Improved Approximation Algorithms for Rectangle Tiling and Packing

Piotr Berman* Bhaskar DasGupta† S. Muthukrishnan‡ Suneeta Ramaswami§

Abstract

We provide improved approximation algorithms for several rectangle tiling and packing problems (RTILE, DRTILE and d-RPACK) studied in the literature. Our algorithms are highly efficient since their running times are near-linear in the sparse input size rather than in the domain size. In addition, we improve the best known approximation ratios, in some cases quite significantly.

1 Introduction

We study several rectangle tiling and packing problems. These are natural combinatorial problems that arise in many applications in databases, parallel computing and image processing. We present new approximation algorithms for these problems. In contrast to the previously known results, we meet a crucial demand of most of these applications, namely, our algorithms work on sparse inputs and/or high dimensions very efficiently. In addition, our algorithms have better approximation bounds than known algorithms. Furthermore, the algorithms are simple to implement. In what follows, we will first formally define the problems before presenting our results.

1.1 The Rectangle Tiling and Packing Problems

We study the following two classes of problems.

RTILE problem. Given a two dimensional array $\mathbf{A}$[1] of size $n \times n$ containing non-negative integers[2], partition $\mathbf{A}$ into at most p rectangular tiles so that the maximum weight of any tile is minimized (a *tile* is any rectangular subarray, a *partition* of an array into tiles is one in which each element of the array falls in precisely one tile with the tiles being disjoint and the *weight* of a tile is the total sum of all elements that fall in it). Our emphasis is on the *sparse* version of the problem, that is, one in which the total number of nonzero entries in $\mathbf{A}$ is at most m which is likely to be smaller than n^2, the total entries in $\mathbf{A}$.

A natural variant of RTILE is its dual, referred to as the DRTILE problem, wherein we are given a bound w and required to minimize the number of tiles needed to partition the given array so that *no* tile has weight larger than w. Another natural variant is the straightforward extension of the RTILE/DRTILE problems to d-dimensions, where $\mathbf{A}$ is a d-dimensional array of size $n \times n \times \cdots \times n$ and d-dimensional hyper-rectangles are considered in the tiling.

d-RPACK problem. Given a set of p weighted d-dimensional, axis-parallel hyper-rectangles with endpoints in $\{1, 2, 3, \ldots, n\}$ and a parameter k, find a collection of at most k disjoint hyper-rectangles of maximum total weight. The set of all possible hyper-rectangles in d dimensions is $n^{\Omega(d)}$, but p is likely to be significantly smaller, that is, the sparse case will be of our interest.

The d-RPACK problem on a set of hyper-rectangles is equivalent to finding the maximum-weight independent set in the intersection graph of these hyper-rectangles, where there is an edge between two nodes representing two distinct hyper-rectangles if and only they intersect (this problem has been studied with an alternate definition of the interference graph in [BNR96]). The dual of the RPACK problem is to find the minimum cardinality subset of disjoint hyper-rectangles with total weight at least w, for some given weight-bound w. Since it is NP-hard to even find a feasible solution for this dual problem [FPT81], it cannot be approximated to within any factor. Hence, we do not consider this dual problem any further.

1.2 Motivating Applications

Rectangle tiling and packing problems as defined above are natural combinatorial problems arising in many scenarios. For motiva-

*Department of Computer Science, Pennsylvania State University, University Park, PA 16802. Email: berman@cse.psu.edu. Supported in part by NSF grant CCR-9700053 and by NLM grant LM05110.

†Department of Computer Science, Rutgers University, Camden, NJ 08102. Email: bhaskar@crab.rutgers.edu. Supported in part by NSF Grant CCR-9800086.

‡AT& T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932. Email: muthu@research.att.com

§Department of Computer Science, Rutgers University, Camden, NJ 08102. Email: rsuneeta@crab.rutgers.edu

[1]For the RTILE/DRTILE problems, we will use bold letters to denote arrays/rectangles, and respective regular letters to denote their weights. In particular, input array $\mathbf{A}$ has weight A, and $\mathbf{R}_i$, the i^{th} row of of a two-dimensional array $\mathbf{A}$, has weight R_i.

[2]All our results will generalize to $n_1 \times n_2$ array. Henceforth, when we refer to *arbitrary array* $\mathbf{A}$ we imply one with non-negative integers.

tion, we will very briefly review one example application scenario for each and leave others to be found in the references.

Equisum Histograms in Databases. Databases such as DB2, SQLServer, Oracle, Sybase, Ingres etc. routinely keep histograms to optimize query execution [K80]. These histograms attempt to capture the distribution of attribute values in the database, for example, the joint distribution of the frequency of attribute values (e.g., age, salary, savings, etc.); the goal is to do so using limited memory and reliable accuracy. While the study of histograms is fundamental in areas such as Statistics, the use of multidimensional histograms in databases was initiated in [MD88], and has been intensely researched since then (see [P97] for a survey).

The RTILE problem is precisely that of constructing what is known as the Equisum histogram on 2 (more generally d) numerical attributes as defined in [MD88]. In this context, $\mathbf{A}$ is defined as follows: $\mathbf{A}[i,j]$ is 1 if there exist database records with attributes values i and j respectively (e.g., salary of 100K and savings 500K, respectively) and 0 otherwise; p, the number of tiles, represents the space allocated for the histogram; and w, the maximum weight of a tile, affects the accuracy of estimating the joint distribution. This gives the RTILE problem where A is a $\{0,1\}$-array. More generally, we may let $\mathbf{A}[i,j]$ be the number of database records that have attribute value i and j respectively, and we would obtain the RTILE problem with non-negative integral elements. Both versions of the problem are of interest in selectivity estimation. (DRTILE problem is the dual of RTILE, which is equally interesting for building histograms.) ∎

An important feature of this application is the sparsity of the input. That is, $\mathbf{A}[i,j]$ is nonzero only for the combination of attribute values that are present in the database, which is typically significantly smaller than the set of all possible attribute value combinations. It is imperative to exploit sparseness in database applications.

RTILE and DRTILE problems, and their variants, are also of interest in other applications in load balancing, database compression, data partitioning for parallel computers, image and video processing, etc. See discussion and references in [MPS99, KMP98].

Database Decision Support. Database mining systems generate association rules — rules of the form $C_1 \rightarrow C_2$, that is, if condition C_1 is satisfied in the database, C_2 follows. Correlation rules of this form need support and confidence to be tagged by database mining systems. For numeric attributes, the conditions on the left take the form of *clusters* (ranges), an anecdotal example being: $(Age \in [25,45]) \cap (Balance \in [15K - 40K]) \rightarrow (CarLoan = Yes)$. (Here, there are two numeric attributes, namely, Age and Balance. In general, there could be many other numeric attributes such as Mutual Fund Investments etc.) Study of such clustered association rules can be found in [FM+96a, FM+96b, RS99]. Database mining systems can now generate the set of all such rules for a given database, given thresholds on confidence and support, and tag each with a weight that shows their gain or value. Following that, database decision systems choose a subset of these rules for further development, such as marketing. A common formulation of this task is to choose k disjoint rules of largest total gain or value (for example, see [FM+96a, RS99] for formulation of this problem and extensive experimental study). This is precisely the d-RPACK problem where the d dimensions correspond to the numeric attributes. ∎

Besides the above application, the d-RPACK problem arises in various resource allocation problems. It is also a natural combinatorial problem when viewed as the maximum-weight independent set problem on the corresponding intersection graph of the hyper-rectangles. In the motivating application above, the set of hyper-rectangles that is generated is typically much smaller than the set of all possible hyper-rectangles. *The focus is therefore on the sparse input case.*

1.3 Summary of Our Results and Related Research

Since both the tiling and packing problems are known to be NP-Hard [FPT81, KMP98] in two or more dimensions[3], goal is to design approximation algorithms with guaranteed performance bounds. Naturally, our focus is to design approximation algorithms with better performance ratios than previously known, but additionally, our focus is to design such approximation algorithms whose time complexity is efficient as a function of the *sparse input size* (e.g., $m+n$ in the RTILE and DRTILE problems in two dimensions, p in the d-RPACK problem) rather than merely being efficient in the universe size (i.e., n^2 and n^d, respectively). None of the existing algorithms for these problems fully meet the latter goal. Table 1 summarizes our main results and compares them with the previously best known results.[4] Note the following:

- *Tiling Problems.* Our algorithms for the RTILE/DRTILE problems take time linear in the

[3] The one dimensional versions of tiling and packing can be solved in polynomial time by dynamic programming.

[4] All logarithms are in base 2 unless the base is mentioned explicitly.

Problem	Our results		Previous best		
	ratio	time-O	ratio	time-O	reference
RTILE, **A** is $\{0,1\}$	2	$n+m$	$9/4$	$n^2+p\log n$	[KMP98]
RTILE	$11/5$	$n+m$	$7/3$	n^2	[S99]
DRTILE, **A** is $\{0,1\}$	2	$n+m$	$9/4$ 2	$n^2+p\log n$ n^{10}	[KMP98] [KMP98]
DRTILE, d-dim	$2d-1$	$d(m+n)$	$2d$	$n^d+pd2^d\log n$	[S99, SS99]
d-RPACK	$(\lfloor 1+\log n\rfloor)^{d-1}$	$dp\log^{\varepsilon} p$ $+dn\frac{\log n}{\log\log n}+pk$	$6\lfloor 1+\log n\rfloor$	$n^2p+np^2\log n$	[KMP98] $d=2$ only
	$\left(\lfloor 1+\frac{\log n}{c}\rfloor\right)^{d-1}$	$p^{(2^c-1)^{d-1}+1}\ k$			

Table 1: Summary of our main results and comparisons with previous results. For the RTILE/DRTILE problems, n is the size of each dimension, m is the number of non-zero entries in the array ($m \le n^d$ for a d-dimensional array), and $p = A/w$. Unless otherwise noted **A** has arbitrary entries. For the d-RPACK problem, p is the total number of hyper-rectangles, n is the size of each dimension, k is the number of rectangles to be selected, and $0 < \varepsilon \le 1$ and $c \ge 1$ are any two constants.

sparse input size, that is $O(m+n)$ time, and are simple to implement.

- *Packing Problems.* The $(\lfloor 1+\log n\rfloor)^{d-1}$-approximation algorithm for packing problem takes time almost linear in sparse input size (in p). No results were previously known for the d dimensional case; straightforward extension of the known result for the two dimensional case [KMP98] would be an $\Omega(\log^{2d-1} n)$ approximation taking time $n^{\Omega(d)}$. Hence, our result is a substantial improvement. We can further improve the approximation ratio of our algorithm at the expense of running time as shown in the table above. This complicates the algorithm but it is theoretically interesting that the $\log n$ barrier does not hold.

1.4 Brief Technical Overview of Results

Tiling Problems. While previous algorithms used the concept of "thick cuts" or "medium cuts" and adopted a divide-and-conquer approach, we adopt a "sweep" based technique and develop the concept of "good rectangles" in the array, i.e., those rectangles whose total weight is at least g (for some parameter g carefully chosen to justify the approximation bounds) and combine these rectangles, whenever necessary, in a local manner. The benefit is that while thick or medium cuts are somewhat expensive to find, good rectangles can be found quite simply by sweeping through the input and combining them is also algorithmically easy. Thus our algorithms have efficient implementation in the sparse input size as applications demand. Somewhat surprisingly, such simple approaches in fact yield improved approximation ratios.

The improvement in the approximation ratio for the DRTILE problem in d dimensions over the previously best known algorithm uses an alternative lower bound in addition to the usual lower bound considered in [KMP98, S99, SS99]. The two lower bounds together lead to an improved analysis of the performance ratio.

Packing problem. The $(\lceil \log(n+1)\rceil)^{d-1}$-approximation algorithm that we present uses a divide-and-conquer technique. The exact solution at each level is found by "translating" the rectangles in the subproblems at each node in the level so that their "lower dimensional projections" do not interfere. For the improved result with approximation ratio $\left(\left\lfloor 1+\frac{\log n}{c}\right\rfloor\right)^{d-1}$ we do this at several consecutive levels of the divide-and-conquer simultaneously. As a result, the algorithm becomes more involved.

1.5 The Map

In Section 2, we present some necessary definitions. Our approximation algorithms for the RTILE and DRTILE problems are in Section 3. Section 4 contains our results on d-RPACK problem. Concluding remarks are in Section 5. Due to space limitations many proofs are omitted.

2 Definitions and Preliminaries

Unless otherwise stated, all rectangles are *axis-parallel* and all weights are *non-negative*. A *d-dimensional hyper-rectangle* is $[x_{11}, x_{12}]\times[x_{21}, x_{22}]\times\cdots\times[x_{d1}, x_{d2}] = \times_{i=1}^{d}[x_{i1}, x_{i2}]$, where each $[x_{i1}, x_{i2}]$ is an interval on the real line and $\times$ denotes the Cartesian product. A 2-dimensional hyper-rectangle is simply called a *rectangle*.

Two d-dimensional hyper-rectangles $\times_{i=1}^{d}[x_{i1}, x_{i2}]$ and $\times_{i=1}^{d}[y_{i1}, y_{i2}]$ intersect if and only if $[x_{i1}, x_{i2}] \cap [y_{i1}, y_{i2}] \neq \emptyset$ for every i. An array is called a $\{0,1\}$-array if all of its entries are either 0 or 1; otherwise it is called a *general* or *arbitrary* array. A *partition* of an array $\mathbf{A}$ is a collection of (axis-parallel) rectangles (also referred to as *tiles*) that cover $\mathbf{A}$ without overlap. The *weight* of a rectangle in a partition of $\mathbf{A}$ is the sum of all array elements covered by the rectangle. If a given $n \times n$ array is sparse, containing m non-zero entries ($n \leq m \leq n^2$), then it can be efficiently represented in $O(m+n)$ space using the standard representation as an array of row lists; the list of the i^{th} row contains an entry of the form (j, x) for every positive array entry $\mathbf{A}[i,j] = x$ and the row lists are sorted by the column numbers of the entries. A similar standard sparse representation can be used for a d-dimensional array $\mathbf{A}$: a non-zero entry $\mathbf{A}[i_1, i_2, \ldots, i_d]$ is represented by a $(d+1)$-tuple $(i_1, i_2, \ldots, i_d \ ; \ \mathbf{A}[i_1, i_2, \ldots, i_d])$ and the tuples are lexicographically sorted in their first d coordinates. Hence, we assume that our sparse matrices are represented this way.

3 Approximating the Tiling Problem

First, we describe a general slicing technique that is used many of our algorithms for RTILE and DRTILE problems. We then focus on the RTILE and DRTILE problems on $\{0,1\}$-arrays and later consider these problems on general arrays.

Slicing Arrays. The goal of many of our algorithms is to divide the input array $\mathbf{A}$ into *good* rectangles, which will mean rectangles that have weight at most g. Selection of g will depend on the particular algorithm. Our algorithms have a common beginning: first, the value of g is established, and then the given array $\mathbf{A}$ is divided into slices according to the weight limit g. The general scheme for the slicing algorithm is shown below. For $j < l$ the rows from $\mathbf{R}_{t[j-1]+1}$ to $\mathbf{R}_{t[j]}$ form the j^{th} slice, denoted $\mathbf{S}_j$. The rows with numbers larger than $t[l-1]$ form the remainder slice, $\mathbf{S}_l$. For $j < l$, we define the *top* of the slice $\mathbf{S}_j$ as $\mathbf{T}_j = \mathbf{R}_{t_j}$. The other rows of this slice form its *base*, $\mathbf{B}_j$.

```
t[0] ← 0;
j ← 1;
slice_weight ← 0;
for i ← 1 to n do
    slice_weight ← slice_weight + R_i;
    if slice_weight > g then
        t[j] ← i;
        j ← j + 1;
        slice_weight ← 0;
l ← j
```

LEMMA 3.1. *If the input array $\mathbf{A}$ has 0 and 1 entries only, we can partition it into $\lceil 2A/g \rceil$ good rectangles in $O(n+m)$ time.*

Proof. We can partition $\mathbf{A}$ into slices and compute all R_i's, S_j's, T_j's and B_j's in time $O(n+m)$. Note that the remainder slice is good and the other slices are not. It suffices, for $j < l$, to divide slice $\mathbf{S}_j$ into $a_j < 2S_j/g$ good rectangles. We consider three cases.

Case 1: $T_j \leq g$. In this case both $\mathbf{T}_j$ and $\mathbf{B}_j$, the top and the base of $\mathbf{S}_j$ are good, so they form our partition into rectangles. Note that $S_j > g$, and thus $a_j = 2 < 2S_j/g$.

Case 2: $T_j > g$ and $2S_j/g \leq 3$. This implies that $B_j < g/2$. Assume that the first k entries of $\mathbf{T}_j$ contain $g - B_j$ 1's. Then the rectangle formed from the first k columns of $\mathbf{S}_j$ must be good, because its weight is at most $(g - B_j) + B_j$. The remaining part of $\mathbf{S}_j$ is also good, because its weight is at most $S_j - (g - B_j) < S_j - g/2 \leq g$.

Case 3: $2S_j/g > 3$. We partition $\mathbf{S}_j$ into $\mathbf{B}_j$ and a partition of $\mathbf{T}_j$ in which all rectangles, with the possible exception of the last one, have weight equal to g. If this partition contains 3 rectangles, then we clearly satisfy $a_j < 2S_j/g$. If it contains $a_j \geq 4$ rectangles, then we have $S_j > (a_j - 2)g \geq a_j g/2$, and thus $a_j < 2S_j/g$.

Note that after the initial computation of the slices we perform only the linear scans of the slice tops, and this can be done in linear time. ❑

The RTILE and DRTILE Problems on $\{0,1\}$-arrays. Using Lemma 3.1, we can prove the following result[5].

THEOREM 3.1. *The RTILE and DRTILE problems on a $\{0,1\}$-array can be approximated to within a factor of 2 in $O(n+m)$ time.*

Proof. First, we prove the result for the RTILE problem. Assume that a given 0-1 array $\mathbf{A}$ can be partitioned into p rectangles, each of weight at most w. We apply the algorithm of Lemma 3.1 for $g = \lceil 2A/p \rceil$ which yields a partition of $\mathbf{A}$ into at most $\lceil 2A/g \rceil \leq 2A/(2A/p) = p$ good rectangles. Because $g \leq 2\lceil A/p \rceil \leq 2w$, our approximation ratio equals 2.

Next, we prove the result for the DRTILE problem. With the same assumption as before, apply the algorithm of Lemma 3.1 for $g = w$. This yields a partition of $\mathbf{A}$ into $q \leq \lceil 2A/w \rceil$ good rectangles. Because $pw \geq A$, we get $p \geq \lceil A/w \rceil \geq 1/2 \lceil 2A/w \rceil \geq 1/2 q$. ❑

[5] We recently learnt that this theorem was also proved independently in a very recent paper [LP00, Theorems 2 and 3].

The RTILE and DRTILE Problems on Arbitrary Arrays. Using the slicing technique above with a novel method of combining slices and using more elaborate accounting of the number of rectangles used, we can show the following result (the proof refers to the slicing notation above).

THEOREM 3.2. *The RTILE problem on arbitrary arrays can be approximated to within a factor of* $11/5$ *in* $O(n+m)$ *time.*

Proof. Assume that the input array $\mathbf{A}$ can be partitioned into p rectangles of weight at most w. Our goal is to find a partition into p rectangles of weights at most $11/5 w$.

Suppose that $x = W/p$ and y is the largest entry of $\mathbf{A}$. To avoid dealing with fractions, we rescale the entries of $\mathbf{A}$ so that $5 = \max(x, y)$. It is clear that after the rescaling we have $w \geq 5$. In the remaining part of this section, we will assume than no array entry exceeds 5, and that a rectangle is good if its weight does not exceed 11. Therefore it suffices to find a partition of $\mathbf{A}$ into at most $\lceil A/5 \rceil$ good rectangles, or, equivalently, a partition into q good rectangles where $5q - A < 5$.

We will partition each slice separately. If $\mathbf{S}_j$ is partitioned into a_j good rectangles, we say that $\mathbf{S}_j$ has the *deficit* of $d_j = 5a_j - S_j$. We need to partition slices in such a way that the sum of their deficits is smaller than 5.

LEMMA 3.2. *Assume that* $T_j \leq 6b + 5$ *for some positive integer* b. *Then we can partition* $\mathbf{T}_j$ *into* b *good rectangles.*

Proof. By induction on b. If $b = 1$, then $T_j \leq 11$ and the claim is obvious. Otherwise, find the longest initial good part of row $\mathbf{T}_j$. Note that the weight of this part exceeds 6, otherwise we would extend it by one more term, thus increasing its weight to at most $6 + 5$. The remaining part of $\mathbf{T}_j$ has weight at most $(b-1)6 + 5$ and by the inductive hypothesis we can partition it into $b - 1$ good parts. ❑

LEMMA 3.3. *Assume that* $T_j = 6b + 5 + y$ *for an integer* $b \geq 2$ *and a real* $y \leq 5$. *Then we can partition* $\mathbf{T}_j$ *either into* b *good rectangles, or into* $b+1$ *rectangles of weight at most* $6 + y$.

Proof. By induction on b. If $b = 2$, find the longest initial good part of $\mathbf{T}_j$. If its weight is at least $6 + y$, then the remaining part of $\mathbf{T}_j$ has weight at most $2 \times 6 + 5 + y - (6 + y) = 11$, and we succeeded to partition $\mathbf{T}_j$ into 2 good rectangles. If this weight is lower than $6 + y$, try the same with the longest final good part of $\mathbf{T}_j$. If we fail both times, then both parts of $\mathbf{T}_j$ that we have obtained have weights between 6 and $6+y$. Consequently, the remaining middle part has the weight of at most $5 + y$.

For the inductive step, find the longest initial good part of $\mathbf{T}_j$. If its weights is at least $6 + y$, then the remaining part of $\mathbf{T}_j$ can be partitioned into $b - 1$ good parts by Lemma 3.2. Otherwise, its weight is between 6 and $6 + y$ and the weight of the remaining part of $\mathbf{T}_j$ is at most $6(b - 1) + 5 + y$, so it can be partitioned in the desired fashion by the inductive hypothesis. ❑

LEMMA 3.4. *If* $S_j \geq 16$ *then we can partition the* j^{th} *slice with the deficit* $d_j \leq -1$.

Proof. We have $S_j = 6a - 2 + x$ for some integer $a \geq 3$ and a real x, $0 \leq x < 6$. By partitioning $\mathbf{S}_j$ into a good rectangles we will obtain the deficit of $5a - (6a - 2 + x) = 2 - a - x \leq -1$.

If $T_j \leq 6a - 1$ then for $b = a - 1$ we have $T_j \leq 6(b + 1) - 1 = 6b + 5$. Using Lemma 3.2 we partition $\mathbf{T}_j$ into $a - 1$ good rectangles, and $\mathbf{B}_j$ provides the a^{th} good rectangle.

In the remaining case we have a real y such that $0 < y < 5$, $T_j = 6a - 1 + y$, and $B_j = S_j - T_j = 6a - 2 + x - (6a - 1 + y) = x - y - 1$. For $b = a - 1$ we have $T_j = 6b+5+y$, thus we can partition $\mathbf{T}_j$ according to Lemma 3.3. If this partition contains $a - 1$ good rectangles, we can add $\mathbf{B}_j$ as the a^{th} good rectangle. If this partition contains a rectangles, each with weight at most $6+y$, then we can extend these rectangles vertically cover $\mathbf{B}_j$. The weight of an extended rectangle is at most $6 + y + B_j = 6 + y + x - y - 1 = 5 + x < 11$, hence each extended rectangle is good. ❑

LEMMA 3.5. *If* $T_j \leq 11$ *then we can partition* $\mathbf{S}_j$ *with the deficit* $d_j \leq -1$.

Proof. We partition $\mathbf{S}_j$ into $\mathbf{T}_j$ and $\mathbf{B}_j$. $\mathbf{B}_j$ is good in every slice, and $\mathbf{T}_j$ is good by the assumption. The deficit is $d_j = 10 - S_j < -1$. ❑

Now we will provide an algorithm that runs in time $O(n+m)$ that finds a partition of $\mathbf{A}$ into at most $\lceil A/5 \rceil$ arrays of weight at most 11. We partition the input array into slices. The slices satisfying the assumption of Lemma 3.4 or Lemma 3.5 are partitioned into good rectangles with the deficit of -1 or less. Below we describe how to handle the remaining slices.

C	D	E
F	G	H

If $T_j > 11$ and $S_j < 16$, we make a preliminary partition of $\mathbf{S}_j$ into six rectangles. First, we partition $\mathbf{T}_j$ into three rectangles, from left to right $\mathbf{C}$, $\mathbf{D}$ and $\mathbf{E}$, so that the middle rectangle $\mathbf{D}$ consists of one entry only, while $C \leq T_j/2$ and $E \leq T_j/2$. Next, we partition $\mathbf{B}_j$ into $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{H}$ so that these rectangles align as shown above.

Before we proceed, we need to observe that rectangle $\mathbf{C} \cup \mathbf{F}$ is good, and that by a symmetric argument, $\mathbf{E} \cup \mathbf{H}$ is good as well. Indeed, $C + F \leq S_j - (D + E) \leq S_j - T_j/2 < 16 - 11/2 = 10\frac{1}{2}$.

Easy Case: $\mathbf{S}_j - \mathbf{C} - \mathbf{F}$ is good, or $\mathbf{S}_j - \mathbf{E} - \mathbf{H}$ is good. We partition $\mathbf{S}_j$ into two good rectangles. Because $S_j > 11$, we have deficit $d_j < -1$.

Hard Case: The easy case does not apply. We will show in a moment that $\mathbf{D} \cup \mathbf{G}$ is good, so we can partition $\mathbf{S}_j$ into three good rectangles. In this case the deficit $d_j = 15 - S_j$ can be positive, so we may be forced to change this initial partition. Therefore we must characterize the weights of the six rectangles that are possible in the hard case.

Because the easy case does not apply, we can conclude that

$$\begin{array}{rrl} & 22 & < (S_j - C - F) + (S_j - E - H) \\ & & = (D + E + G + H) + (C + D + F + G) \\ & & = S_j + D + G \\ & & \leq S_j + 5 + B_j \\ & & < S_j + 5 + (S_j - 11) \\ \equiv & 28 & < 2S_j \\ \equiv & S_j & > 14 \end{array}$$

Therefore in the hard case we have $S_j = 14 + x$ for some $0 < x < 2$, and the deficit is $d_j = 1 - x$, $-1 < d_j < 1$.

We can also show that in hard case the following statements are true:

(a) $B_j = F + G + H < 3 + x$;
(b) $C + F < 3 + x$ and $E + H < 3 + x$;
(c) $F < 2x$ and $H < 2x$.

Inequality **(a)** follows from $B_j = S - j - T_j = 14 + x - T_j < 14 + x - 11$.

Inequality **(b)** follows from $S_j - C - F > 11$ (because the easy case does not apply) and thus $C + F < S_j - 11 = 3 + x$.

Suppose that **(c)** does not hold, w.l.o.g $F \geq 2x$; this and **(a)** implies $G < 3 - x$, thus $S_j - C - F = (C + F) + D + G < (3 + x) + 5 + (3 - x) = 11$, which means that $\mathbf{S}_j - C - F$ is good and the easy case applies.

By substituting $1 - d_j$ for x, we can rewrite **(a-c)** as follows:

(a) $B_j = F + G + H < 4 - d_j$;
(b) $C + F < 4 - d_j$ and $E + H < 4 - d_j$;
(c) $F < 2 - 2d_j$ and $H < 2 - 2d_j$.

Now for $j = 1, \ldots, l - 1$ we consider the accumulated deficit $\Delta_j = \sum_{k=1}^{j} d_k$. Our goal is to keep Δ_j small, even though some terms in this sum, those that correspond to the hard cases, may be positive. Consider the smallest j such that $\Delta_j \geq 1$. Because each deficit is lower than 1, this implies that $d_j > 0$ and $d_j + d_{j-1} > 0$.

Our strategy is that whenever $\Delta_j \geq 1$, we partition again $\mathbf{S}_{j-1} \cup \mathbf{S}_j$, this time into 5 rectangles rather than 6. This repartition subtracts 5 from the accumulated deficit, so it drops below -3. This way we never let Δ_j stay above 1 for $j < l$. To finish, we will need to account separately for the remainder slice $\mathbf{S}_l$ that can have larger deficit than 1.

In accounting for the remainder slice $\mathbf{S}_l$, consider first the case when the slice $\mathbf{S}_{l-1}$ is not an example of the hard case. Then $\Delta_{l-1} = \Delta_{l-2} + d_{l-1} < 1 - 1 = 0$, hence $\Delta_l < d_l < 5$. In turn, if $\mathbf{S}_{l-1}$ is an example of the hard case, then we can partition it into $\mathbf{C} \cup \mathbf{F}$, a rectangle of weight at most $4 - d_{l-1} < 5$, $\mathbf{E} \cup \mathbf{H}$, a rectangle with the same weight estimate, and $\mathbf{D} \cup \mathbf{G}$, a rectangle with of weight at most $5 + 4 - d_{l-1} < 10$. If $S_l \leq 1$, then we can extend these three rectangles vertically up to cover $\mathbf{S}_l$; if $S_l > 1$, then $d_l < 4$ and $\Delta_l < 1 + 4 = 5$.

To finish the proof, we return to the case where we need to partition $\mathbf{S}_{j-1} \cup \mathbf{S}_j$ again. Because $0 < d_j < 1$ and $d_{j-1} > -d_j$, both slices in question are examples of the hard case. We partition $\mathbf{S}_j$ into $\mathbf{C}$, $\mathbf{D}$ etc., and $\mathbf{S}_{j-1}$ into $\mathbf{C}'$, $\mathbf{D}'$ etc.

We must have one of the configurations shown here, or a symmetric one. In the left configuration, we can use four arrays, the left columns of four arrays forms one, the right column another, while the remaining ones are $\mathbf{D} \cup \mathbf{G}$ and $\mathbf{D}' \cup \mathbf{G}'$. The weights of these arrays can be estimated as follows:

C	D	E
F	G	H
C′	D′	E′
F′	G′	H′

C	D	E
F	G	H
C′	D′	E′
F′	G′	H′

$C + F + C' + F' < 4 - d_j + 4 - d_{j-1} = 8 - (d_j + d_{j-1}) < 8$;
$E + H + E' + H' < 8$ by a symmetric argument;
$D + G < 5 + B_j = 5 + 4 - d_j < 9$;
$D' + G' < 5 + B_{j-1} = 5 + 4 - d_{j-1} < 10$.

In the right configuration we can use five arrays. The first is $\mathbf{B}_{j-1}$. The second is $\mathbf{C} \cup \mathbf{D}$ of weight at most $4 - d_j + 5 = 9 + d_j < 9$. The third is $\mathbf{E}$. The fourth is $\mathbf{C}'$ extended upwards, to contain $\mathbf{F}$, $\mathbf{G}$ and a fragment of $\mathbf{H}$; its weight is at most $4 - d_{j-1} + B_j \leq 8 - (d_{j-1} + d_j) < 8$. The last one is $\mathbf{D}' \cup \mathbf{E}'$ extended upwards to contain the remaining fragment of $\mathbf{H}$; its weight is at most $5 + 4 - d_{j-1} + 2 - 2d_j = 11 - (d_{j-1} + d_j) - d_j < 11$. This completes the proof of Theorem 3.2. ❑

Next, we turn our attention to the DRTILE problem in d-dimensions. Using slicing techniques, and combination of two lower bounds, we can prove the following result.

THEOREM 3.3. *The DRTILE problem on arbitrary arrays in d dimensions ($d \geq 1$) can be approximated to within a factor of $2d - 1$ in $O(d(m + n))$ time.*

Proof. For simplicity of notation, we will assume, without loss of generality, that $w = 1$ (if the weight limit

is different, we simply rescale the entries). This implies that every entry of $\mathbf{A}$ is between 0 and 1. We refer to this condition as $\mathbf{A}$ being 1-*bounded.* For convenience, we define a 0-dimensional array to consist of a single entry. Let k^* be the minimum number of rectangles in a partition of the input array $\mathbf{A}$ of total weight A such that each rectangle has weight no more than 1.

For $d = 0$ our algorithm returns the input array. For dimension $d > 0$, our algorithm uses the algorithm for dimension $d-1$.

Assume that $\mathbf{B}$ is a d-dimensional array where the the i^{th} index ranges between 1 and n_i. The projection of $\mathbf{B}$ is the $(d-1)$-dimensional array $\overline{\mathbf{B}}$ defined by the equation below:

$$\bar{b}_{i_1,\ldots,i_{d-1}} = \sum_{i_d=1}^{n_d} b_{i_1,\ldots,i_{d-1},i_d}$$

The d-dimensional algorithm can be described briefly as follows:

1. We find the largest index $k_1 \leq n_d$ such that the array $\mathbf{A}_1$ formed from the entries of $\mathbf{A}$ with $i_d \in (k_0 = 0, k_1]$ satisfies the condition that $\overline{\mathbf{A}_1}$ is 1-bounded; if $k_1 < n_d$, we find the largest $k_2 \leq n_d$ such that the array $\mathbf{A}_2$ formed from entries of $\mathbf{A}$ with $i_d \in (k_1, k_2]$ satisfies the condition that $\overline{\mathbf{A}_2}$ is 1-bounded, etc. In this way we partition $\mathbf{A}$ into s *slices* $\mathbf{A}_1, \ldots, \mathbf{A}_s$.
2. For $1 \leq r \leq s$ apply the $(d-1)$-dimensional algorithm to $\overline{\mathbf{A}}_r$, the projection of the r^{th} slice.
3. For $1 \leq r \leq s$ replace each slice $\overline{\mathbf{B}}$ from the partition of $\overline{\mathbf{A}}_r$ with $\mathbf{B} = \overline{\mathbf{B}} \times [k_{r-1}+1, \ldots, k_r]$.

LEMMA 3.6. *In the notation of the above algorithm, no partition of* $\mathbf{A}$ *into tiles of weight at most* 1 *has fewer than* s *tiles.*

Proof. For $1 \leq r < s$ we can find an index vector ι_r such that the entry of $\overline{\mathbf{A}}_r$ with indexed with ι_r would exceed 1 if we would extend the range of the last index in $\overline{\mathbf{A}}_r$ from $(k_{r-1}, k_{r+1}]$ to $(k_{r-1}, k_{r+1}+1]$ (this follows from way the algorithm defines k_r's). Define ι_s as an arbitrary index vector from the proper range. Now we define b_r to be the entry of $\mathbf{A}$ with index vector $(\iota_r, k_{r-1}+1)$. It is easy to see that no two b_r's can belong to the same tile. ❑

LEMMA 3.7. *If the algorithm produces a partition of* $\mathbf{A}$ *into* t *tiles, then the weight of* $\mathbf{A}$ *satisfies* $2dA \geq t-1$.

Proof. By induction on d. For $d = 0$ the claim is trivial, so it suffices to prove the inductive step.

Assume that the r^{th} slice was partitioned into t_r tiles. By applying the inductive hypothesis to the projections of the slices of $\mathbf{A}$ we obtain

$$2(d-1)A = 2(d-1)\sum_{r=1}^{s} \overline{A}_r \geq \sum_{r-1}^{s}(t_r - 1) = t - s$$

On the other hand, the sum of weights of every consecutive pair of slices exceeds 1; otherwise the projection of their union would be 1-bounded, a contradiction. Thus

$$2A > s - 1$$

We obtain the claim by adding these two inequalities. ❑

Now, we obtained the following inequalities: $t - s \leq 2(d-1)A \leq 2(d-1)k^*$ and $s \leq k^*$. The approximation bound of our theorem follows form adding these two inequalities together.

Now we prove the time-complexity of our algorithm. Since after Step 1 the projections of different slices are disjoint, it suffices to show how to find the indices $k_1, k_2, \ldots, k_s$ and the projections of all the slices in $O(m+n)$ time, assuming that $\mathbf{A}$ is given in its standard sparse representation as described in Section 2. For each index j in the d^{th} dimension of $\mathbf{A}$ $(1 \leq j \leq n_d)$, let L_j be the list (in any arbitrary order) of tuples $(i_1, i_2, \ldots, i_{d-1}, j\ ;\ \mathbf{A}[i_1, i_2, \ldots, i_{d-1}, j])$. The lists $L_1, L_2, \ldots, L_n$ can be computed in a total $O(n+m)$ time simply by traversing each of the m $(d+1)$-tuples of $\mathbf{A}$ and inserting these tuples in the appropriate L_j based on their d^{th} coordinate. Since the $(d+1)$-tuples of $\mathbf{A}$ are given in a lexicographically sorted order on their first d coordinates, it is easy to find in $O(m)$ time the at most m $(d-1)$-tuples $(i_1, i_2, \ldots, i_{d-1})$, where for every such tuple there is some j and x such that $(i_1, i_2, \ldots, i_{d-1}, j\ ;\ x)$ is a $(d+1)$-tuple of $\mathbf{A}$. Next, we maintain an m-element array $\mathbf{B}$ corresponding to these at most m $(d-1)$-tuples. Initially, $\mathbf{B}[(i_1, i_2, \ldots, i_{d-1})] = 0$ for every tuple $(i_1, i_2, \ldots, i_{d-1})$. We examine the lists $L_1, L_2, L_3, \ldots$ in order and for each entry $(i_1, i_2, \ldots, i_{d-1}, j\ ;\ x)$ in L_j, we increment $\mathbf{B}[(i_1, i_2, \ldots, i_{d-1})]$ by x. We stop as soon as we arrive at an index j such that, after the appropriate L_j has been processed, $\mathbf{B}[(i_1, i_2, \ldots, i_{d-1})] > 1$ for some $(i_1, i_2, \ldots, i_{d-1})$. Then, k_1 is $j-1$. We reset $\mathbf{B}[(i_1, i_2, \ldots, i_{d-1})] = 0$ for all i and continue the same procedure starting from list L_j. This takes a total of $O(n+m)$ time. It is easy to compute the projections of each slice also within the same time bound. This completes the proof of Theorem 3.3. ❑

4 Approximating d-RPACK

In this section we first give a simple, efficient approximation algorithm for d-RPACK with approximation ratio $\lceil \log(n+1) \rceil^{d-1}$, and then show how to further improve the performance ratio of the algorithm at the expense of increasing the running time. We consider a slightly more *general version* of the d-RPACK problem where $D_1, D_2, \ldots, D_p$ are the p weighted hyper-rectangles, with $D_i = \times_{j=1}^{d}[b_{i,j}, e_{i,j}]$ having its endpoints $b_{i,j}, e_{i,j} \in \{1, 2, \ldots, n_j\}$, that is, j^{th} dimension of the hyper-rectangles has its own range $\{1, 2, \ldots, n_j\}$. We assume, without loss of generality, that $p = \Omega(n_1)$.

4.1 $(\lfloor 1 + \log n \rfloor)^{d-1}$-approximation algorithm for d-RPACK

For a d-dimensional hyper-rectangle D_i, let $D_i + \alpha$ denote the d-dimensional hyper-rectangle $\times_{j=1}^{d}[b_{i,j} + \alpha, e_{i,j} + \alpha]$ and $D_i^{-\beta}$ denote the $(d-\beta)$-dimensional hyper-rectangle $\times_{j=1}^{d-\beta}[b_{i,j}, e_{i,j}]$ for $0 < \beta < d$. Extend these notations to an arbitrary set X of d-dimensional hyper-rectangles by defining $X + \alpha = \{x + \alpha \,|\, x \in X\}$ and $X^{-\beta} = \{x^{-\beta} \,|\, x \in X\}$. For a set of hyper-rectangles X, let $|X|$ denote the number of hyper-rectangles in X.

THEOREM 4.1. *The general version of the d-RPACK problem can be approximated to within a factor of $\prod_{i=2}^{d}(\lfloor 1 + \log n_i \rfloor)$ of the optimum in $O(p(d-1)\log^{\varepsilon} p + \sum_{i=2}^{d} n_i \frac{\log n_i}{\log\log n_i} + pk) = O(dp\log^{\varepsilon} p + \sum_{i=1}^{d} n_i \frac{\log n_i}{\log\log n_i} + pk)$ time (for any $0 < \varepsilon \leq 1$).*

Setting $n_i = n$ for all i in Theorem 4.1, we obtain:

COROLLARY 4.1. *The d-RPACK problem can be approximated to within a factor of $(\lfloor 1 + \log n \rfloor)^{d-1}$ of the optimum in $O(dp\log^{\varepsilon} p + n\frac{\log n}{\log\log n} + pk)$ time (for any $0 < \varepsilon \leq 1$).*

Proof-sketch of Theorem 4.1. We note that the 1-RPACK problem, which we call as the *Interval Packing* (IPACK) problem, is easily solvable in $O(pk)$ time via dynamic programming technique[6]. For notational convenience, let $q = \lfloor 1 + \log n_d \rfloor$. We prove the result by induction on d. For $d = 1$, the result follows directly via solution of IPACK. Inductively, assume that we have an algorithm for $(d-1)$-RPACK and consider the d-RAPCK problem on the set of hyper-rectangles R. We divide R, in $O(p\log^{\varepsilon} p + n_d\frac{\log n_d}{\log\log n_d})$ time, into at most q disjoint collections $R_1, R_2, \ldots, R_q$ such that an optimum solution of the d-RPACK problem on each R_i is also an optimum solution of the $(d-1)$-RPACK problem on R_i^{-1}. By inductive hypothesis, we approximate the $(d-1)$-RPACK problem on each R_i^{-1} separately within a factor of $\prod_{i=2}^{d-1}(\lfloor 1 + \log n_i \rfloor)$ of the optimum for each R_i^{-1}. By pigeonhole principle, the best of all these solutions is within a factor of $q\prod_{i=2}^{d-1}(\lfloor 1 + \log n_i \rfloor) = \prod_{i=2}^{d}(\lfloor 1 + \log n_i \rfloor)$ of the optimum solution, and the total time taken is $O(\sum_{i=1}^{q}|R_i|(d-2)\log^{\varepsilon}|R_i| + \sum_{i=2}^{d-1} n_i\frac{\log n_i}{\log\log n_i} + |R_i|k + p\log^{\varepsilon} p + n_d\frac{\log n_d}{\log\log n_d}) = O(p(d-1)\log^{\varepsilon} p + \sum_{i=2}^{d} n_i\frac{\log n_i}{\log\log n_i} + pk) = O(dp\log^{\varepsilon} p + \sum_{i=1}^{d} n_i\frac{\log n_i}{\log\log n_i} + pk)$ time.[7]

Here are the ideas behind how to find these sets $R_1, R_2, \ldots, R_q$. For a set X of d-dimensional hyper-rectangles and a number a, let us define the following subsets of X:

$$\begin{aligned} \mathit{Int}(a, X) &= \{r | r = \times_{j=1}^{d}[b_j, e_j] \in X, b_d \leq a \leq e_d\} \\ \mathit{Above}(a, X) &= \{r | r = \times_{j=1}^{d}[b_j, e_j] \in X, a < b_d\} \\ \mathit{Below}(a, X) &= \{r | r = \times_{j=1}^{d}[b_j, e_j] \in X, e_d < a\} \end{aligned}$$

To build the sets $R_1, R_2, \ldots, R_q$, we will first divide the given set of p rectangles into n disjoint groups $S_1, S_2, \ldots, S_{n_d}$ (some of which may be empty), and combine these groups to get $R_1, \ldots, R_q$. For clarity, when necessary we will indicate a set S_t as $S_t(i_t, j_t)$ to emphasize that endpoints of any hyper-rectangle $\times_{j=1}^{d}[b_j, e_j] \in S_t(i_t, j_t)$ satisfy $b_d, e_d \in \{i_t, i_t + 1, \ldots, j_t\}$. Each set S_i also has a *state*, which can be *varying* meaning that the content of S_i may be subject to further modifications, or *fixed* meaning that the content of S_i will not be subject to any further modification. Modification of the content of a set will also be termed as *refining* the set. We start by setting $S_1(1, n_d) = R$, $S_2 = S_3 = \cdots = S_n = \emptyset$, and setting the state of each set S_i as varying. Next, we refine the set S_1, S_2 and S_3 by setting $S_1(1, n_d) = \mathit{Int}(\lceil (n_d + 1)/2 \rceil, S_1(1, n_d))$, $S_2(1, \lceil (n_d + 1)/2 \rceil - 1) = \mathit{Below}(\lceil (n_d + 1)/2 \rceil, S_1(1, n_d))$ and $S_3(\lceil (n_d + 1)/2 \rceil + 1, n_d) = \mathit{Above}(\lceil (n_d+1)/2 \rceil, S_1(1, n_d))$, and set the state of $S_1(1, n_d)$ as fixed. It is shown in the full version of the paper that the d-RPACK problem on S_1 is precisely the $(d-1)$-RPACK problem on S_1^{-1}. We iteratively refine $S_2(1, \lceil (n_d + 1)/2 \rceil - 1)$ and $S_3(\lceil (n_d + 1)/2 \rceil + 1, n_d)$ in a similar manner; complete details are available in the full version of the paper. ❑

4.2 Further Improved Approximation Ratio for d-RPACK

THEOREM 4.2. *For every $L \geq 2$, the general version of the d-RPACK problem can be approximated to within a factor of $\prod_{i=2}^{d}(\lfloor 1 + \log_L n_i \rfloor)$ of the optimum in $O(p^{(L-1)^{d-1}+1}\, k)$ time.*

[6] Since the endpoints of the p intervals are in $\{1, 2, \ldots, n_1\}$, they can be sorted in $O(p + n_1) = O(p)$ time

[7] Here, we use the simple fact that when we reach the 1-RPACK problems at the base level of recursion, all the endpoints of p intervals divided into at most $\min\{p, \prod_{i=2}^{d}(\lfloor 1 + \log n_i \rfloor)\}$ groups can be sorted together in $O(p)$ time.

Proof. In the following discussions, each rectangle r has the following attributes: weight $w(r)$ and, for coordinate $i \in [1,d]$, starting and terminating values $s_i(r)$ and $t_i(r)$. Such a rectangle is a set of integer vectors $\times_{i=1}^{d}[s_i(r),t_i(r)]$. The objective again is to find a subset S of the set of given rectangles such that its elements are pairwise disjoint, $|S| \leq k$ and its total weight, $\sum_{r\in S} w(r)$ is maximum.

To formulate our approximation algorithm, we will first define a very special subproblem which has a polynomial time solution. An instance of d-RPACK is A-restricted if $A \subset \times_{i=2}^{d}[1,n_i]$ and each given rectangle intersects the set $[1,n_1]\times A$.

We will show an *exact* algorithm for A-restricted d-RPACK that runs in time $O(p^{|A|+1}k)$.

We first order the given rectangles in such a way that if $r \preceq r'$ then $s_1(r) \leq s_1(r')$. Consider a legal solution S—the rectangles in S are disjoint and $|S| \leq k$. We define the j^{th} element of S to be $r \in S$ such that $|\{r' \in S : \ r' \preceq r\}| = j$. Next, if r is the j^{th} element of S than the j^{th} cut of S is the following subset: $\{r' \in S : \ r' \preceq r \ \wedge \ t_1(r') \geq s_1(r)\}$. If $j = |S|$, then the j^{th} element of S is the *terminal element* of S. We define the *terminal cut* of S similarly.

LEMMA 4.1. *If S is a legal solution with at least j elements, then the j^{th} cut of S has at most $|A|$ elements.*

Proof. Let us order the set A. The first $a \in A$ such that $r \cap [1,n_1]\times\{a\} \neq \emptyset$ will be denoted $\alpha(r)$. Because our input is A restricted, $\alpha(r)$ is defined for every input rectangle.

Now assume that r is the j^{th} element of S, $s = s_1(r)$ and T is the j^{th} cut of S. Consider $r' \in T$. Because $r' \leq r$ we have $s_1(r') \leq s$. Because $r' \in T$, we have $t_1(r') \geq s$. Thus $s \in [s_1(r'),t_1(r')]$.

Now consider $r', r'' \in T$ such that $\alpha(r') = \alpha(r'') = a$. Then $s \times a \in r' \cap r''$. Because rectangles in T are disjoint, we can conclude that $r' = r''$. Thus $\alpha : T \xrightarrow{1-1} A$ which shows that $|T| \leq |A|$. ❑

LEMMA 4.2. *Assume that S is a legal solution, $|S| < k$, r is the terminal element of S and $r \prec r'$. Then $S \cup \{r'\}$ is a legal solution if and only if r' does not intersect any rectangles from the terminal cut of S.*

Proof. Let T be the terminal cut of S. It suffices to show that r' does not intersect any rectangles from $S - T$. Because every rectangle in S precedes r, $r'' \in S - T$ only if $t_1(r'') < s_1(r)$. Because $r \prec r'$, $s_1(r) \leq s_1(r')$. Thus $t_1(r'') < s_1(r')$, which implies that $r' \cap r'' = \emptyset$. ❑

Now we can define a *plausible j^{th} cut of a legal solution* as a set of rectangles C such that $|C| \leq j$ and C is its own terminal cut. We also say that $\emptyset$ is a plausible 0^{th} cut of a legal solution. We define of *nodes* to be pairs (j,S) such that S is a plausible j^{th} cut of a legal solution. A pair $((j,S),(j+1,S'))$ forms an *edge* if it is plausible that for some legal solution S is the j^{th} cut and S' is the $(j+1)^{\text{st}}$ cut. More formally, S is a plausible j^{th} cut, for every $r \in S$ we have $r \prec r'$ and S' is the terminal cut of $S \cup \{r'\}$. The *weight* of this edge is equal to $w(r')$. One can see that there is 1-1 correspondence between legal solutions and the paths in this graph that start at $(0,\emptyset)$. The optimum solution corresponds to the longest path. Because this is an acyclic graph, we can find the longest path in time proportional to the number of edges.

It is easy to see that we have less than $p^{|A|}/|A|! \times k$ nodes, and that out-degree of a node is lower than p. This finishes the sketch and analysis of the exact algorithm for A-restricted d-RPACK. ■

Our approximation algorithm has the following idea. We partition the input set into $\prod_{i=2}^{d}\lfloor 1+\log_L n_i\rfloor$ subsets, and then we find an exact solution for each subset.

We start from partitioning the range $[1,n_i]$ into $\lfloor 1+\log_L n_i\rfloor$ subsets. We say that the j^{th} lattice is the set of integers that are divisible by L^j but are not divisible by L^{j+1}.

Next, we say that an interval $[s,t]$ is of class j if it contains numbers from the j^{th} lattice, but it does not contain any number from $(j+1)^{\text{st}}$ lattice. In such a case we write $class(s,t) = j$. It is easy to see that $class(s,t) \leq \log_L t$.

We partition our input set of rectangle into equivalence classes of the following relation: $r \bowtie r'$ if an only if $\bigwedge_{i=2}^{d} class(s_i(r),t_i(r)) = class(s_i(r'),t_i(r'))$. Because $t_i(r) \leq n_i$, there are at most $\times_{i=2}^{d}\lfloor 1+\log_L n_i\rfloor$ many classes. It remains to show how to reduce the problem of finding an optimum solution for a single class. Consider a class C such that $r \in C \equiv \bigwedge_{i=2}^{d} class(s_i(r),t_i(r)) = j_i$. First we find the connected components of C in respect to $r \cap r' \neq \emptyset$. Observe for each component D, and $r, r' \in D$ the following holds for $i \in [2,d]$

$$\begin{aligned} & [s_i(r),t_i(r)] \cap [s_i(r'),t_i(r)] = \emptyset \\ \equiv \ & [s_i(r) \ \mathbf{mod}\ L^{j_i+1}, t_i(r) \ \mathbf{mod}\ L^{j_i+1}] \\ & \quad \cap [s_i(r') \ \mathbf{mod}\ L^{j_i+1}, t_i(r) \ \mathbf{mod}\ L^{j_i+1}] \end{aligned}$$

Therefore the problems remains equivalent if we transform every rectangle using **mod** operations as describe above, and then translate each component along x_1 axis to assure that the rectangles from different components are still disjoint.

We define $A_i = \{aL^{j_i} : \ a \in [1,L-1]\}$. After the last transformation set D becomes A-restricted for $A = \times_{i=2}^{d} A_i$. ❑

Setting $L = 2^c$ and $n_j = n$ for all j, we obtain:

COROLLARY 4.2. *For every $c \geq 1$, there is an approximation algorithm for d-RPACK that runs in $O(p^{(2^c-1)^{d-1}+1}\ k)$ time with an approximation ratio of $\left(\left\lfloor 1 + \frac{\log n}{c} \right\rfloor\right)^{d-1}$.*

The fastest version of this algorithm uses $L = 2$, and in this case we have $|A| = 1$ in the proof. For larger values of L our estimate for the running time increases drastically, and the result may seem to be of only theoretical interest. However, in a practical instance, our estimate can be unduly pessimistic, since the number of different plausible cuts is likely to be much smaller than our worst case bound of the number of all subsets of the given set of rectangles that have at most $|A|$ elements. Moreover, in different dimensions we can use different values of L. The overall effect of these engineering optimizations may be a practical algorithm with an approximation ratio better than the one in Section 4.1.

5 Concluding Remarks

A general open issue is whether the approximation bounds in this paper can be further improved. It is known that it is NP-hard to approximate the RTILE problem to a ratio better than $5/4$ [KMP98], hence a gap still remains. Our algorithm for the RTILE problem for the $\{0,1\}$-arrays[8] in fact shows that there is a hierarchical partition of the given array in which the maximum weight of a tile is at most $2\lceil A/p \rceil$ since we used $\lceil A/p \rceil$ as a (trivial) lower bound on the optimum. It is easy to generate examples such that no tiling exists with the maximum tile of weight at most $5/3\lceil W/p \rceil$. Hence, our approach for tiling on $\{0,1\}$-arrays will not yield better than $5/3$ approximation unless new lower bound techniques are used. We used one such new lower bound for the DRTILE problem in the proof of Theorem 3.3 (in Lemma 3.6) to get the improved approximation ratio of $2d-1$.

Initially, we suspected that $(\lfloor 1 + \log n \rfloor)^{d-1}$ is a lower bound on the approximation ratio that is attainable for the d-RPACK problem. However, to our surprise, we were able to obtain an approximation ratio which can be better by any arbitrary constant factor in Section 4.2. It remains open to understand the limits on the approximability of this problem, in particular, in d dimensions. Also, whether the ideas here will prove useful in applications domains remains to be seen.

References

[BNR96] V. Bafna, B. Narayanan and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis parallel rectangles). *Discrete Applied Mathematics*, 41–53, 1996.

[FM+96a] T. Fukuda, Y. Morimoto, S. Morishita and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, Algorithms and Visualization. *Proc. ACM SIGMOD*, 1996.

[FM+96b] T. Fukuda, Y. Morimoto, S. Morishita and T. Tokuyama. Mining optimized association rules for numerical attributes. *Proc. ACM Principles of Database Systems*, 1996.

[FPT81] R. Fowler, M. Paterson, and S. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Proc. Letters*, 12, 133–137, 1981.

[KMP98] S. Khanna, S. Muthukrishnan, and M. Paterson. Approximating rectangle tiling and packing. *Proc Symp. on Discrete Algorithms (SODA)*, 384–393, 1998.

[K80] R. P. Kooi. *The optimization of queries in relational databases.* PhD thesis, Case Western Reserve University, Sept 1980.

[LP00] K. Lorys and K. Paluch. Rectangle Tiling. *Third International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2000)*, Lecture Notes in Computer Science 1913, 206–213, Sept. 2000.

[MD88] M. Muralikrishna and David J Dewitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. *Proc. of ACM SIGMOD Conf*, pages 28–36, 1988.

[MPS99] S. Muthukrishnan, V. Poosala and T. Suel. On rectangular partitions in two dimensions: Algorithms, complexity and applications. *Intl. Conf. Database Theory* (ICDT), 236-256, 1999.

[O88] M. H Overmars. Computational geometry on a grid: an overview, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F40, Edited by R. A. Earnshaw, Springer-Verlag Berlin Heidelberg, 167-184, 1988.

[P97] V. Poosala. *Histogram-based estimation techniques in databases.* PhD thesis, Univ. of Wisconsin-Madison, 1997.

[RS99] R. Rastogi and K. Shim Mining optimized support rules for numerical attributes. *Proc. Intl Conf. Data Engineering*, 1999.

[S99] J. P. Sharp. Tiling Multi-Dimensional Arrays. *Foundations of Computing Theory*, 1999.

[SS99] A. Smith and S. Suri. Rectangular tiling in multi-dimensional arrays. *Proc. ACM-SIAM Symp on Discrete Algorithms* (SODA), 1999.

[8] The same effect happens with arbitrary arrays as well.