

Approximation Algorithms for Min-Max Generalization Problems¹

Piotr Berman, Pennsylvania State University
Sofya Raskhodnikova, Pennsylvania State University

We provide improved approximation algorithms for the *min-max generalization problems* considered by Du, Eppstein, Goodrich, and Lueker [Du et al. 2009]. Generalization is widely used in privacy-preserving data mining and can also be viewed as a natural way of compressing a dataset. In min-max generalization problems, the input consists of data items with weights and a lower bound w_{lb} , and the goal is to partition individual items into groups of weight at least w_{lb} , while minimizing the maximum weight of a group. The rules of legal partitioning are specific to a problem. Du et al. consider several problems in this vein: (1) partitioning a graph into connected subgraphs, (2) partitioning unstructured data into arbitrary classes and (3) partitioning a 2-dimensional array into contiguous rectangles (subarrays) that satisfy the above weight requirements.

We significantly improve approximation ratios for all the problems considered by Du et al., and provide additional motivation for these problems. Moreover, for the first problem, while Du et al. give approximation algorithms for specific graph families, namely, 3-connected and 4-connected planar graphs, no approximation algorithm that works for all graphs was known prior to this work.

Categories and Subject Descriptors: C.2.1 [Discrete Mathematics]: Combinatorial Algorithms; C.2.2 [Discrete Mathematics]: Graph Algorithms

General Terms: Graph Algorithms, Approximation Algorithms

Additional Key Words and Phrases: Generalization problems, k -anonymity, bin covering, rectangle tiling

ACM Reference Format:

Piotr Berman and Sofya Raskhodnikova, 2013. Approximation Algorithms for Min-Max Generalization Problems. *ACM V*, N, Article A (January YYYY), 23 pages.
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

We provide improved approximation algorithms for the *min-max generalization problems* considered by Du, Eppstein, Goodrich, and Lueker [Du et al. 2009]. In min-max generalization problems, the input consists of data items with weights and a lower bound w_{lb} , and the goal is to partition individual items into groups of weight at least w_{lb} , while minimizing the maximum weight of a group. The rules of legal partitioning are specific to a problem. Du et al. consider several problems in this vein: (1) partitioning a graph into connected subgraphs, (2) partitioning unstructured data into arbitrary classes and (3) partitioning a 2-dimensional array into contiguous rectangles (subarrays) that satisfy the above weight

¹A preliminary version of this article appeared in the proceedings of APPROX 2010 [Berman and Raskhodnikova 2010].

This work is supported by the National Science Foundation, under grant CCF-0729171 and CAREER grant CCF-0845701.

Authors' address: Computer Science and Engineering Department, Pennsylvania State University, USA.

Authors' email: {berman, sofya}@cse.psu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0000-0000/YYYY/01-ARTA \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

requirements. We call these problems (1) MIN-MAX GRAPH PARTITION, (2) MIN-MAX BIN COVERING and (3) MIN-MAX RECTANGLE TILING.

Du et al. motivate the min-max generalization problems by applications to privacy-preserving data mining. Generalization is widely used in the data mining community as means for achieving k -anonymity (see [Ciriani et al. 2008] for a survey). Generalization involves replacing a value with a less specific value. To achieve k -anonymity each record should be generalized to the same value as at least $k - 1$ other records. For example, if the records contain geographic information (such as GPS coordinates), and the plane is partitioned into axis-parallel rectangles each containing locations of at least k records, to achieve k -anonymity the coordinates of each record can be replaced with the corresponding rectangle. Generalization can also be viewed as a natural way of compressing a dataset.

We briefly discuss several other applications of generalization. Geographic Information Systems contain very large data sets that are organized either according to the (almost) planar graph of the road network, or according to geographic coordinates (see, e.g., [Garcia et al. 1998]). These sets have to be partitioned into *pages* that can be transmitted to a mobile device or retrieved from secondary storage. Because of the high overhead of a single transmission/retrieval operation, we want to ensure that each page satisfies the minimum size requirement, while controlling the maximum size. When the process that is exploring a graph needs to investigate a node whose information it has not retrieved yet, it has to request a new page. Therefore, pages are more useful if they contain information about connected subgraphs. MIN-MAX GRAPH PARTITION captures the problem of distributing information about the graph among pages.

MIN-MAX BIN COVERING is a variant of the classical BIN COVERING problem. In the classical version, the input is a set of items with positive weights and the goal is to pack items into bins, so that the number of bins that receive items of total weight at least 1 is maximized (see [Assmann et al. 1984; Csirik et al. 2001; Jansen and Solis-Oba 2003] and references therein). Both variants are natural. For example, when Grandfather Frost² partitions presents into bundles for kids, he clearly wants to ensure that each bundle has items of at least a certain value to make kids happy. Grandfather Frost could try to minimize the value of the maximum bundle, to avoid jealousy (MIN-MAX BIN COVERING), or to maximize the number of kids who get presents (classical BIN COVERING). MIN-MAX BIN COVERING can also be viewed as a variant of scheduling on parallel identical machines where, given n jobs and their processing times, the goal is to schedule them on m identical parallel machines while minimizing *makespan*, that is, the maximum time used by any machine [Graham et al. 1979]. In our variant, the number of machines is not given in advance, but instead, there is a lower bound on the processing time. This requirement is natural, for instance, when “machines” represent workers that must be hired for at least a certain number of hours.

Rectangle tiling problems with various optimization criteria arise in applications ranging from databases and data mining to video compression and manufacturing, and have been extensively studied [Manne 1993; Khanna et al. 1998; Sharp 1999; Smith and Suri 2000; Muthukrishnan et al. 1999; Berman et al. 2001; Berman et al. 2002; 2003]. The min-max version can be used to design a Geographic Information System, described above. If the data is a set of coordinates specifying object positions, as opposed to a road network, we would like to partition it into pages that correspond to rectangles on the plane. As before, we would like to ensure that pages have at least the minimum size while controlling the maximum size.

²Grandfather Frost is a secular character that played the role of Santa Claus for Soviet children. The Santa Claus problem [Bansal and Sviridenko 2006] is not directly related to the Grandfather Frost problem. In the Santa Claus problem, each kid has an arbitrary value for each present, and the Santa’s goal is to distribute presents in such a way that the least lucky kid is as happy as possible.

1.1. Problems

In each of the problems we consider, the input is an item set \mathcal{I} , non-negative weights w_i for all $i \in \mathcal{I}$ and a non-negative bound w_{lb} . For $\mathcal{I}' \subseteq \mathcal{I}$, we use $w(\mathcal{I}')$ to denote $\sum_{i \in \mathcal{I}'} w_i$. Each problem below specifies a class of *allowed* subsets of \mathcal{I} . A valid solution is a partition P of \mathcal{I} into allowed subsets such that $w(\mathcal{I}') \geq w_{\text{lb}}$ for each $\mathcal{I}' \in P$. The goal is to minimize the cost of P , defined as $\max_{\mathcal{I}' \in P} w(\mathcal{I}')$.

In MIN-MAX GRAPH PARTITION, \mathcal{I} is the vertex set V of an (undirected) graph (V, E) , and a subset of V is allowed if it induces a connected subgraph. In MIN-MAX BIN COVERING, every subset of \mathcal{I} is allowed. A partition of \mathcal{I} is called a *packing*, and the parts of a partition are called *bins*. In MIN-MAX RECTANGLE TILING, $\mathcal{I} = \{1, \dots, m\} \times \{1, \dots, n\}$, and the allowed sets are rectangles, i.e., sets of the form $\{a, \dots, b\} \times \{c, \dots, d\}$. A partition of \mathcal{I} into rectangles is called a *tiling*.

All three min-max problems above are NP-complete. Moreover, if $P \neq \text{NP}$ no polynomial time algorithm can achieve an approximation ratio better than 2 for MIN-MAX BIN COVERING (and hence for MIN-MAX GRAPH PARTITION) or better than 1.33 for MIN-MAX GRAPH PARTITION on 3-connected planar graphs and MIN-MAX RECTANGLE TILING [Du et al. 2009].

1.2. Our Results and Techniques

Our main technical contribution is a 3-approximation algorithm for MIN-MAX GRAPH PARTITION. The remaining algorithms are very simple, even though the analysis is non-trivial.

MIN-MAX GRAPH PARTITION. We present the first polynomial time approximation algorithm for MIN-MAX GRAPH PARTITION. Du et al. gave approximation algorithms for specific graph families, namely, a 4-approximation for 3-connected and a 3-approximation for 4-connected planar graphs. We design a 3-approximation algorithm for the general case, simultaneously improving the approximation ratio and applicability of the algorithm. We also improve the approximation ratio for 4-connected planar graphs from 3 to 2.5.

Our 3-approximation algorithm for MIN-MAX GRAPH PARTITION constructs a 2-tier partition where nodes are partitioned into *groups*, and groups are partitioned into *super-groups*. Intuitively, supergroups represent parts in a legal partition, while groups represent (nearly) indivisible subparts. There are three types of supergroups (viewed as graphs on groups): *group-pairs*, *triangles* and *stars*. The initial 2-tier partition is obtained greedily and then transformed using 4 carefully designed transformations until all stars with more than 3 groups are *structured*: meaning that each of them has a well-defined central node, and all noncentral groups in those stars are only adjacent to central nodes (possibly of multiple stars). All other supergroups have weight at most 3 and are used as parts in the final solution. Structured stars are more tricky to deal with. They are reorganized into

Table 1. Approximation Ratios for Min-Max Generalization Problems. See Theorems 2.1 and 2.2 for tighter statements on MIN-MAX GRAPH PARTITION. (Note: MIN-MAX GRAPH PARTITION generalizes MIN-MAX BIN COVERING, and hence inherits its inapproximability.)

Min-Max Problem	Hardness [Du et al. 2009]	Ratio in [Du et al. 2009]	Our ratio
GRAPH PARTITION	2	—	3
on 3-connected planar graphs	1.33	4	
on 4-connected planar graphs	—	3	
BIN COVERING	2	$2 + \varepsilon$ in time exp in ε^{-1}	2
RECTANGLE TILING	1.33	5	4
with 0-1 entries	—	—	3

new supergroups: their central groups remain unchanged while the remaining groups are redistributed using a scheduling algorithm for GENERALIZED LOAD BALANCING (GLB). Roughly, central nodes play a role of machines and the groups that we need to redistribute play a role of jobs to be scheduled on these machines. A group can be "scheduled" on a certain machine only if it is adjacent to the central node corresponding to the machine. The fact that noncentral groups are adjacent *only* to central nodes of structured stars implies that their nodes have to be distributed among parts containing central nodes in *every* legal partition. This allows us to prove that the parts produced by the scheduling algorithm have weight at most 3 times the optimum, even though we cannot give an absolute bound on the weight of each part. The factor 3 comes from two sources: the scheduling algorithm we use gives a 2-approximation for partitioning central nodes and noncentral groups, and we increase the weight of each part by at most 1 by adding noncentral nodes from the corresponding central group. The final part of the algorithm repairs parts of insufficient weight to obtain the final partition.

The scheduling algorithm used to repartition high-weight supergroups in the 2-tier partition is a 2-approximation for GLB, presented in [Kleinberg and Tardos 2006] and based on the algorithm of Lenstra, Shmoys and Tardos [Lenstra et al. 1990] for SCHEDULING ON UNRELATED PARALLEL MACHINES. In GLB, the input is the set M of m parallel machines, the set J of n jobs, and for each job $j \in J$, the processing time t_j and the set M_j of machines on which the job j can be scheduled. The goal is to schedule each job on one of the machines while minimizing the makespan. Our use of the scheduling algorithm is *gray-box* in the following sense: our algorithm runs the scheduling algorithm in a black-box manner. However, in the analysis, we look inside the black box. Namely, we consider the linear programming relaxation of GLB used in the approximation algorithm presented in [Kleinberg and Tardos 2006], and show that a valid solution of MIN-MAX GRAPH PARTITION corresponds to a feasible solution to that linear program. (Notably, a valid solution of MIN-MAX GRAPH PARTITION does not necessarily correspond to a solution of GLB.) Then we apply (a straightforward strengthening of) Theorem 11.33 in [Kleinberg and Tardos 2006] (a specialization of the Rounding Theorem of Lenstra et al. to the case of GLB) to show that the LP used by their algorithm yields a good solution for our problem.

For partitioning 4-connected planar graphs, following Du et al., we use the fact that such graphs have Hamiltonian cycles [Tutte 1956] which can be found in linear time [Chiba and Nishizeki 1989]. Our algorithm is simple and efficient: It goes around the Hamiltonian cycle and greedily partitions the nodes, starting from the lightest contiguous part of the cycle that satisfies the weight lower bound. If the last part is too light, it is combined with the first part. Thus, the algorithm runs in linear time. Our algorithm and analysis apply to any graph that contains a Hamiltonian cycle which can be computed efficiently or is given as part of the input.

MIN-MAX BIN COVERING. We present a simple 2-approximation algorithm that runs in linear time, assuming that the input items are sorted by weight³. Du et al. gave a schema with approximation ratio $2 + \varepsilon$, and time complexity exponential in ε^{-1} . They also showed that approximation ratio better than 2 cannot be achieved in polynomial time unless $P=NP$. Thus, we completely resolve the approximability of this problem.

Our algorithm greedily places items in the bins in the order of decreasing weights, and then redistributes items either in the last three bins or in the first and last bins.

MIN-MAX RECTANGLE TILING. We improve the approximation ratio for this problem from 5 to 4. We can get a better ratio of 3 when the entries in the input array, representing the weights of the items, are restricted to be 0 or 1. This case covers the scenarios where each

³This assumption can be eliminated at the expense of a more complicated argument.

entry indicates the presence or absence of some object, as in applications with geographic data, such as GPS coordinate data originally considered by Du et al.

Our algorithm builds on the *slicing and dicing* method introduced by Berman et al. [Berman et al. 2002]. The idea is to first partition the rectangle horizontally into *slices*, and then partition slices vertically. The straightforward application of *slicing and dicing* gives ratio 5. We improve it by doing simple preprocessing. For the case of 0-1 entries, the preprocessing step is more sophisticated. It splits the rectangle into many subrectangles, amenable to slicing and dicing.

Summary and Organization. We summarize our results in Table. I. The results on MIN-MAX GRAPH PARTITION are stated in Theorems 2.1 and 2.2 in Section 2, on MIN-MAX BIN COVERING, in Theorem 3.1 in Section 3, and on MIN-MAX RECTANGLE TILING, in Theorems 4.2 and 4.3 in Section 4. Theorems 2.1 and 2.2 give tighter upper bounds on the solution cost than shown in the table.

Terminology and Notation. Here we describe terminology and notation common to all technical sections. We use *opt* to denote the cost of an optimal solution. Recall that w_{lb} is a lower bound on the weight of each part and is given as part of the input. By definition, $\text{opt} \geq w_{\text{lb}}$ for all min-max generalization problems.

Definition 1.1. An item (or a set of items) is *fat* if it has weight at least w_{lb} , and *lean* otherwise. We apply this terminology to nodes and sets of nodes in an instance of MIN-MAX GRAPH PARTITION, and to elements and rectangles in MIN-MAX RECTANGLE TILING.

A solution is *legal* if it obeys the *minimum weight constraint*, i.e., all parts are fat.

2. MIN-MAX GRAPH PARTITION

We present two approximation algorithms for MIN-MAX GRAPH PARTITION whose performance is summarized in Theorems 2.1 and 2.2.

THEOREM 2.1. *There is a polynomial time algorithm that approximates MIN-MAX GRAPH PARTITION with ratio 3. Moreover, every partition produced by the algorithm has parts of weight at most $\text{opt} + 2w_{\text{lb}}$.*

THEOREM 2.2. *There is a linear (in the number of nodes) time algorithm that, given a graph and a Hamiltonian cycle for that graph, approximates MIN-MAX GRAPH PARTITION on the input graph with ratio 2.5. Moreover, the partition produced by the algorithm has parts of weight at most $\text{opt} + 1.5w_{\text{lb}}$.*

Since in 4-connected planar graphs, a Hamiltonian cycle can be found in linear-time (using the algorithm of Chiba and Nishizeki [Chiba and Nishizeki 1989]), we get the following corollary immediately from Theorem 2.2.

COROLLARY 2.3. *There is a linear (in the number of nodes) time algorithm that approximates MIN-MAX GRAPH PARTITION on 4-connected planar graphs with ratio 2.5. Moreover, every partition produced by the algorithm has parts of weight at most $\text{opt} + 1.5w_{\text{lb}}$.*

Sections 2.1– 2.5 are devoted to the proof of Theorem 2.1. Theorem 2.2 is proved in Section 2.6.

Recall that an input in MIN-MAX GRAPH PARTITION is a graph (V, E) with node weights $w : V \rightarrow \mathbb{R}^+$ and a weight lower bound w_{lb} . Without loss of generality assume that $w_{\text{lb}} = 1$. (All weights can be divided by w_{lb} to obtain an equivalent instance with $w_{\text{lb}} = 1$.) For now, we also assume that all nodes in the graph are *lean*. (Recall Definition 1.1 of *fat* and *lean*.) We remove this assumption in Section 2.4.

As described in Section 1.2, our algorithm first constructs a 2-tier partition into groups and supergroups, where each supergroup is a group-pair, a triangle or a star (Section 2.1),

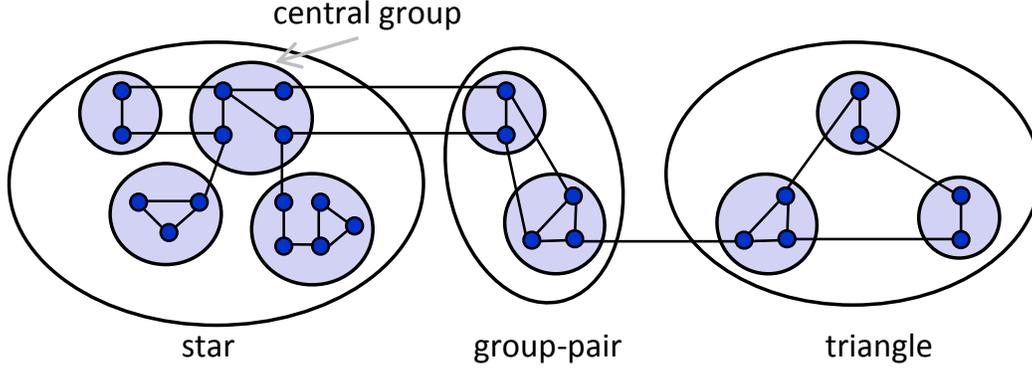


Fig. 1. An example of a 2-tier partition. Dark (blue) circles are vertices of the input graph, light (lavender) circles are groups and white ovals are supergroups.

then transforms it until all stars of large weight have well-defined central nodes, and all noncentral groups in those stars are only connected to central nodes (Sections 2.2 and 2.3) and finally solves an instance of GENERALIZED LOAD BALANCING (Section 2.5), interprets it as a partition and adjusts it to get the final solution.

2.1. A Preliminary 2-Tier Partition

We start by defining a 2-tier partition. (See illustration in Figure 1.) It consists of *groups* and *supergroups*. Intuitively, supergroups are parts in the partition that the algorithm is working on and groups are nearly indivisible subgraphs.

Definition 2.4 (2-tier partition). A 2-tier partition of a graph (V, E, w) containing only lean nodes is a partition of V into lean sets, called *groups*, together with a partition of the groups into fat sets, called *supergroups*. The set of nodes in a group, or in a supergroup, must induce a connected graph. The set of groups contained in a supergroup S is denoted by $\mathcal{G}(S)$.

Since groups are lean and supergroups are fat, each supergroup contains at least two groups. In Definition 2.5 below we assign names to some types of groups and supergroups. See Figures 1 and 2 for an illustration. We say that a subset of nodes S (e.g., a group) and a node v are *adjacent* if the input graph contains an edge (u, v) for some $u \in S$. Two subsets of nodes S_1 and S_2 are *adjacent* if S_1 is adjacent to some node in S_2 .

Definition 2.5 (Group-pair, triangle and star supergroups; central group).

- A supergroup is a *group-pair* if it consists of two groups.
- A supergroup is a *triangle* if it consists of three pairwise adjacent groups.
- A supergroup S with three or more groups is a *star* if it forms a star graph on groups, *i.e.*, it contains a group G , called *central*, such that groups in $\mathcal{G}(S) - \{G\}$ form connected components of $S - G$.

LEMMA 2.6 (INITIAL PARTITION). *Given a connected graph on lean nodes, a 2-tier partition with the following properties can be computed in polynomial time:*

- a. each supergroup is a *group-pair*, a *triangle* or a *star* and
- b. $w(G) + w(H) \geq 1$ for all adjacent groups G and H .

PROOF. First, form the groups greedily: Make each node a group. While there are two groups G and H such that $G \cup H$ is lean and connected, merge G and H .

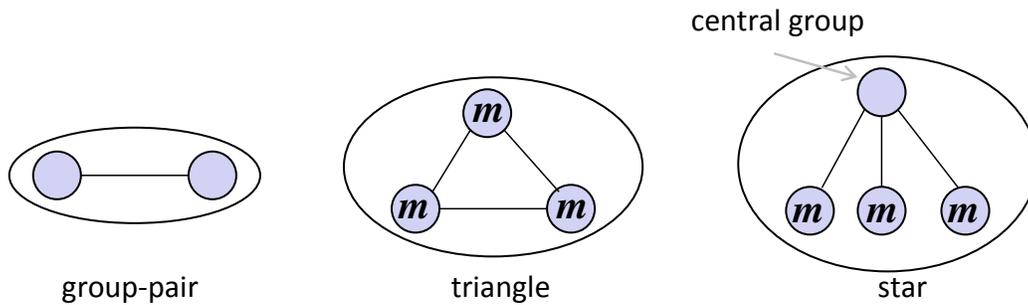


Fig. 2. An illustration for Definitions 2.5 and 2.7: types of supergroups and mobile groups. Mobile groups are indicated with an “ m ”.

Second, form group-pairs greedily: While there are two adjacent groups G and H that are not included in a supergroup, form a supergroup $G \cup H$.

Next, insert remaining groups into supergroups: For each group G still not included in a supergroup, pick an adjacent group H . Since the second step halted, H is in some group-pair created in that step. Insert G into H 's supergroup.

Finally, break down large supergroups that are not stars: Consider a group-pair P created in the second step from groups G and H , and let S be the supergroup that was formed from P . Suppose S has 4 or more groups, but is not a star. Since groups in $S - P$ are not connected, and neither G nor H can become the central group of S , there are two different groups G' and H' in S that are adjacent to G and H , respectively. Let S_1 be the union of G , G' and all other groups in S that are not adjacent to H . Replace S with S_1 and $S - S_1$.

In the resulting 2-tier partition, all supergroups with 4 or more groups are stars, so item (a) of the lemma holds. Item (b) is guaranteed by the first step of the construction. \square

2.2. Improving the Initial 2-Tier Partition

In this section, we modify the initial 2-tier partition, while maintaining property (a) and a weaker version of property (b) of Lemma 2.6. As we are working on our 2-tier partition, we will rearrange groups and supergroups. A group is called *mobile* if when it is removed from its supergroup, the modified supergroup satisfies property (a) of Lemma 2.6: namely, it is a group-pair or a star. (Observe that one cannot obtain a triangle by removing a group from a group-pair, a triangle or a star.)

Definition 2.7 (Mobile group). A group is *mobile* if it is not in a group-pair and it is not a central group. (See illustration in Figure 2).

The goal of this phase of the algorithm is to separate supergroups into the ones that will be repartitioned by the scheduling algorithm and the ones that will be used in the final partition as they are. The latter will include all group-pairs and triangles. Such a supergroup has at most 3 groups and, consequently, weight at most 3. That is, it is sufficiently light to form a part in a 3-approximate solution. Some stars will be repartitioned and some will remain as they are. The scheduling algorithm will repartition only well structured stars: each such star will have a unique *central* node in its central group, and mobile groups will be connected only to central nodes (possibly in multiple stars). Central groups of such stars will be allocated their own parts in the final partition. Mobile groups will be distributed among these parts by the scheduling algorithm. To guarantee that the optimal distribution of central nodes and mobile groups into parts is a (fractional) solution to the scheduling instance we construct, we require that mobile groups can connect by an edge only to central nodes of well structured stars. Noncentral nodes of central groups will join the parts of their

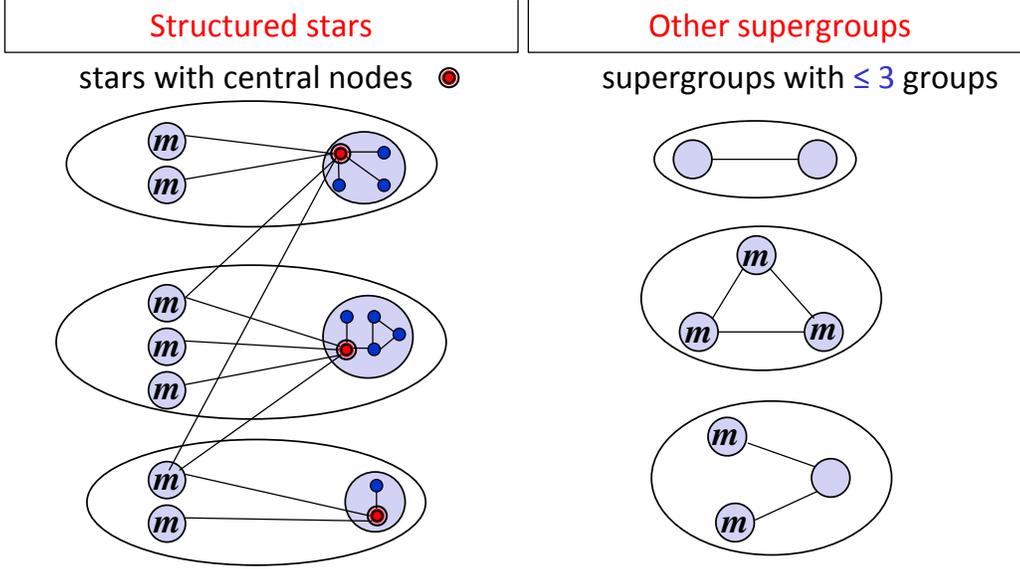


Fig. 3. The goal of the second phase of our algorithm: structured stars and other supergroups.

central nodes after the scheduling algorithm produces a 2-approximate solution. Since, by definition, each group is lean, even after adding central groups, we will still be able to guarantee a 3-approximation. The current phase of the algorithm ensures that each star that is not well structured has 3 groups, and thus can be a part in the final partition.

We explain this phase of the algorithm by specifying several transformations of a 2-tier partition (see Figures 4 and 5). The algorithm applies these transformations to the initial 2-tier partition from Lemma 2.6. Each transformation is defined by the *trigger* and the *action*. The algorithm performs the action for the first transformation for which the trigger condition is satisfied for some group(s) in the current 2-tier partition. This phase terminates when no transformation can be applied.

The purpose of the first transformation, **CombG**, is to ensure that $w(G) + w(H) \geq 1$ for all adjacent groups G and H , where one of the groups is mobile. Even though an even stronger condition, property (b) of Lemma 2.6, holds for the initial 2-tier partition, it might be violated by other transformations. The second transformation, **FormP**, eliminates edges between mobile groups. The third transformation, **SplitC**, ensures that each central group has a unique central node to which mobile groups connect. To accomplish this, while there is a central group G that violates this condition, **SplitC** splits G into two parts, each containing a node to which mobile groups connect. Later, it rearranges resulting groups and supergroups to ensure that all previously achieved properties of our 2-tier partition are preserved (in some cases, relying on **CombG** and **FormP** to reinstate these properties).

If the previously described transformations cannot be applied, star supergroups in the current 2-tier partition are almost well structured, according to the description after Definition 2.7: they have unique central nodes, and all mobile groups connect only to these central nodes, with one exception—they could still connect to group-pairs.

Definition 2.8 (Structured and unstructured stars). A star is *unstructured* if it has a mobile group adjacent to a group-pair or (recursively) to an unstructured star. More formally, a star S_1 is *unstructured* if for some $k \geq 2$, the current 2-tier partition contains supergroups

Fig. 4. Transformations. (Perform the first one that applies.)

<p>• CombG = Combine groups. Trigger: A mobile group H is adjacent to another group G and $G \cup H$ is lean. Action: Remove H from its supergroup and merge the two groups.</p> <p>• FormP = Form a group-pair. Trigger: Two mobile groups are adjacent, and they belong either to two different supergroups or, if this is applied at the end of the transformation SplitC, to a supergroup with more than three groups. Action: Remove them from their supergroup(s) and combine them into a group-pair.</p> <p>• SplitC = Split a central group. Trigger: In a star S, the central group G contains two different nodes u and v adjacent to different mobile groups H_u and H_v, respectively (not necessarily from $\mathcal{G}(S)$). Action: Split G into two connected sets, G_u and G_v, containing u and v, respectively. Split S into S_u and S_v, by attaching each mobile group to G_u or G_v. If $H_u \in \mathcal{G}(S)$ attach H_u to G_u. Similarly, if $H_v \in \mathcal{G}(S)$ attach H_v to G_v. [LeanLean case]: If both S_u and S_v are lean, we turn them into groups, and, as a result, S becomes a group-pair. [FatFat case]: If both S_u and S_v are fat, they become new supergroups. Now assume that S_u is fat and S_v is lean. [FatLean-IN case]: If $H_v \in \mathcal{G}(S)$ then change the partition of S by replacing G and H_v with G_u and S_v. If new S has 4 or more groups, but is not a star, that is, some groups in $\mathcal{G}(S) - \{G_u\}$ are adjacent, treat these groups as mobile and apply CombG or FormP while the trigger conditions for these transformations are satisfied. [FatLean-OUT case]: If $H_v \notin \mathcal{G}(S)$ then remove S_v from G and S and treat it like a mobile group adjacent to H_v and apply CombG or FormP.</p> <p>• ChainR = Chain Reconnect. Trigger: An unstructured star S has 4 or more groups. By Definition 2.8, we have a chain of supergroups $S = S_1, \dots, S_k$, where S_1, \dots, S_{k-1} are stars, S_k is a group-pair and for $i = 1, \dots, k-1$ a mobile group of S_i is adjacent to S_{i+1}. More specifically, since CombG and FormP cannot be applied, for $i = 1, \dots, k-2$ a mobile group of S_i is adjacent to the central group of S_{i+1}. Action: For $i = 1, \dots, k-1$, move a mobile group from S_i to S_{i+1}.</p>

S_2, \dots, S_k , where S_2, \dots, S_{k-1} are stars, S_k is a group-pair and for $i = 1, \dots, k-1$ a mobile group of S_i is adjacent to S_{i+1} . If a star is not *unstructured*, it is called *structured*.

The purpose of **ChainR** is to ensure that each remaining unstructured star has at most 3 groups. **ChainR** is triggered if there is an unstructured star S with 4 or more groups. This can happen only if S is connected by a chain of unstructured stars to a group-pair. The mobile groups along this chain are reconnected, as explained in Figure 4 and illustrated in Figure 5. This completes the description of transformations and this phase of the algorithm.

2.3. Analysis of the Phase of the Algorithm That Generates the Improved Partition

We analyze the properties of a 2-tier partition to which our transformations cannot be applied in Lemma 2.9 and bound the running time of this stage of the algorithm in Lemma 2.10.

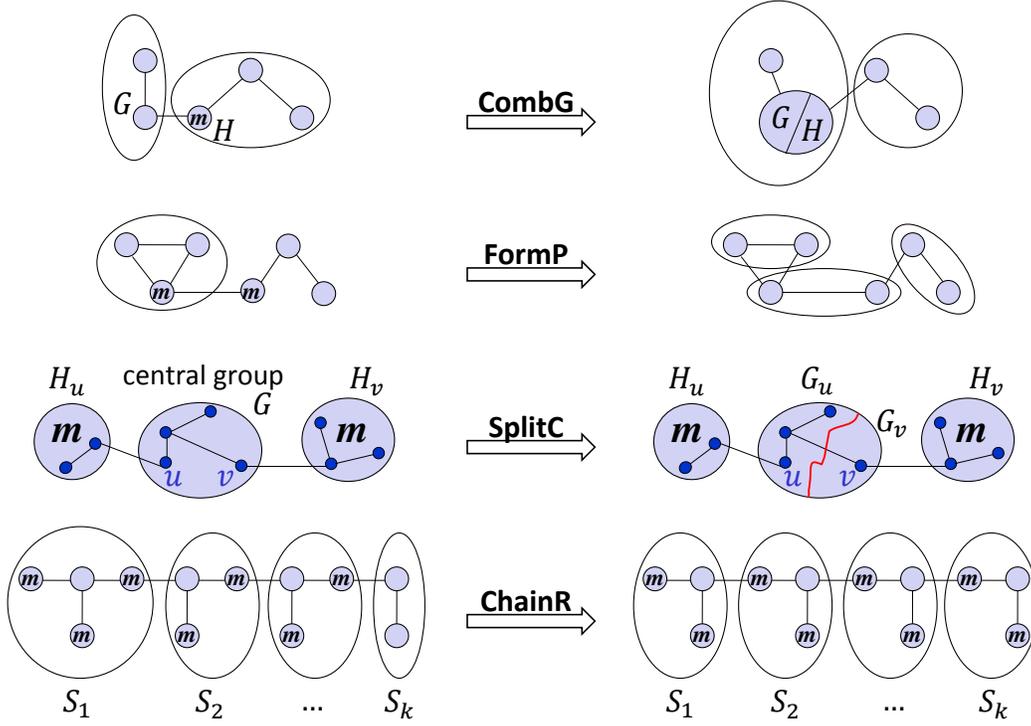


Fig. 5. Transformations. **SplitC** has four cases: all split a central group G into two parts, G_u and G_v , and combine them with groups H_u and H_v to form new groups or supergroups (depending on the weight of the resulting pieces).

LEMMA 2.9. *When transformations **CombG**, **FormP**, **SplitC** and **ChainR** cannot be applied, the resulting 2-tier partition satisfies the following:*

- If G is a center group and H is a mobile group of the same supergroup then $w(G) + w(H) \geq 1$.*
- No edges exist between mobile groups except for groups in the same triangle.*
- Each star S has exactly one node in its central group which is adjacent to mobile groups. We call it the central node of S and denote it by $c(S)$.*
- Each supergroup with 4 or more groups is a structured star.*

PROOF. The lemma follows directly from the definition of the transformations. Note that Lemma 2.6(a) holds as an invariant under all transformations. This fact makes it easier to verify the following claims. If (a) does not hold, we can apply **CombG**. If (b) does not hold, we can apply **CombG** or **FormP**. If (c) does not hold, we can apply **CombG**, **FormP** or **SplitC**. If (d) does not hold, we can apply **ChainR**. \square

LEMMA 2.10. *The algorithm performing transformations defined in Figure 4 on an input 2-tier partition until no transformations are applicable runs in polynomial time.*

PROOF. It is easy to see that performing each transformation and verifying the trigger conditions takes polynomial time. It remains to show that this stage of the algorithm terminates after a polynomial number of transformations. We define two measures of progress, which can be improved only a polynomial number of times, and show that each transformation improves at least one of the two measures. The first measure, *excess*, captures excess

of groups in supergroups. The smallest number of groups in a supergroup is 2. For the first extra group, excess of the supergroup is set to 1. Each additional group incurs additional excess 2.

Definition 2.11 (Excess). For a supergroup S with k groups in $\mathcal{G}(S)$, $excess(S) = \max(2k - 5, 0)$. Total *excess* is the sum of excesses of supergroups.

The second measure of progress is the number of nodes in central groups. Each transformation is classified as one of two types, reducing one of the two measures:

Definition 2.12 (Transformation types). A transformation is of type (A) if it decreases total *excess*, and of type (B) if it decreases the number of nodes in central groups without increasing the total excess.

We can perform at most $2|V|$ transformations of type (A) because total *excess* is always in the interval $[0, 2|V|]$. Transformations of type (A) can increase the number of nodes in central groups. However, without making a transformation of type (A), we can perform at most $|V|$ transformations of type (B). Therefore, an algorithm that applies transformations of types (A) and (B) terminates after at most $2|V|^2$ transformations. Claim 2.13 demonstrates that each transformation is of types (A) or (B). Thus, this phase of the algorithm runs in polynomial time. \square

CLAIM 2.13. *Each transformation in Figure 4 is of types (A) or (B), specified in Definition 2.12.*

PROOF. **CombG** and **FormP** remove mobile groups, thus decreasing total excess. **FormP** also forms a new supergroup, but with zero excess. These transformations are of type (A).

When **ChainR** is performed on a chain S_1, \dots, S_k , supergroup S_1 must have 4 or more groups. Since we pay 1 in excess for the third group and 2 for every additional group in a supergroup, $excess(S_1)$ drops by 2 when **ChainR** removes one of the groups in S_1 . Since S_k has two groups, its excess increases by 1 when **ChainR** adds a group to it. In the intermediate supergroups touched by **ChainR**, the number of groups remains unchanged. Therefore, their excess does not change, and the overall excess drops by 1. Thus, **ChainR** is also a transformation of type (A).

It remains to analyze the type of **SplitC**. In the **LeanLean** case, **SplitC** changes a supergroup with positive excess into a group-pair (with excess 0). In the **FatFat** case, **SplitC** splits a supergroup into two, so even though the number of groups may increase by one, the total excess has to decrease.

In the **FatLean-IN** case, $H_v \subset S_v$. If G_u becomes the new central group then we decrease the number of nodes in the central group, and the transformation has type (B). If not, and S has 3 groups, we create a triangle and eliminate the central group, making this a transformation of type (B). Otherwise, the new group S_v is connected to another mobile group of S and we perform transformation **CombG** or **FormP**, of type (A).

Finally, in the **FatLean-OUT** case, we remove S_v from S , decreasing the number of nodes in the central group of S . Next we perform a transformation, **CombG** or **FormP**, on S_v and H_v . If this transformation is **CombG**, the total excess remains the same, so the overall transformation has type (B). If it is **FormP**, the overall transformation has type (A). \square

2.4. A 2-Tier Partition on Graphs with Arbitrary Weights

In this section, we remove the assumption that all nodes in our input graph are lean. To obtain a 2-tier partition P of a graph (V, E, w) with arbitrary node weights, first allocate a separate supergroup for each fat node. Let V_{lean} be the set of lean nodes. Form *isolated groups* from lean connected components of V_{lean} . For fat connected components of V_{lean} ,

compute the 2-tier partition using the method from Sections 2.1 and 2.2. The next lemma states the main property of the resulting partition P . It follows directly from Lemma 2.9.

LEMMA 2.14 (MAIN). *Consider the 2-tier partition P of a graph (V, E, w) , obtained as described above. Let C be the set consisting of fat nodes and central nodes of structured stars in that 2-tier partition. Then mobile groups of structured stars are connected components of $V - C$.*

PROOF. By definition, each group is connected. It remains to show that a node in a mobile group of a structured star cannot be adjacent to nodes of $V - C$ which are in different groups. By definition of V_{lean} and C , a node of $V - C$ can be adjacent only to nodes in its connected component of V_{lean} . Recall that each group in a star is either central or mobile. A node in a mobile group cannot be adjacent to a node in a different mobile group by Lemma 2.9(b). It cannot be adjacent to a noncentral node in a central group by Lemma 2.9(c). Finally, in a structured star, it cannot be adjacent to a node in an unstructured star or a group-pair, by Definition 2.8. \square

2.5. Reduction to Scheduling and the Final Partition

We reduce MIN-MAX GRAPH PARTITION to Generalized Load Balancing (GLB), and use a 2-approximation algorithm presented in [Kleinberg and Tardos 2006] for GLB (which is based on the 2-approximation algorithm of Lenstra et al. for SCHEDULING ON UNRELATED PARALLEL MACHINES) to get a 3-approximation for MIN-MAX GRAPH PARTITION.

The number of parts in the final partition will be equal to the number of fat nodes plus the number of supergroups in the 2-tier partition of Section 2.4. We use all group-pairs, triangles and unstructured stars as parts in the final partition. By Definition 2.5 and Lemma 2.9(d), each such supergroup has weight less than 3. We use central nodes of structured stars and fat nodes as *seeds* of the remaining parts: namely, in the final partition, we create a part for each central node and each fat node, and partition the remaining groups among these parts using a reduction to GLB.

Now we explain our reduction. Recall that in GLB, the input is the set M of m parallel machines, the set J of n jobs, and for each job $j \in J$, the processing time t_j and the set M_j of machines on which the job j can be scheduled. The starting point of the reduction is the 2-tier partition from Section 2.4. We create a machine for every node in C , where C is the set consisting of fat nodes and central nodes of structured stars, as defined in Lemma 2.14; that is, we set $M = C$. We create a job in J for every node in C , every isolated group and every mobile group of a structured star. To simplify the notation, we identify the names of the machines and jobs with the names of the corresponding nodes and groups. A job corresponding to a node i in C can be scheduled only on machine i , that is, $M_i = \{i\}$. A job corresponding to a mobile or an isolated group j can be scheduled on machine i iff group j is connected to a node $i \in C$. This defines $M(j)$. We set $t_j = w(j)$ for all $j \in J$.

We run the algorithm described in Chapter 11.7 of [Kleinberg and Tardos 2006] for GLB (which is based on the algorithm of [Lenstra et al. 1990] for SCHEDULING ON UNRELATED PARALLEL MACHINES) on the instance defined above. The solution returned by the algorithm is interpreted as a partition of the nodes of the original graph as follows. If job j is scheduled on machine i then node (or group) j is assigned to part i of the partition. Each central group is assigned to the same part as the central node of the group.

The final part of the algorithm repairs lean parts in the resulting partition. One subtle point here is that parts are produced using the scheduling algorithm. However, to repair them we rely on the 2-tier partition that was produced before running the scheduling algorithm. So, when we refer to supergroups and groups, they are from this 2-tier partition.

While there is a lean part P in the partition, reassign a group as follows. Let S be the star in the 2-tier partition whose center was a seed for P . (A lean part cannot have a fat node as a seed.) Let C be the central group of S . Then, by construction, P contains C .

Remove a mobile group of S , say H , from its current part and insert it into P . Now, by Lemma 2.9(a), $w(P) \geq w(C) + w(H) \geq 1$ because P contains C and H .

This repair process terminates because each part is repaired at most once. Since we repair P using a mobile group from the supergroup corresponding to P (that is, the supergroup from the 2-tier partition whose center is C), the subsequent repairs of other parts do not remove H from part P . Later, even if P loses a mobile group when we repair some other part P' , the weight of P still satisfies: $w(P) \geq w(C) + w(H) \geq 1$. Thus, after a number of steps which is at most the number of parts, all parts become fat.

LEMMA 2.15. *The final partition returned by the algorithm above has parts of weight at most $\text{opt} + 2$.*

PROOF. To analyze our algorithm, we consider the linear program (LP) used in Chapter 11.7 of [Kleinberg and Tardos 2006] to solve GLB. This program has a variable x_{ji} for each job $j \in J$ and each machine $i \in M_j$. Setting $x_{ji} = t_j$ indicates that job j is assigned to machine i , while setting $x_{ji} = 0$ indicates the opposite. We relax $x_{ji} \in \{0, t_j\}$ to $x_{ji} \geq 0$ to get an LP. Let J_i be the set of jobs that can be assigned to machine i . The algorithm for GLB solves the following LP, where the first set of constraints ensures that each job is assigned to 1 machine, and the second set, that the makespan is at most L .

$$\begin{aligned} \text{Minimize } L \text{ subject to } & \sum_{i \in M_j} x_{ji} = t_j & \forall j \in J; \\ & \sum_{j \in J_i} x_{ji} \leq L & \forall i \in M; \\ & x_{ji} \geq 0 & \forall j \in J \text{ and } i \in M_j. \end{aligned}$$

Then the solution of this LP is rounded to obtain x_{ji} 's with values in $\{0, t_j\}$, forming the output of the algorithm for GLB. The following theorem is a strengthening of (11.33) in [Kleinberg and Tardos 2006], which follows directly from the analysis presented there. (It is also an easy strengthening of the Rounding Theorem of Lenstra et al. [Lenstra et al. 1990] for the special case of GLB.)

THEOREM 2.16 (STRENGTHENING OF (11.33) IN [KLEINBERG AND TARDOS 2006]).

Let L^ be the value of the objective function returned by the LP above. Let U be the set of jobs with unique machines, i.e., $U = \{j : |M_j| = 1\}$, and let \mathbf{t} be the maximum job processing time among the jobs not in U , i.e., $\mathbf{t} = \max_{j \in J \setminus U} t_j$. In polynomial time, the algorithm in [Kleinberg and Tardos 2006] outputs a solution to GLB with the maximum machine load at most $L^* + \mathbf{t}$.*

For completeness, we present the proof, closely following [Kleinberg and Tardos 2006].

PROOF. Given a solution (x, L) to the LP above, consider the following bipartite graph $B(x) = (V(x), E(x))$. The nodes are the set of machines and the set of jobs, that is, $V(x) = M \cup J$. There is an edge $(i, j) \in E(x)$ if and only if $x_{ji} > 0$.

The algorithm in [Kleinberg and Tardos 2006] (pp. 641–643) first solves the LP and then efficiently converts an arbitrary solution of the LP to a solution with the same load L and no cycles in the graph $B(x)$. After that, each connected component of $B(x)$ is a tree (with nodes corresponding to jobs and machines). The algorithm roots each connected component at an arbitrary node. Each job that corresponds to a leaf is assigned to the machine that corresponds to its parent. Each job that corresponds to an internal node is assigned to an arbitrary child of that node.

It remains show that each machine has load at most $L^* + \mathbf{t}$. Let $S_i \subseteq J_i$ be the set of jobs assigned to machine i . Then S_i contains those neighbors of node i that have degree 1 in $B(x)$, plus possibly one neighbor ℓ of larger degree: namely, the parent of i . Since ℓ

has degree larger than 2, it does not belong to the set U of jobs with unique machines. Therefore, $t_\ell \leq \mathbf{t}$. For all jobs j in S_i that correspond to nodes of degree 1, machine i is the only machine that is assigned any part of job j by the solution x . That is, $x_{ji} = t_j$. Consequently,

$$\sum_{j \in S_i, j \neq \ell} t_j \leq \sum_{j \in J_i} x_{ji} \leq L^*.$$

We conclude that, for all i , the load on machine i is at most $L^* + \mathbf{t}$. \square

Note that jobs with non-unique machines in our instance of GLB correspond to groups in our 2-tier partition. Since groups are lean by definition, $\mathbf{t} < 1$. By Theorem 2.16, we get a solution for our instance of GLB with maximum machine load less than $L^* + 1$.

Next, we argue that every legal partition of nodes in C , in mobile groups of structured stars and in isolated groups (namely, all nodes for which we created corresponding jobs in our GLB instance) has cost at least L^* . This will imply that $opt \geq L^*$. Consider a group G that is either mobile or isolated. By Lemma 2.14, this group is a connected component of $V - C$. Let c_1, \dots, c_k be the nodes of C that are adjacent to G . Consider a node $v \in G$. In a legal partition of the graph, v must be in the same part as one of c_i 's: if v belongs to a part P of that partition, P must contain a path that starts at v and ends outside of G (because G is lean), and this path must contain one of the nodes c_i . Therefore, in a legal solution, the weight $w(G)$ is distributed in some manner between the parts that contain c_i 's. Thus, there is at most one part per c_i . If two nodes from C are in the same part, splitting it into two parts can only decrease the cost. Setting x_{G_i} to the fraction of G 's weight that ended up in the same part as node i for all groups G and $i \in C$, results in a valid fractional solution to the LP. Thus, $opt \geq L^*$.

We argued that the solution to our instance of GLB output by the algorithm in [Kleinberg and Tardos 2006] has cost at most $L^* + 1 \leq opt + 1$. Adding the nodes from the central groups to each part increased the cost by at most 1. The repairing phase that enforced the lower bound, raised the weight of each repaired part to at most 2, and could only decrease the weight of remaining parts. Thus, the cost of the partition returned by our algorithm is at most $opt + 2$.

Theorem 2.1 follows from Lemma 2.15. This completes the analysis of our approximation algorithm for MIN-MAX GRAPH PARTITION.

2.6. MIN-MAX GRAPH PARTITION on graphs with given Hamiltonian cycles

Here we prove Theorem 2.2 that states that MIN-MAX GRAPH PARTITION problem on a graph with an explicitly given Hamiltonian cycle can be approximated with ratio 2.5 in linear time. As before, we assume without loss of generality that $w_{lb} = 1$.

PROOF OF THEOREM 2.2. Our algorithm partitions the nodes using only the edges from the Hamiltonian cycle. Let P_0 be the minimum-weight contiguous part of the cycle which satisfies the weight constraint, namely, $w(P_0) \geq 1$. We rename the nodes, so that the cycle is $(0, 1, \dots, n - 1)$ and $P_0 = \{0, \dots, i_0\}$. We pack nodes greedily into parts P_0, \dots, P_m so that

$$P_k = \{i_{k-1} + 1, \dots, i_k\},$$

$$w(P_k) - w_{i_k} < 1,$$

and (except for $k = m$) $w(P_k) \geq 1$. If $w(P_m) < 1$ we combine parts P_0 and P_m . This completes the description of the algorithm.

Let \mathbf{w} denote the maximum weight of a node. For $k \in [1, m]$, we can bound the weight of P_k from above as follows:

$$w(P_k) < 1 + w_{i_k} \leq 1 + \mathbf{w} \leq 1 + opt.$$

Thus, if $w(P_m) \geq 1$, our partition is a 2-approximation of the optimum.

It remains to bound $w(P_0) + w(P_m)$, assuming $w(P_m) < 1$. If $\mathbf{w} \geq 1$ then, by definition of P_0 , we have $w(P_0) \leq \mathbf{w} \leq \text{opt}$, and thus $w(P_0) + w(P_m) < \text{opt} + 1$. A similarly easy case occurs when $w(P_0) \leq 1.5$: namely, $w(P_0) + w(P_m) \leq 2.5$.

For the rest of the proof, assume $\mathbf{w} < 1$ and $w(P_0) > 1.5$, that is, every contiguous part of the cycle has weight strictly below 1 or strictly above 1.5.

Definition 2.17 (Heavy node). A node i is *heavy* if $w_i \geq 0.5$ and *light* otherwise.

LEMMA 2.18. *Let i be a heavy node and Q_i be the maximal part of the cycle that contains i and no other heavy node. Then $w(Q_i) < 1$.*

PROOF. We use the assumption that every contiguous part of the cycle has weight either less than 1 or more than 1.5. Suppose for the sake of contradiction that $w(Q_i) \geq 1$. Then we can keep decreasing Q_i by removing light nodes while Q_i remains a contiguous part of the cycle of weight at least 1. When we cannot continue, the final result is a contiguous part of the cycle of weight between 1 and 1.5. This is a contradiction. Thus, $w(Q_i) < 1$. \square

Since Lemma 2.18 holds for all heavy nodes, $\sum_{i=0}^{n-1} w_i$ is smaller than the number of heavy nodes in the graph. In every legal solution, the number of parts is smaller than the number of heavy nodes in the graph because each part must have weight at least 1. Therefore, by Pigeonhole Principle, every solution must have a part with two heavy nodes. Let \mathbf{m} be the minimum weight of a heavy node. Then $\text{opt} \geq 2\mathbf{m}$, and consequently,

$$\mathbf{m} + 1 \leq \text{opt} - \mathbf{m} + 1. \quad (1)$$

Now suppose that i and j are consecutive heavy nodes and $w_i = \mathbf{m}$. Then $\{i\} \cup Q_j$ is a contiguous part of the cycle, and it has weight at least 1 and at most $\mathbf{m} + 1$. By definition of P_0 , its weight is also bounded above by $\mathbf{m} + 1$. By (1), $w(P_0) \leq \text{opt} - \mathbf{m} + 1$. Thus, $w(P_0) + w(P_m) < \text{opt} - \mathbf{m} + 2 \leq \text{opt} + 1.5$. \square

3. MIN-MAX BIN COVERING

In this section, we present our algorithm for MIN-MAX BIN COVERING.

THEOREM 3.1. MIN-MAX BIN COVERING *can be approximated with ratio 2 in time linear in the number of items, assuming that the items are sorted by weight.*

PROOF. Without loss of generality assume that $w_{1b} = 1$, $\mathcal{I} = \{1, \dots, n\}$ and $w_1 \geq w_2 \geq \dots \geq w_n$.

If $w(\mathcal{I}) < 1$, there is no legal packing. If $1 \leq w(\mathcal{I}) < 3$, a legal packing consists of at most 2 bins. Therefore, $\text{opt} \geq w(\mathcal{I})/2$. Thus, $w(\mathcal{I}) \leq 2\text{opt}$, and we get a 2-approximation by returning one bin $B_1 = \mathcal{I}$.

Theorem 3.1 follows from Lemma 3.2, dealing with instances with $w(\mathcal{I}) \geq 3$. \square

LEMMA 3.2. *Given a MIN-MAX BIN COVERING instance \mathcal{I} with n items and weight $w(\mathcal{I}) \geq 3$, a solution with cost at most $\text{opt} + 1$ can be found in time $O(n)$.*

PROOF. We compute a preliminary packing greedily, filling successive bins with items in order of decreasing weights, and moving to a new bin when the weight of the current bin reaches or exceeds 1. Let B_1, \dots, B_k be the resulting bins. All bins B_i in the preliminary packing satisfy $w(B_i) < w_1 + 1 \leq \text{opt} + 1$. In addition, $w(B_i) \geq 1$ for all bins, excluding B_k .

If $w(B_k) \geq 1$, the preliminary packing is legal and has cost at most $\text{opt} + 1$. Otherwise, if $w_1 \geq 1$ then $B_1 = \{1\}$, and we can combine B_1 and B_k into a bin of weight below $\text{opt} + 1$. Also, if $w(B_{k-1}) + w(B_k) \leq 2$, we obtain a legal packing of cost at most $\text{opt} + 1$ by combining B_{k-1} and B_k . In the remainder of the proof, we show how to rearrange items in B_k when

$$w_1 < 1; \quad (2)$$

$$w(B_k) < 1; \tag{3}$$

$$w(B_{k-1}) + w(B_k) > 2 \tag{4}$$

to obtain a legal packing with cost at most $opt + 1$.

Definition 3.3. A bin B is *good* if $w(B) \in [1, 2]$. A packing where all bins are good is called *good*.

Equation (2) implies that all bins in the preliminary packing, excluding B_k , are good.

OBSERVATION 3.4. *If $i \in B_j$ then $w(B_j) < 1 + w_i$. Thus, $w(B_j) - w_i < 1$.*

Definition 3.5. An item i is called *small* if $w_i \leq 1/2$, and *large* otherwise.

Since $w(B_k) < 1$, $w(B_{k-1}) < 2$ and $w(\mathcal{I}) \geq 3$, the number of bins, k , is at least 3. The remaining proof is broken down into cases, depending on exactly where in bins B_{k-2} , B_{k-1} and B_k the first small item appears. Depending on the case, we repack either bins B_{k-2} , B_{k-1} , B_k or bins B_1 and B_k to ensure that all the resulting bins satisfy the weight lower bound. (Note that if $k = 3$ then $B_1 = B_{k-2}$.)

Case 1: B_{k-1} contains a small item, say i . If $w(B_{k-2}) + w_i \leq 2$, we move i to B_{k-2} , and combine B_{k-1} with B_k . The weight of the combined B_{k-1} and B_k is at least 2 (by (4)) minus 0.5 (because we removed a small item). On the other hand, $w(B_{k-1} \setminus \{i\}) < 1$ and $w(B_k) < 1$, by Observation 3.4 and (3), respectively. Therefore, the weight of the new bin is below 2. That is, the resulting packing is good.

Now assume $w(B_{k-2}) + w_i > 2$, and let j be the lightest item in B_{k-2} . By Observation 3.4 and the assumption, $w_j + w_i > 1$. Since item i is small, item j must be large. While there exists $g \in B_k$, such that $w(B_{k-2}) + w_g \leq 2$, we move g from B_k to B_{k-2} . Afterwards, we replace B_{k-1} and B_k with their union C . If $w(C) \leq 2$, we got a good packing. Otherwise, B_k had an item left, call it g , that did not fit in B_{k-2} . We remove i and g from C , replace j with g in B_{k-2} , and form a new bin $\{i, j\}$.

To see that the resulting packing is good, consider the three bins we created. Bin $\{i, j\}$ is good—we already observed that $w_i + w_j > 1$, and two items cannot exceed weight 2. The weight of the new B_{k-2} cannot be too large because we replaced a large item, j , with a small item, g . Since g did not fit into B_{k-2} without violating the weight upper bound of 2, the weight of new B_{k-2} is $(w(B_{k-2}) + w_g) - w_j \geq 2 - 1 = 1$. Finally, the weight of C cannot be too small because it exceeded 2 before we removed two small items, i and g ; it cannot be too large because $w(B_{k-1}) - w_i < 1$ and $w(B_k) < 1$.

Case 2: B_k contains a large item, say i . Then \mathcal{I} contains $2k - 1$ large items and, consequently, any legal packing has a bin with an odd number of large items, say a . In an optimal packing, a must be 1 or 3 because more than 3 large items can always be partitioned into two bins of weight at least 1 each.

If there is an optimal packing with $a = 3$ then we can merge the last two bins. The resulting bin contains the three smallest large items plus the small items of B_k . The three smallest large items weigh at most opt because some bin in the optimal solution contains 3 large items. The small items of B_k weigh less than $1/2$ because B_k contains a large item and its total weight $w(B_k) < 1$ by (3). Therefore, the weight of the new bin is below $opt + 1/2$.

Now suppose $a = 1$ in some optimal packing. Since B_k contains a large item, each preceding bin must contain two large items. In particular, $B_1 = \{1, 2\}$, where $w_1 \geq w_2$. The weight of a bin with one large item is at most the weight of the heaviest large item plus the weight of all small items, i.e., $w_1 + (w(B_k) - w_i)$. Since there is such a bin in an optimal solution, $w_1 + (w(B_k) - w_i) \geq 1$. Thus, we can swap items 1 and i in bins B_1 and B_k to obtain a good packing.

Our algorithm for Case 2 checks if swapping items 1 and i in bins B_1 and B_k results in a good packing. If yes, it proceeds with the swap. Otherwise, it merges the last two bins.

Case 3: (the remaining case) B_k consists of small items, and all other bins, of two large items. If the optimal packing has two large items in a bin then $opt \geq w(B_{k-1})$, and it suffices to combine the last two bins. We cannot verify the above condition on the optimal packing, but we will show how to construct a good packing when this condition is violated. Our algorithm will follow the steps of the construction, and resort to combining the last two bins if it does not obtain a good packing.

Now assume that in the optimal packing no bin contains two large items, and consequently, for $i = 1, 2, 3, 4$ item i belongs to a bin $\{i\} \cup C_i$ where $C_i \subset B_k$. Let D be a maximal subset of B_k such that $w_1 + w(D) < 1$. Let j be any item in $B_k - D$ and $D' = B_k - D - \{j\}$. We repack B_1 and B_k as $B'_1 = \{1, j\} \cup D$ and $B'_k = \{2\} \cup D'$.

One can see that $1 \leq w(B'_1) < 1 + w_j$. (The first inequality follows from the maximality of D). Clearly, $w(B'_k) \leq w(B_1) + w(B_k) - 1 < 2 + 1 - 1$. It remains to show that $w(B'_k) \geq 1$. Note that B_k contains 3 disjoint sets, C_2, C_3, C_4 , each of weight at least $1 - w_2$. Consequently, $B_k - \{j\}$ contains at least two such sets, and $w(D) + w(D') \geq 2(1 - w_2)$. Thus,

$$w(D') \geq 2(1 - w_2) - (1 - w_1) \geq 1 - w_2.$$

□

4. MIN-MAX RECTANGLE TILING

We present two approximation algorithms for MIN-MAX RECTANGLE TILING whose performance is summarized in Theorems 4.2 and 4.3, presented in Sections 4.1 and 4.2, respectively. Recall that an input in MIN-MAX RECTANGLE TILING consists of the side lengths of the rectangle, m and n , the weights $w : \mathcal{I} \rightarrow \mathbb{R}^+$ for $\mathcal{I} = \{1, \dots, m\} \times \{1, \dots, n\}$ and a weight lower bound w_{lb} . We represent the weights by an $m \times n$ array.

Definition 4.1. A *tile* is a subrectangle of weight at least w_{lb} .

4.1. MIN-MAX RECTANGLE TILING with Arbitrary Weights

Our first algorithm works for the general version of MIN-MAX RECTANGLE TILING.

THEOREM 4.2. MIN-MAX RECTANGLE TILING can be approximated with ratio 4 in time $O(mn)$.

Proof. Our algorithm first preprocesses the input array to ensure that the last row is fat. (Recall that *fat* and *lean* were defined in Definition 1.1.) Then it greedily *slices* the rectangle, that is, partitions it using horizontal lines. The resulting groups of consecutive rows are called *slices*. Finally, each slice is greedily *diced* using vertical lines into subrectangles, which form the final tiles.

Preprocessing. W.l.o.g., assume that the weight of the input rectangle is at least w_{lb} . (Otherwise, there is no legal tiling.) Let R_i denote the i th row of the input array. While R_m is thin, we perform a step of preprocessing that replaces the last two rows, R_{m-1} and R_m , with row $R_{m-1} + R_m$ (and decrements m by 1). When R_m is thin, every subset of R_m is thin, and cannot be a valid tile. Thus, every element of R_m has to be in the same tile as the element directly above it. Therefore, a preprocessing step does not change the set of valid tilings of the input rectangle.

Slicing. In a step of slicing, we start at the top (that is, go through the rows in the increasing order of indices). Let j be the smallest index such that remaining (not yet sliced) top rows up to row R_j form a fat rectangle. Then we cut horizontally between rows R_j and R_{j+1} , and call the top set of rows a slice. We continue on the subrectangle formed by the bottom rows. Since the preprocessing ensured that the last row is fat, all resulting slices are fat.

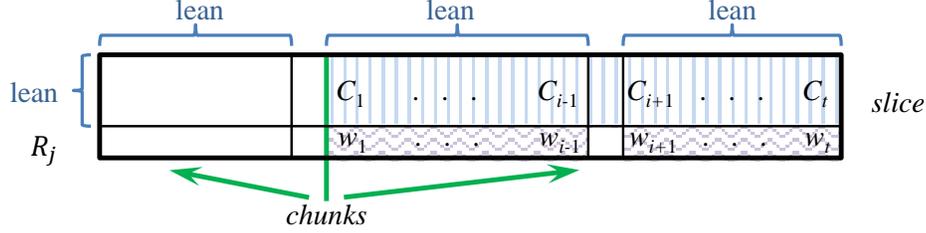


Fig. 6. Anatomy of a slice in the proof of Theorem 4.2.

Dicing. In a step of dicing, analogously to the slicing step, we cut up a slice vertically, dicing away minimal fat sets of leftmost columns, unless the remaining columns form a lean rectangle. The resulting subrectangles are fat, by definition. They form the tiles in the final partition.

Analysis. Consider a tile produced by our algorithm. The rectangle formed by all rows of the tile, excluding the bottom row, is lean because it is obtained by partitioning a valid slice. Thus, the weight of this rectangle is less than w_{lb} , and consequently, less than opt . Let C_1, \dots, C_t be the columns of the tile (partial columns of the input array), and w_1, \dots, w_t be the entries in the bottom row of the slice. Let i be the smallest index such that C_1, \dots, C_i form a fat rectangle. (If this tile is the last one in its slice, then i might be less than t .) By the choice of i , the rectangle formed by C_1, \dots, C_{i-1} is lean, and so is the rectangle formed by C_{i+1}, \dots, C_t . In addition, C_i without w_i is also lean, because it is a subset of the lean part of the slice. Finally, since w_i has to participate in a tile, $w_i \leq opt$. Consequently, the weight of the tile is smaller than $opt + 3w_{lb} \leq 4opt$.

It is easy to implement the algorithm so that each step performs a constant number of operations per array entry. Thus, it can be executed in time $O(mn)$. \square

4.2. MIN-MAX RECTANGLE TILING with 0-1 Weights

We can get a better approximation ratio when the entries in the input array are restricted to be 0 or 1. This case covers the scenarios where each entry indicates the presence or absence of some object.

THEOREM 4.3. MIN-MAX RECTANGLE TILING *with 0-1 entries can be approximated with ratio 3 in time $O(mn)$.*

PROOF. W.l.o.g., assume that the weight of the input rectangle is at least w_{lb} . (Otherwise, there is no legal tiling.) Our algorithm first computes a preliminary partition of the input rectangle, where each part has weight at most $3opt$ or is a *sliceable subrectangle*, according to Definition 4.4. We explain how to partition a sliceable subrectangle in the proof of Lemma 4.5.

Definition 4.4 (Sliceable rectangle). We call a rectangle *sliceable* if one of the following holds:

- (a) it has a fat row (or column) on the boundary;
- (b) it is fat but has only lean rows (or columns).

LEMMA 4.5. *A sliceable rectangle with 0-1 entries of size $a \times b$ can be partitioned into tiles of weight at most $3w_{lb} - 2$ in time $O(ab)$.*

PROOF. Suppose a rectangle satisfies (a) and, without loss of generality, consider the case when the bottom row is fat. We apply slicing and dicing from the proof of Theorem 4.2 to partition the rectangle into tiles. (Note that we do not perform the preprocessing step,

since it might create non-Boolean entries.) The analysis is the same, but now we also use the facts that $w_i \leq 1$ and that each lean piece has weight at most $w_{\text{lb}} - 1$. It implies that the weight of each tile is at most $3(w_{\text{lb}} - 1) + 1 = 3w_{\text{lb}} - 2$, as required.

Now suppose a rectangle satisfies (b) and, without loss of generality, consider the case when it has only lean rows. We apply slicing from the proof of Theorem 4.2 while the total weight of the remaining rows R_{j+1}, \dots, R_a is at least w_{lb} and then make each slice into a tile. For all slices, besides the last one, the last row added to the slice increases its weight from at most $w_{\text{lb}} - 1$ and by at most $w_{\text{lb}} - 1$ because, without this row, we would have a lean slice and because the row itself is lean. For the last slice, even when we obtain a fat collection of rows, we might need to add the remaining rows because their total weight is at most $w_{\text{lb}} - 1$. However, the weight of the last slice is still at most $3(w_{\text{lb}} - 1)$, as required. \square

It remains to explain how to compute a preliminary partition into sliceable subrectangles and parts of weight at most 3opt . First, we preprocess the input rectangle by partitioning it into groups of consecutive rows $L_0, F_1, L_1, \dots, F_k, L_k$, where each group F_i consists of a single fat row and each group L_i is either empty or comprised of lean rows. We consider groups of rows called *sandwiches*, defined next and illustrated in Figure 7.

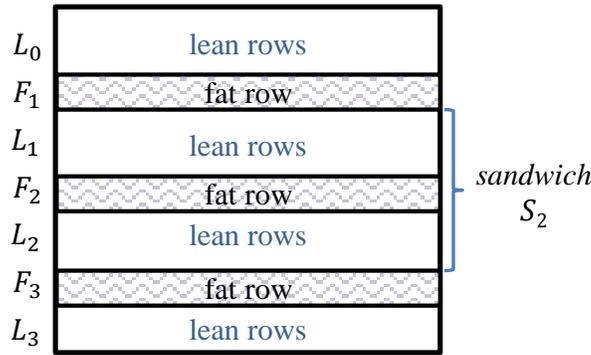


Fig. 7. An illustration for Definition 4.6: a sandwich.

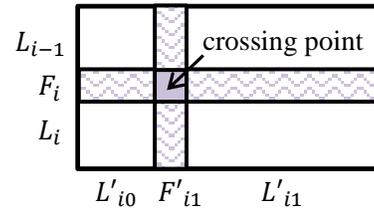


Fig. 8. An illustration for Definition 4.8: a crossing point.

Definition 4.6 (Sandwich). The *sandwich* S_i is the union of groups $L_{i-1} \cup F_i \cup L_i$.

CLAIM 4.7. Let $S_i = L_{i-1} \cup F_i \cup L_i$ be a sandwich, where both L_{i-1} and L_i are lean. Then S_i has either

- at most one fat column or
- two fat columns, each of weight exactly w_{lb} .

PROOF. If S_i has k fat columns, the sum of their entries in F_i is at most k , and the sum of their entries in the rest of S_i is at most $w(L_{i-1}) + w(L_i) \leq 2(w_{\text{lb}} - 1)$. Therefore, all k fat columns together weigh at most $2w_{\text{lb}} + k - 2$.

If $k = 2$, the total weight of the two fat columns is at most $2w_{\text{lb}}$. Since, by definition, each fat column has weight at least w_{lb} , both fat columns must have weight exactly w_{lb} . If $k > 2$, one of the fat columns has weight at most $1 + \frac{2}{k}(w_{\text{lb}} - 1) < w_{\text{lb}}$, a contradiction. \square

If at least one of the groups L_i is empty or fat, we obtain sliceable subrectangles by splitting the input rectangle into L_i and the two parts above and below L_i . (One or more of these parts may be empty.)

Otherwise, if at least one of the sandwiches S_i has zero or two fat columns, we split the input rectangle into S_i and the two sliceable subrectangles above and below S_i . If S_i has no

fat columns, then S_i itself is sliceable. If S_i has two fat columns, consider the partition of S_i into groups of consecutive columns $L'_{i0}, F'_{i1}, L'_{i1}, F'_{i2}, L'_{i2}$, where each group F'_{ij} consists of a single fat column and each group L'_{ij} is comprised of lean columns. If at least one of L'_{ij} 's is fat, we obtain sliceable subrectangles by splitting S_i into L'_{ij} and the two parts to the left and to the right of it. Otherwise, we split S_i into $L'_{i0} \cup F'_{i1}$ and $L'_{i1} \cup F'_{i2} \cup L'_{i2}$, both of weight below $3w_{\text{lb}}$, since, by Claim 4.7, $w(F'_{i1}) = w(F'_{i2}) = w_{\text{lb}}$.

Similarly, if at least one of the sandwiches S_i has exactly one fat column F'_{i1} which splits it into $L'_{i0}, F'_{i1}, L'_{i1}$, where at least one of L'_{ij} 's is fat, we obtain sliceable subrectangles by splitting S_i into L'_{ij} and the remaining part to the left or to the right of it.

Finally, we perform the same steps with the roles of rows and columns switched.

By previous steps and Claim 4.7, in the remaining case, each group L_i is nonempty and lean, and each sandwich S_i has exactly one fat column, F'_{i1} , which splits it into $L'_{i0}, F'_{i1}, L'_{i1}$, where both L'_{i0} and L'_{i1} are lean. Recall that each sandwich has exactly one fat row, F_i . The next definition is illustrated in Figure 8.

Definition 4.8 (Crossing point). We call the common entry of F_i and F'_{i1} the *crossing point* of S_i .

OBSERVATION 4.9. *Every fat subrectangle of S_i contains its crossing point.*

PROOF. This holds because each of L_{i-1}, L_i, L'_{i0} and L'_{i1} is lean. \square

OBSERVATION 4.10. *Each row and column of the input rectangle has weight less than $2w_{\text{lb}}$.*

PROOF. Consider any row. In its sandwich S_i , both column groups L'_{i0} and L'_{i1} are lean, and its entry in the fat column F'_{i1} is at most 1. Thus, each row has weight at most $2(w_{\text{lb}} - 1) + 1 = 2w_{\text{lb}} - 1$. Since we applied all the steps with roles of rows and columns switched, each column also has weight at most $2w_{\text{lb}} - 1$. \square

CLAIM 4.11. *There are less than m crossing points.*

PROOF. If some column C_i contains more than two crossing points, then at least two of them are from disjoint sandwiches or, more precisely, two disjoint fat columns of sandwiches. But then $w(C_i) \geq 2w_{\text{lb}}$, contrary to Observation 4.10.

We already showed how to partition into sliceable subrectangles when C_1 or C_m are fat or when two consecutive columns are both fat. Thus, in the current case, there are less than $m/2$ fat columns and less than $2m/2 = m$ crossing points. \square

Our preliminary partition for this case depends on whether the input rectangle has a fat subrectangle that contains no crossing points.

LEMMA 4.12. *There is an algorithm that, given an $m \times n$ rectangle with 0-1 entries and its crossing points, in time $O(mn)$ either determines that there is no fat subrectangle that contains no crossing points or finds a maximal such subrectangle.*

PROOF. We start by computing the crossing points, sorted by their row. To do that, we go through the sandwiches and apply Definition 4.8. This can be done in $O(mn)$ time. Next, we describe how to find all maximal subrectangles that do not contain crossing points. Later, we will compute their weights.

Consider a maximal rectangle that does not contain crossing points. Because it cannot expand up, it contains either some point in the top row or some point directly below a crossing point. This motivates the following definition. A point is called a *seed* if it is in the top row of the input rectangle or if it is directly below a crossing point. Since there are m points in the top row, by Claim 4.11, there are less than $2m$ seeds.

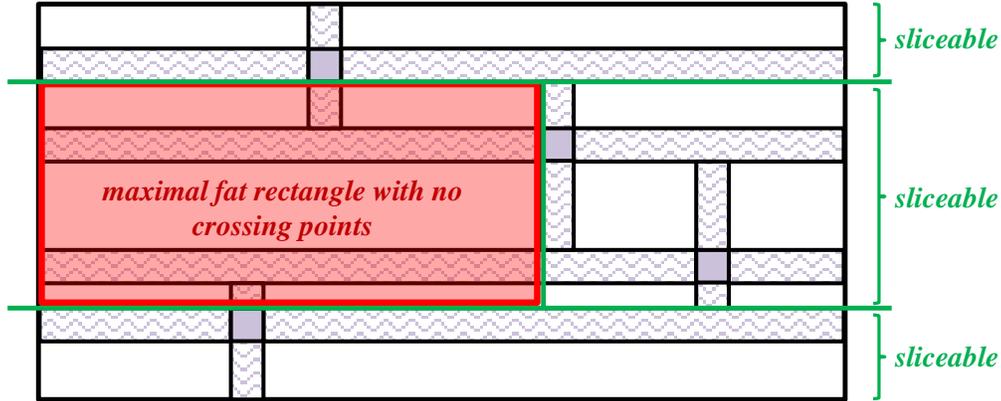


Fig. 9. A maximal fat rectangle and the resulting preliminary partition.

For each seed, we will find, in time $O(n)$, all maximal subrectangles that contain no crossing points and have the given seed on its upper boundary. We initialize our subrectangle to contain the row with the seed. This row is lean because the top row is lean, and there is at least one lean row between each pair of fat rows, so the row below a crossing point is also lean. (See Figures 7 and 8.) By Definition 4.8, the initial subrectangle contains no crossing points.

We expand the current subrectangle downwards, by adding rows until run out of rows or we encounter a row with a crossing point. We output the current subrectangle, then, in the former case, we stop, and in the latter case, we add the next row with the crossing point. Suppose the crossing point is in column c . If the seed is in the column c as well, we have already returned all maximal rectangles that contain no crossing points and have the given seed on its upper boundary. If the seed is in a column to the left (respectively, right) of c , we remove all columns with index c or higher (respectively, lower) from the current subrectangle, and continue the process described in this paragraph.

Since there are $O(m)$ seeds, and for each seed, the procedure above runs in $O(n)$ time, we find all maximal subrectangles with no crossing points in $O(mn)$ time. It remains to show how to find the weight of each reported subrectangle. Let $W[r, c]$ be the weight of the subrectangle that spans the first r rows and the first c columns. Array W can be computed in time $O(mn)$. Let $[a, b]$ denote $\{a, a + 1, \dots, b\}$. Then the weight of the array $[a, b] \times [c, d]$ is

$$w([a, b] \times [c, d]) = W[c, d] + W[a - 1, b - 1] - W[a - 1, d] - W[b, c - 1].$$

Thus, we can compute the weight of the reported subrectangles, and determine if any of them are fat in $O(mn)$ time. \square

Suppose the algorithm in Lemma 4.12 outputs a rectangle, call it A . Recall that A is a maximal fat subrectangle that contains no crossing points. Since A cannot expand further, on each side it is either on the rectangle boundary or has a crossing point immediately next to it. (See Figure 9.) By definition, each crossing point is contained in a fat row. Thus, if there are any rows above (below) A , they form a sliceable subrectangle. Let R be the subrectangle consisting of rows of A . We partition the input rectangle into R and the two sliceable rectangles above and below R . Since the rows immediately above and below R are fat (if they exist), R is a union of sandwiches. Consequently, by Definition 4.8, if A does not lie on the left (respectively, right) boundary, the subrectangle of R to the left (respectively, right) of A is sliceable. Thus, we can further partition R into A and the two

sliceable subrectangles to the left and to the right of A . By Observation 4.9, A itself is also sliceable because all its rows are lean.

Finally, if the input rectangle does not have a fat subrectangle that contains no crossing points, in a legal partition each entry must be in the same tile as one of the crossing points. That is, the number of tiles in a legal partition is at most the number of crossing points, which is equal to k , the number of fat rows F_i . Let i be such that $w(F_i \cup L_i)$ is minimized. Then the total weight of the input rectangle is at least $k \cdot w(F_i \cup L_i)$, so the optimal tiling must have a tile of weight at least $w(F_i \cup L_i)$. That is, $opt \geq w(F_i \cup L_i)$, and therefore

$$w(S_i) = w(L_{i-1}) + w(F_i \cup L_i) < w_{lb} + opt \leq 2opt.$$

Thus, we can partition the rectangle into S_i and the sliceable parts above and below S_i .

We explained how to obtain, in $O(mn)$ time, a preliminary partition of the input rectangle, where each part has weight at most $3opt$ or is a sliceable subrectangle. Now the theorem follows from Lemma 4.5.

REFERENCES

- S. F. Assmann, David S. Johnson, Daniel J. Kleitman, and Joseph Y.-T. Leung. 1984. On a Dual Version of the One-Dimensional Bin Packing Problem. *J. Algorithms* 5, 4 (1984), 502–525.
- Nikhil Bansal and Maxim Sviridenko. 2006. The Santa Claus problem. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 31–40. DOI: <http://dx.doi.org/10.1145/1132516.1132522>
- Piotr Berman, Bhaskar DasGupta, and S. Muthukrishnan. 2002. Slice and dice: a simple, improved approximate tiling recipe. In *SODA*, David Eppstein (Ed.). ACM/SIAM, 455–464.
- Piotr Berman, Bhaskar DasGupta, and S. Muthukrishnan. 2003. Approximation algorithms for MAX-MIN tiling. *J. Algorithms* 47, 2 (2003), 122–134.
- Piotr Berman, Bhaskar DasGupta, S. Muthukrishnan, and Suneeta Ramaswami. 2001. Improved approximation algorithms for rectangle tiling and packing. In *SODA*. 427–436.
- Piotr Berman and Sofya Raskhodnikova. 2010. Approximation Algorithms for Min-Max Generalization Problems. In *APPROX-RANDOM (Lecture Notes in Computer Science)*, Maria J. Serna, Ronen Shaltiel, Klaus Jansen, and José D. P. Rolim (Eds.), Vol. 6302. Springer, 53–66.
- N. Chiba and T. Nishizeki. 1989. The Hamiltonian cycle problem is linear-time solvable for 4-connected planar graphs. *J. Algorithms* 10, 2 (1989), 187–211. DOI: [http://dx.doi.org/10.1016/0196-6774\(89\)90012-6](http://dx.doi.org/10.1016/0196-6774(89)90012-6)
- V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. 2008. k -Anonymous Data Mining: A Survey. In *Privacy-Preserving Data Mining: Models and Algorithms*, Charu C. Aggarwal and Philip S. Yu (Eds.). Springer, Chapter 5.
- János Csirik, David S. Johnson, and Claire Kenyon. 2001. Better approximation algorithms for bin covering. In *SODA*, S. Rao Kosaraju (Ed.). ACM/SIAM, 557–566.
- Wenliang Du, David Eppstein, Michael T. Goodrich, and George S. Lueker. 2009. On the Approximability of Geometric and Geographic Generalization and the Min-Max Bin Covering Problem. In *WADS*. 242–253. DOI: http://dx.doi.org/10.1007/978-3-642-03367-4_22
- Y. J. Garcia, M. A. Lopez, and S. T. Leutenegger. 1998. A greedy algorithm for bulk loading R-trees. In *GIS '98: Proceedings of the 1998 ACM Int. Symp. on Advances in Geographic Information Systems*. ACM, 163–164.
- R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics* 5 (1979), 287–326.
- Klaus Jansen and Roberto Solis-Oba. 2003. An asymptotic fully polynomial time approximation scheme for bin covering. *Theor. Comput. Sci.* 306, 1-3 (2003), 543–551.
- Sanjeev Khanna, S. Muthukrishnan, and Mike Paterson. 1998. On Approximating Rectangle Tiling and Packing. In *SODA*, Howard J. Karloff (Ed.). ACM/SIAM, 384–393.
- Jon M. Kleinberg and Éva Tardos. 2006. *Algorithm design*. Addison-Wesley. I–XXIII, 1–838 pages.
- J. K. Lenstra, D. B. Shmoys, and É. Tardos. 1990. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.* 46, 3 (1990), 259–271. DOI: <http://dx.doi.org/10.1007/BF01585745>
- F. Manne. 1993. *Load Balancing in Parallel Sparse Matrix Computation*. Ph.D. Dissertation. University of Bergen, Norway.

- S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. 1999. On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity, and Applications. In *ICDT*. 236–256.
- Jonathan P. Sharp. 1999. Tiling Multi-dimensional Arrays. In *FCT (Lecture Notes in Computer Science)*, Gabriel Ciobanu and Gheorghe Păun (Eds.), Vol. 1684. Springer, 500–511.
- Adam Smith and Subhash Suri. 2000. Rectangular Tiling in Multidimensional Arrays. *J. Algorithms* 37, 2 (2000), 451–467.
- W. T. Tutte. 1956. A theorem on planar graphs. *Trans. Amer. Math. Soc.* 82 (1956), 99–116.