within $O(1)$ of the minimum.
2. Compute a partition P_B of $his(B)$ into 3D rectangles $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2)$ whose number is within $O(1)$ of the minimum.
3. Initialize an interval tree S on the k- and k'-coordinates of the rectangles in P_A and P_B. For each 3D rectangle $rec_A(i_1, i_2, k_1, k_2, h_1, h_2) \in P_A$, insert $[k_1, k_2]$ with a pointer to $rec_A(i_1, i_2, k_1, k_2, h_1, h_2)$ into S.
4. Initialize interval lists $Start_j$, End_j, for $j = 1, \ldots, n$. For each rectangle $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2) \in P_B$ report all intervals $[k_1, k_2]$ in S that intersect $[k'_1, k'_2]$. For each such $[k_1, k_2]$ with a pointer to $rec_A(i_1, i_2, k_1, k_2, h_1, h_2)$, insert the interval $[i_1, i_2]$ with the weight $(h_2 - h_1) \times (h'_2 - h'_1) \times \#[k_1, k_2] \cap [k'_1, k'_2]$ into the lists $Start_{j_1}$ and End_{j_2}.
5. Initialize the weighted segment tree U on endpoints $1, \ldots, n$. For $j = 1, \ldots, n$, iterate the following steps. For $j > 1$, remove all weighted intervals $[i_1, i_2]$ on the list End_{j-1} from U. Insert all weighted intervals $[i_1, i_2]$ on the list $Start_j$ into U. For $i = 1, \ldots, n$, set $C_{i,j}$ to the value returned by U in response to the weight query for i (see also Fig. 6).

Lemma 3.3 *Let $int(P_A, P_B)$ be the number of pairs $rec_A(i_1, i_2, k_1, k_2, h_1, h_2) \in P_A$, $rec_B(k'_1, k'_2, j_1, j_2, h'_1, h'_2) \in P_B$, for which $[k_1, k_2] \cap [k'_1, k'_2] \neq \emptyset$. Algorithm 1 runs in $\tilde{O}(n^2 + int(P_A, P_B)) = \tilde{O}(n^2 + r_A r_B)$ time.*

Proof To implement steps 1 and 2 in $\tilde{O}(n^2)$ time, use the algorithm from Theorem 2.5 in Sect. 2.2. Step 3 can be implemented in $\tilde{O}(n + r_A + r_B) = O(n^2)$ time by Fact 1. In

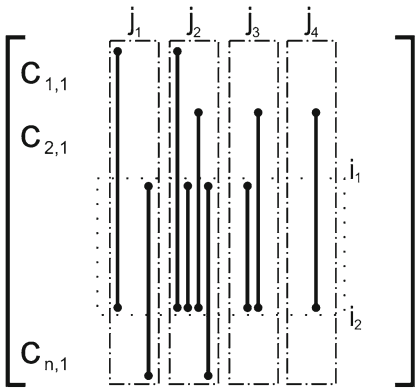

Fig. 6 Illustrating how to fill in the entries of C column-wise in Step 5 of Algorithm 1 by sweeping and updating the weighted segment tree U from the left to the right. In particular, $[i_1, i_2] \in Start_{j_2} \cap End_{j_3}$ holds

Step 4, the intersection queries to S take $\tilde{O}(int(P_A, P_B))$ time by Fact 1. In Step 5, the initialization of the weighted segment tree U takes $\tilde{O}(n)$ time by Lemma 3.1. Next, the updates of the weighted segment tree U take $\tilde{O}(int(P_A, P_B))$ time by Lemmas 3.1 and 3.2, while computing all columns of C takes $\tilde{O}(n^2)$ time by Lemma 3.1. □

Theorem 3.4 *The matrix product of two $n \times n$ matrices A, B with nonnegative integer entries can be computed in $\tilde{O}(n^2 + r_A r_B)$ time.*

Proof Algorithm 1 yields the theorem. Its correctness follows from Lemma 3.2 by the description preceding its pseudocode. The upper time bound follows from Lemma 3.3. □

3.3 Algorithm 2

When only one of the matrices A and B admits a partition of its 3D histogram into relatively few 3D rectangles and we have to assume the trivial partition of the other one into at most n^2 3D rectangles, the upper bound of Theorem 3.4 in terms of r_A, r_B and n seems too weak. In this case, an upper bound in terms of $int(P_A, P_B)$ and n in Lemma 3.3 may be much better. To derive a better upper bound in terms of just $\min\{r_A, r_B\}$ and n, we shall design another algorithm based on the slicing of the 3D histogram admitting a partition into relatively few 3D rectangles. Intuitively, the new algorithm goes over vertical consecutive slices of the 3D histogram corresponding to the rows of the input matrix A. It records the changes between two consecutive slices by identifying the so called differentiating strips for them. Our definitions ensure that the total number of changes is bounded by $O(r_A)$ which allows a running time of $\tilde{O}(n(n + r_A))$.

We need now to formalize the new approach. For an $n \times n$ matrix D with nonnegative integer entries and $i = 1, \ldots, n$, let $slice_i(D)$ stand for the part of $his(D)$ between the two planes perpendicular to the Y-axis whose intersection with the XY plane are the horizontal lines $i - 1$ and i on the $[0, n] \times [0, n]$ grid. In other words, $slice_i(D)$ is a 3D histogram for the i-th row. *Note that if we allow an orthogonal 2D histogram to have some non-base edges overlapping with its base then $slice_i(D)$ can be identified*

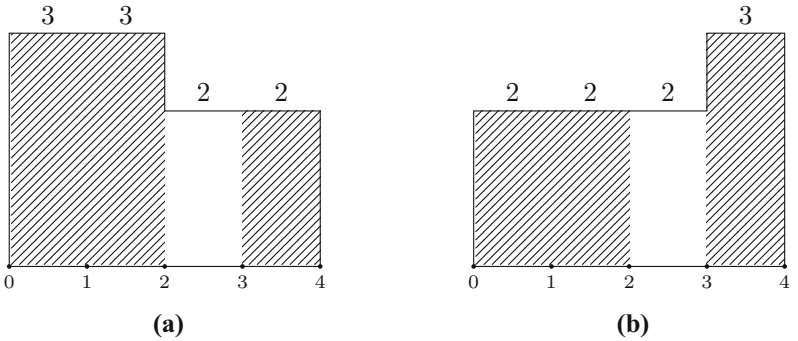

Fig. 7 Let $slice_1(D)$ be the 2D histogram on the left and $slice_2(D)$ the 2D histogram on the right. Differentiating strips are shaded. Here, $gd(slice_1(D), slice_2(D)) = 2$

with an orthogonal 2D histogram whose non-base edges overlapping with the base correspond to maximal sequences of consecutive zero entries in the ith row of D. See Fig. 7 for an example.

By a vertical *strip*, we shall mean the closed part of the XY plane bounded by two vertical straight-lines. For a vertical strip s and a $slice_i(D)$, $k_1(s)$ stands for the lowest column number of D such that the projection of the cube corresponding to the entry $D_{i,k_1(s)}$ (see Fig. 4) on the XY plane is fully contained in s. Symmetrically, $k_2(s)$ stands for the largest column number of D such that the projection of the cube corresponding to the entry $D_{i,k_2(s)}$ on the XY plane is fully contained in s.

Finally, a vertical strip s is *differentiating* for two orthogonal histograms H_1 and H_2 on the XY plane with a common horizontal (i.e., parallel to the X-axis) base if it passes through single edges of H_1 and H_2, that are parallel to the base and do not overlap. More precisely,

1. for $i = 1, 2$, s contains exactly one maximal subsegment e_i of an edge of H_i different from and parallel to the base of the histograms, and
2. the subsegments e_1 and e_2 do not overlap.

We shall also denote the difference between the Y coordinates of e_2 and e_1 by $h(s)$.

The idea behind our concept of *geometric distance* between two orthogonal histograms H_1 and H_2 with a common horizontal base is to capture the number of changes necessary to transform H_1 into H_2 or *vice versa*. It can be regarded as a generalization of the Hamming distance between two binary sequences. We define the geometric distance between the two histograms as the number of maximal differentiating (vertical) strips for the histograms. Next, for $slice_i(D)$ and $slice_k(D)$, we define the geometric distance $gd(slice_i(D), slice_k(D))$ as that for the corresponding orthogonal 2D histograms placed on the XY plane above the X-axis and to the right of the Y-axis such that the interval $[0, n]$ on the X-axis forms their common base.

Lemma 3.5 *For an $n \times n$ matrix D with nonnegative integer entries, $\sum_{i=1}^{n-1} gd(slice_i(D), slice_{i+1}(D)) = O(r_D)$ holds.*

Proof Consider the partition of $hist(D)$ into at most $4r_D$ 3D rectangles produced by our 4-approximation algorithm given in the preceding section. Note that assuming that

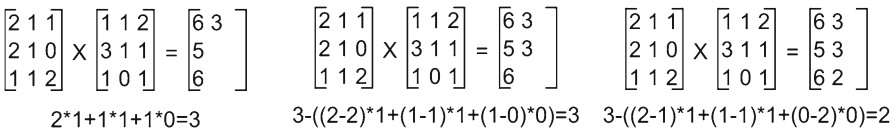

Fig. 8 An example of how Algorithm 2 fills in the entries in a column of the output matrix

the XY plane is the base plane for $hist(D)$, each face of a 3D rectangle in the partition that is parallel to the XZ plane, lies on a vertical plane bounding one or two slices of $hist(D)$.

Next, for any pair of consecutive slices of $hist(D)$, consider the vertical plane which separates them. In the aforementioned rectangular partition of D, the number of vertical and/or horizontal edges of the 3D rectangles in the partition which lie on this separating plane has to be at least proportional to the geometric distance $gd(\ ,\ )$ between the two slices. Since any 3D rectangle can contribute with at most four such vertical and four such horizontal edges lying on the bounding planes between slices, the lemma follows. (Analogous bounds may also be obtained by using the arguments from the proof of Theorem 2.5.) □

Now the idea of our second algorithm is simple. First we precompute differentiating strips for each pair of consecutive slices of the matrix A. Next,for each column j of the matrix B, we compute the first entry $C_{1,j}$ in the j-th column of the output matrix C from scratch. Then, we compute the consecutive $C_{i+1,j}$ entry by updating the previous $C_{i,j}$ entry on the basis of the precomputed differentiating strips. The total cost of updates for the column j will be proportional to $O(r_A)$ by Lemma 3.5.

Algorithm 2 *Input:* Two $n \times n$ matrices A and B with nonnegative integer entries. *Output:* The matrix product C of A and B.

1. For $i = 1, \ldots, n-1$, find the differentiating strips for $slice_i(A)$ and $slice_{i+1}(A)$, and for each such strip s determine the indices $k_1(s)$ and $k_2(s)$, and the difference $h(s)$. (Note that the interval of entries $A_{i,k_1(s)}, \ldots, A_{i,k_2(s)}$ in the i-th row of A is "covered" by s, and $h(s)$ is the difference between the common value of each entry in $A_{i,k_1(s)}, \ldots, A_{i,k_2(s)}$ and the common value of each entry in $A_{i+1,k_1(s)}, \ldots, A_{i+1,k_2(s)}$).
2. For $j = 1, \ldots, n$, iterate the following steps:
   (a) Initialize a data structure T_j that for a pair k_1, k_2 of indices reports $\sum_{k_1}^{k_2} B_{k,j}$, using Fact 2.
   (b) Compute $C_{1,j}$.
   (c) For $i = 1, \ldots, n-1$, iterate the following steps:
      i. Set $C_{i+1,j}$ to $C_{i,j}$.
      ii. For each differentiating strip s for $slice_i(A)$ and $slice_{i+1}(A)$, compute $\sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$ using T_j and set $C_{i+1,j}$ to $C_{i+1,j} + h(s) \sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$ (see Fig. 8 for an illustration).

Lemma 3.6 *Algorithm 2 runs in* $\tilde{O}(n(n + r_A))$ *time.*

Proof Step 1 can be easily implemented in $O(n^2)$ time. Step 2 (a) takes $\tilde{O}(n)$ time according to Fact 2 while Step 2 (b) can be trivially implemented in $O(n)$ time. Finally, based on Step 1, Step 2 (c)-ii takes $\tilde{O}(gd(slice_i(D), slice_{i+1}(D))$ time. It follows that Step 2 (c) can be implemented in $\tilde{O}(\sum_{i=1}^{n-1} gd(slice_i(A), slice_{i+1}(A)))$ time, i.e., in $\tilde{O}(r_A)$ time by Lemma 3.5. Consequently, Step 2 takes $\tilde{O}(n(n + r_A))$ time. □

Theorem 3.7 *The arithmetic matrix product of two $n \times n$ matrices A, B with non-negative integer entries can be computed in $\tilde{O}(n(n + \min\{r_A, r_B\}))$ time.*

Proof The correctness of Algorithm 2 follows from the observation that a differentiating strip s for $slice_i(A)$ and $slice_{i+1}(A)$ yields the difference $h(s) \sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$ between $C_{i+1,j}$ and $C_{i,j}$ just on the fragment corresponding to $A_{i,k_1(s)}, \ldots, A_{i,k_2(s)}$ and $A_{i+1,k_1(s)}, \ldots, A_{i+1,k_2(s)}$, respectively.

Lemma 3.6 yields the upper bound $\tilde{O}(n(n+r_A))$ on the running time. The analogous bound $\tilde{O}(n(n+r_B))$ follows from the equalities $AB = (B^T A^T)^T, his(B) \equiv his(B^T)$, and consequently $r_B = r_{B^T}$. □

3.4 Algorithm 3

In Algorithm 2, the linear order in which the $C_{i,j}$ are updated to $C_{i+1,j}$ for $i = 1, \ldots, n-1$, along the row order of the matrix A is not necessarily optimal. The total number of updates of $C_{i,j}$ to $C_{i+1,j}$ for $i = 1, \ldots, n-1$ is proportional to the sum of geometric distances between consecutive slices of $hist(A)$, which is $O(r_A)$ by Lemma 3.5. We can think about the latter sum as the cost of a particular line spanning tree in a complete undirected graph whose vertices are in one-to-one correspondence with the slices of $his(A)$ and where each edge $\{i, j\}$ in is assigned the weight $gd(slice_i(A), slice_j(A))$. Following the Boolean case [2,5], it may be more efficient to update $C_{i,j}$ while traversing a minimum spanning tree for the slices of $his(A)$ under the geometric distance. Note that the cost of the minimum spanning tree and hence the total number of updates of entries in the j-th column of C might be substantially smaller than the cost of the aforementioned line tree and it will never exceed $O(r_A)$. Here, however, we encounter the difficulty of constructing such an optimal spanning tree or a close approximation in substantially subcubic time. The next lemma will be useful.

Lemma 3.8 *Consider the family of orthogonal planar histograms with the base $[0, n]$ for any $n \geq 2$ and integer coordinates of its vertices in $[0, 2^K - 2]$, where $K = O(\log n)$. There is an $O(n)$-time transformation of any histogram H in the family into a $0-1$ string $t(H)$ such that, for any H_1 and H_2 in the family, $gd(H_1, H_2) \leq hd(t(H_1), t(H_2)) \leq K \cdot gd(H_1, H_2)$, where $hd(,)$ stands for the Hamming distance.*

Proof Any histogram H in the family is uniquely represented by the vector $(H[1], \ldots, H[n]) \in \{1, \ldots, 2^K - 1\}^n$, where $H[1], \ldots, H[n]$ are the values of Y coordinates of the points on the "roof" of H increased by one with X coordinates $0.5, 1.5, \ldots, n - 0.5$ respectively.

For any $y \in \{0, \ldots, 2^K - 1\}$ denote its binary representation of length exactly K (padded with leading zeros if necessary) as $\mathrm{bin}(y)$.

Let $f(H,i) = \begin{cases} \text{bin}(H[i]), & i = 1 \vee i > 1 \wedge H[i] \neq H[i-1] \\ \text{bin}(0), & \text{otherwise.} \end{cases}$

The transformation t is then defined as $t(H) = f(H,1) \cdots f(H,n)$. We have $hd(t(H_1), t(H_2)) = \sum_{i=1}^{n} hd(f(H_1,i), f(H_2,i))$ and

$$gd(H_1,H_2) = \begin{cases} 1, & H_1[1] \neq H_2[1] \\ 0, & \text{otherwise} \end{cases} + \sum_{i=2}^{n} \begin{cases} 1, & (H_1[i] \neq H_1[i-1] \vee H_2[i] \neq H_2[i-1]) \wedge (H_1[i] \neq H_2[i]) \\ 0, & \text{otherwise.} \end{cases}$$

Consider all possibilities that contribute exactly one to $gd(H_1, H_2)$:

1. $H_1[1] \neq H_2[1]$. In this case $f(H_1,1) = \text{bin}(H_1[1])$, $f(H_2,1) = \text{bin}(H_2[1])$ and $1 \leq hd(\text{bin}(H_1[1]), \text{bin}(H_2[1])) \leq K$.
2. $2 \leq i \leq n \wedge H_1[i] \neq H_1[i-1] \wedge H_2[i] = H_2[i-1] \wedge H_1[i] \neq H_2[i]$. In this case $f(H_1,i) = \text{bin}(H_1[i])$, $f(H_2,i) = \text{bin}(0)$ and $1 \leq hd(\text{bin}(H_1[i]), \text{bin}(0)) \leq K$.
3. $2 \leq i \leq n \wedge H_1[i] = H_1[i-1] \wedge H_2[i] \neq H_2[i-1] \wedge H_1[i] \neq H_2[i]$. See case 2.
4. $2 \leq i \leq n \wedge H_1[i] \neq H_1[i-1] \wedge H_2[i] \neq H_2[i-1] \wedge H_1[i] \neq H_2[i]$. See case 1.

To complete the proof, observe that in all other cases $hd(f(H_1,i), f(H_2,i)) = 0$. □

Fact 3 (Section 3.3 in [6]). *For $\epsilon > 0$, a $(1+\epsilon)$-approximate minimum spanning tree for a set of n points in R^d with integer coordinates in $O(1)$ under the L_1 or L_2 metric can be computed by a Monte Carlo algorithm in $O(dn^{1+1/(1+\epsilon)})$ time.*

By combining the transformation of Lemma 3.8 with Fact 3 applied to the L_1 metric in $\{0,1\}^n$ and selecting $\epsilon = \log n$, we obtain the following lemma.

Lemma 3.9 *Let A be an $n \times n$ matrix with nonnegative integer entries in $[0, n^{O(1)}]$. An $O(\log^2 n)$-approximate minimum spanning tree for the set of slices of $his(A)$ under the $gd(\,)$ distance can be constructed by a Monte Carlo algorithm in $\tilde{O}(n^2)$ time.*

By using Lemma 3.9, we obtain the following generalization of Algorithm 2.

Algorithm 3 *Input:* Two $n \times n$ matrices A and B with nonnegative integer entries in $[0, n^{O(1)}]$.
Output: The matrix product C of A and B.

1. Find an $O(\log^2 n)$-approximate spanning tree S for $slice_i(A), i = 1, \ldots, n$, under the geometric distance and a traversal (i.e., a non-necessarily simple path visiting all vertices) U of S.
2. For any pair $(slice_i(A), slice_l(A))$, where the latter slice follows the former in the traversal, find the differentiating strips for $slice_i(A)$ and $slice_l(A)$, and for each such strip s the indices $k_1(s)$ and $k_2(s)$, and the difference $h(s)$.
3. For $j = 1, \ldots, n$, iterate the following steps:

(a) Initialize a data structure T_j that for a pair k_1, k_2 of indices reports $\sum_{k_1}^{k_2} B_{k,j}$, using Fact 2.
(b) Compute $C_{q,j}$ where q is the index of the slice from which the traversal U of S starts.
(c) While following U, iterate the following steps:
   i. Set i, l to the indices of the previously traversed slice and the currently traversed slice, respectively.
   ii. Set $C_{l,j}$ to $C_{i,j}$.
   iii. For each differentiating strip s for $slice_i(A)$ and $slice_l(A)$, compute $\sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$ using T_j and set $C_{l,j}$ to $C_{l,j} + h(s) \sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$.

Definition 3.10 For an $n \times n$ matrix D with non-negative integer entries in $[0, n^{O(1)}]$, let M_D stand for the minimum cost of a spanning tree of $slice_i(D)$, $i \in [1, n]$.

Lemma 3.11 *Algorithm 3 runs in $\tilde{O}(n(n + M_A))$ with high probability.*

Proof The approximate minimum spanning tree S in Step 1 can be constructed by a Monte Carlo algorithm in $\tilde{O}(n^2)$ time by Lemma 3.9. Its traversal can be easily found in $O(n)$ time. Since the length of the traversal is linear in n, Step 2 can be easily implemented in $O(n^2)$ time analogously as the corresponding Step 1 in Algorithm 2. Finally, based on Step 2, Step 3 (c)-ii takes $\tilde{O}(gd(slice_i(D), slice_l(D)))$ time analogously as Step 2 (c)-ii in Algorithm 2. Let U stand for the set of directed edges forming the traversal of the spanning tree S. It follows that Step 3 (c) can be implemented in $\tilde{O}(\sum_{(i,l)\in U} gd(slice_i(A), slice_l(A)))$ time, i.e., in $\tilde{O}(M_A)$ time by Lemma 3.9. Consequently, Step 3 takes $\tilde{O}(n(n + M_A))$ time. □

By Lemma 3.11 and a proof analogous to that of Theorem 3.7, we obtain a randomized generalization of Theorem 3.7 for matrices with nonnegative polynomially bounded integer entries.

Theorem 3.12 *Let A, B be two $n \times n$ matrices A, B with nonnegative integer entries in $[0, n^{O(1)}]$. The arithmetic matrix product of A and B can be computed by a randomized algorithm in $\tilde{O}(n(n + \min\{M_A, M_{B^T}\}))$ time with high probability.*

Proof The correctness of Algorithm 3 analogously as that of Algorithm 2 follows from the observation that a differentiating strip s for $slice_i(A)$ and $slice_{i+1}(A)$ yields the difference $h(s) \sum_{k=k_1(s)}^{k_2(s)} B_{k,j}$ between $C_{i+1,j}$ and $C_{i,j}$ just on the fragment corresponding to $A_{i,k_1(s)}, \ldots, A_{i,k_2(s)}$ and $A_{i+1,k_1(s)}, \ldots, A_{i+1,k_2(s)}$, respectively. Lemma 3.11 yields the upper bound $\tilde{O}(n(n + M_A))$. The symmetric one $\tilde{O}(n(n + M_{B^T}))$ follows from the equality $AB = (B^T A^T)^T$. □

4 Final Remarks

A natural question is: In Theorem 2.5 in Sect. 2.2, would it help to replace the fast but suboptimal algorithm from Lemma 2.3 by a (slower) algorithm that optimally rectangulates the 2D projection? The answer is that it may yield improved results in

certain cases, but it would not give a better approximation factor than 4 in general. An example of this is when the optimal 3D partition consists of k cubes of decreasing sizes lying on top of each other. Then the 2D projection is k concentric squares of different sizes and an optimal rectangulation of the corresponding 2D projection consists of $4k - 3$ rectangles. Here, the approximation factor tends to 4 as k increases, and we conclude that the fast algorithm from Lemma 2.3 is good enough.

As mentioned in Sect. 1, the general problem of computing a minimum 3D rectangular partition of an *unrestricted* orthogonal polyhedron is NP-hard [4]. However, it is unknown whether the problem is NP-hard or not in the special case where the input is a *3D histogram*. Although the existence of a polynomial-time algorithm for this particular problem variant would not affect the time complexity of our matrix multiplication algorithms (because the constant-factor approximations of r_A and r_B incurred by Theorem 2.5 are absorbed into the asymptotic running times anyway), it might be an interesting theoretical issue to resolve.

The 4-approximation algorithm for minimum rectangular partition of a 3D histogram in case the histogram is $his(D)$ for an input $n \times n$ matrix D with nonnegative integer entries can be implemented in $O(n^2)$ time. Also note that the resulting partition of $his(D)$ can be used to form a compressed representation of D requiring solely $\tilde{O}(r_D)$ bits if the values of the entries in D are $n^{O(1)}$-bounded.

Our geometric algorithms for integer matrix multiplication can also be applied to derive faster $(1+\epsilon)$-approximation algorithms for integer matrix multiplication; if the range of an input matrix D is $[0, n^{O(1)}]$, then round each entry to the smallest integer power of $(1 + \epsilon)$ that is not less than the entry. The resulting matrix D' has only a logarithmic number of different entry values and hence $r_{D'}$ may be much less than r_D.

We also note that our algorithms and upper time-bounds for integer $n \times n$-matrix multiplication can be extended to integer *rectangular* matrix multiplication in a straightforward way. In particular, if the input matrices A, B have sizes $p \times q$ and $q \times r$, respectively, then the upper time-bounds of Theorems 3.4, 3.7 generalize to $\tilde{O}(pq + qr + \min\{r_A r, r_B p, r_A r_B\})$.

The parameters r_D and M_D can be seen as far-going generalizations of the concept of matrix sparsity. Note that if D has at most s non-zero entries then $r_D = O(s)$. If r_D is low then D can be partitioned into relatively few rectangular blocks of entries with uniform values. Matrices roughly describing some country or urban landscape (e.g., in graphics) can have this property. Hence, our algorithms for geometric matrix multiplications would be especially efficient in implementation of linear transformations of the aforementioned descriptions. Another example are adjacency matrices D of uniform disk graphs, induced by point sets of bounded density within a unit square, for which M_D is known to be substantially subquadratic (see Lemma 8 in [10]).

Finally, our geometric algorithms for matrix multiplication can be adapted to compute other matrix product of two integer $n \times n$ matrices, e.g., their distance product, i.e., the matrix product of the matrices over the semi-ring $(\mathbb{Z}, \min, +)$, within the same asymptotic complexity. In the case of Algorithm 1, we need to assume that the rectangular partitions of the matrices consist solely of 3D rectangles placed on the base plane. Then in step 4 of the adapted Algorithm 1, $h'_1 = h_1 = 0$, and we just assign the weight $h_1 + h_2$ to the interval $[i_1, i_2]$. We need also to adapt the weighted segment tree U in step 5, by assigning to each node of the data structure the minimum (instead of the

sum) of the weights of the intervals it represents. Thus, the answer to a weight query becomes the minimum of the weights of intervals covering the query point. Since our 4-approximation algorithm for minimum rectangular partition of a 3D histogram produces solely partitions satisfying the aforementioned additional assumption, the asymptotic complexity of the adapted algorithm remains the same. For Algorithms 2 and 3, we need just to replace sums with minima and multiplications with additions (in step 2(c)-ii of Algorithm 2 and step 3(c)-iii of Algorithm 3, respectively). Also, the data structure T_j needs to be modified to return the values of partial minima instead of partial sums in both cases.

Acknowledgements J.J. was funded by The Hakubi Project at Kyoto University. C.L. was supported in part by Swedish Research Council Grant 621-2011-6179.

References

1. Bansal, N., Williams, R.: Regularity lemmas and combinatorial algorithms. Theory Comput. **8**(1), 69–94 (2012)
2. Björklund, A., Lingas, A.: Fast Boolean matrix multiplication for highly clustered data. In: Proceedings of WADS 2001, LNCS, vol. 2125, pp. 258–263
3. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Santa Clara (2008)
4. Dielissen, V.J., Kaldewaij, A.: Rectangular partition is polynomial in two dimensions but NP-complete in three. Inf. Process. Lett. **38**(1), 1–6 (1991)
5. Gąsieniec, L., Lingas, A.: An improved bound on boolean matrix multiplication for highly clustered data. In: Proceedings of WADS 2003, LNCS, vol. 2748, pp. 329–339
6. Indyk, P.: High-dimensional Computational Geometry. PhD dissertation, Stanford University, September (2000)
7. Keil, J.M.: Polygon Decomposition. Dept. Comput. Sc. Univ. Saskatchewan, Survey (1996)
8. Le Gall, F.: Powers of tensors and fast matrix multiplication. In: Proceedings of the 39th ISSAC 2014, pp. 296–303
9. Lingas, A.: A geometric approach to Boolean matrix multiplication. In: Proceedings of ISAAC 2002, LNCS, vol. 2518, pp. 501–510
10. Lingas, A., Sledneu, D.: A combinatorial algorithm for all-pairs shortest paths in directed vertex-weighted graphs with applications to disc graphs. In: Proceedings of SOFSEM 2012, LNCS, pp. 373–384
11. Lipski, W.: Finding a Manhattan path and related problems. Networks **13**(3), 399–409 (1983)
12. Mehlhorn, K.: Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry. EATCS Monographs on Theoretical Computer Science. Springer, Berlin (1984)
13. Muthukrishnan, S., Poosala, V., Suel, T.: On rectangular partitionings in two dimensions: algorithms, complexity, and applications. In: Proceedings of ICDT'99, LNCS Vol. 1540, pp. 236–256
14. Sack, J.-R., Urrutia, J. (eds.): Handbook of Computational Geometry. Elsevier, Amsterdam (2000)