Algorithmica (2017) 77:1105–1127
DOI 10.1007/s00453-016-0137-8

A Constant Factor Approximation Algorithm for the Storage Allocation Problem

Reuven Bar-Yehuda[1] · **Michael Beder**[1] ·
Dror Rawitz[2]

Received: 22 April 2014 / Accepted: 25 February 2016 / Published online: 3 March 2016

Abstract We study the storage allocation problem (SAP) which is a variant of the unsplittable flow problem on paths (UFPP). A SAP instance consists of a path $P = (V, E)$ and a set J of tasks. Each edge $e \in E$ has a capacity c_e and each task $j \in J$ is associated with a path I_j in P, a demand d_j and a weight w_j. The goal is to find a maximum weight subset $S \subseteq J$ of tasks and a height function $h : S \to \mathbb{R}^+$ such that (i) $h(j) + d_j \leq c_e$, for every $e \in I_j$; and (ii) if $j, i \in S$ such that $I_j \cap I_i \neq \emptyset$ and $h(j) \geq h(i)$, then $h(j) \geq h(i) + d_i$. SAP can be seen as a rectangle packing problem in which rectangles can be moved vertically, but not horizontally. We present a polynomial time $(9 + \varepsilon)$-approximation algorithm for SAP. Our algorithm is based on a variation of the framework for approximating UFPP by Bonsma et al. [FOCS 2011] and on a $(4 + \varepsilon)$-approximation algorithm for δ-small SAP instances (in which $d_j \leq \delta \cdot c_e$, for every $e \in I_j$, for a sufficiently small constant $\delta > 0$). In our algorithm for δ-small instances, tasks are packed carefully in strips in a UFPP manner, and then a $(1 + \varepsilon)$ factor is incurred by a reduction from SAP to UFPP in strips. The strips are stacked to form a SAP solution. Finally, we provide a $(10 + \varepsilon)$-approximation algorithm for SAP on ring networks.

A preliminary version was presented at the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2013.

✉ Dror Rawitz
dror.rawitz@biu.ac.il

Reuven Bar-Yehuda
reuven@cs.technion.ac.il

Michael Beder
bederml@cs.technion.ac.il

1 Department of Computer Science, Technion, 32000 Haifa, Israel

2 Faculty of Engineering, Bar-Ilan University, 52900 Ramat Gan, Israel

Keywords Approximation algorithms · Bandwidth allocation · Rectangle packing · Storage allocation · Unsplittable flow

1 Introduction

In the UNSPLITTABLE FLOW PROBLEM ON PATHS (UFPP) an instance consists of a path $P = (V, E)$ with m edges and a set J of n tasks. Each edge $e \in E$ has a capacity c_e. Each task $j \in J$ has a starting vertex $s_i \in V$, ending vertex $t_i \in V$, a demand d_j and a weight w_j. We denote the path from s_j to t_j by I_j, and we say that $j \in J$ *uses* an edge $e \in E$ if $e \in I_j$. Given a set S of tasks and an edge $e \in E$, define $S(e) = \{j \in S : e \in I_j\}$ to be the set of tasks in S that use e. A feasible UFPP solution is a set of tasks $S \subseteq J$ such that $\sum_{j \in S(e)} d_j \leq c_e$, for every $e \in E$. The goal in UFPP is to find a feasible solution of maximum weight.

We study a variant of UFPP called the STORAGE ALLOCATION PROBLEM (SAP). In SAP we have an additional constraint: it is also required that every task in the solution is given the same contiguous portion of the resource in every edge along its path. More formally, a feasible SAP solution is a subset $S \subseteq J$ and a height function $h : S \to \mathbb{R}^+$ such that

1. $h(j) + d_j \leq c_e$, for every $j \in S$ and $e \in I_j$, and
2. if $j, i \in S$ such that $I_j \cap I_i \neq \emptyset$ and $h(j) \geq h(i)$, then $h(j) \geq h(i) + d_i$.

It follows that SAP is a rectangle packing problem in which each rectangle of height d_j can be moved vertically, but not horizontally. We note that while any SAP solution induces a UFPP solution, the converse is not always true, as shown in Fig. 1.

SAP naturally arises in scenarios where tasks require contiguous static portions of a resource. An object may require a contiguous range of storage space (e.g., memory allocation) for a specific time interval ($[s_j, t_j)$ for task j). A task may require bandwidth, but will only accept a contiguous set of frequencies or wavelengths. The resource may be a banner, where each task is an advertisement that requires a contiguous portion of the banner.

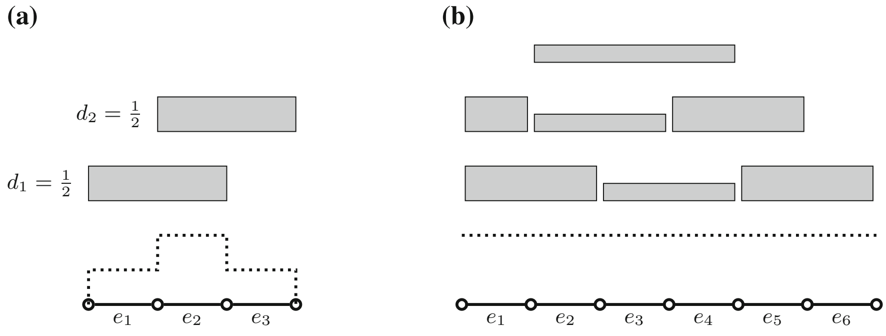


Fig. 1 The *dotted line* represents the capacity of the edges, and the strips correspond to tasks. *Thick strips* have demand $\frac{1}{2}$, while *thin strips* have demand $\frac{1}{4}$. The task sets in both instances form UFPP solutions. However, in both instances there is no SAP solution that contains all tasks (the instance on the right was given in [18]). **a** $c_{e_1} = c_{e_3} = 0.5$ and $c_{e_2} = 1$. **b** $c_e = 1$ for every e

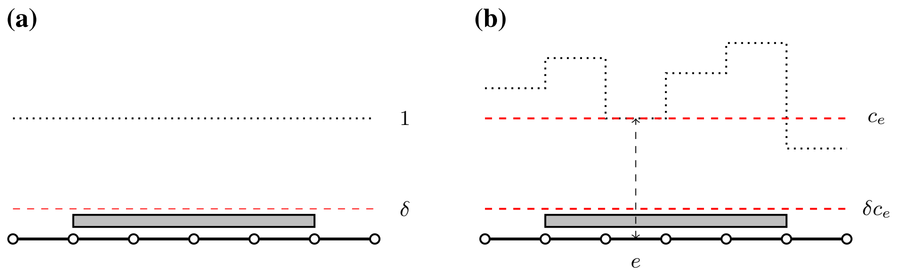


Fig. 2 Example of δ-small tasks. **a** Uniform capacities. **b** Non uniform capacities

Given a SAP or a UFPP instance, an edge $e \in E$ is called a *bottleneck edge* of a task j, if $c_e = \min_{f \in I_j} c_f$. Define $b(j) \triangleq \min_{f \in I_j} c_f$, namely $b(j)$ is the capacity of a bottleneck edge of j. Given $\delta > 0$, a task j is called δ-*small* if $d_j \leq \delta b(j)$, otherwise it is called δ-*large*. A SAP or a UFPP instance is called δ-*small* (δ-*large*) if $d_j \leq \delta b(j)$ ($d_j > \delta b(j)$), for every $j \in J$ (see example in Fig. 2). In the special case of SAP with uniform capacities (SAP- U), all edges in I_j are bottleneck edges, for every task j. The same goes for UFPP with uniform capacities (UFPP- U). An instance in which the maximum demand is bounded by the minimum edge capacity, i.e., $\max_j d_j \leq \min_e c_e$, is said to satisfy the *no-bottleneck assumption* (NBA).

SAP- U is strongly related to the Dynamic Storage Allocation problem (DSA), where one is given a path and a set of tasks, and the goal is to find the minimum uniform capacity for all edges together with a SAP solution that contains all tasks.

1.1 Related Work

The special case of SAP- U (or UFPP- U) with unit capacities and demands is the Maximum Independent Set problem in interval graphs which is solvable in polynomial time (see, e.g., [25]). Both SAP- U and UFPP- U are NP-hard, since they contain Knapsack as the special case in which the paths of all requests share an edge. When the number of edges in P is constant, UFPP is a special case of Mutli- Dimensional Knapsack and hence admits a PTAS [21].

Bar-Noy et al. [5] designed local ratio algorithms for UFPP- U and SAP- U with ratio 3 and 7, respectively. The latter was obtained using a reduction from SAP- U to UFPP- U that was based on an algorithm for the Dynamic Storage Allocation Problem (DSA) by Gergov [24]. An extension of SAP- U in which each task j has a time window was studied in [5,26]. Calinescu et al. [13] developed a randomized approximation algorithm for UFPP- U with expected performance ratio of $2 + \varepsilon$, for every $\varepsilon > 0$. They obtained this result by dividing the given instance into an instance with large tasks and an instance with small tasks. They use dynamic programming to compute an optimal solution for the large instance, and a randomized LP-based algorithm to obtain a $(1 + \varepsilon)$-approximate solution for the small instance. They also present a 3-approximation algorithm for UFPP- U that is different from the one given in [5].

Chen et al. [18] studied the special case of SAP- U where the capacity is K, where K is an integer, and all demands are integers in the range $\{1, \ldots, K\}$. They developed an $O(n(nK)^K)$ time dynamic programming algorithm to solve this special case of SAP- U, and also gave an approximation algorithm with ratio $\frac{e}{e-1} + \varepsilon$, for any $\varepsilon > 0$, assuming that $d_j = O(1)$, for every j. Bar-Yehuda et al. [6] presented approximation algorithms for SAP- U that is based on a reduction from SAP- U to UFPP- U that works on very small instances, namely on instances in which $d_j \leq \delta$, for a constant $\delta > 0$. (Here we assume that the uniform capacity is 1). The reduction is based on an algorithm for DSA by Buchsbaum et al. [12]. Bar-Yehuda et al. also presented a dynamic programming algorithm for large instances of SAP- U, and this led to two approximation algorithms for SAP- U; a randomized algorithm with ratio $2 + \varepsilon$ and a deterministic algorithm with ratio $\frac{2e-1}{e-1} + \varepsilon < 2.582$.

Bansal et al. [3] gave a deterministic quasi-polynomial time approximation scheme for instances of UFPP assuming that all capacities and demands are quasi-polynomial, thereby ruling out an APX-hardness result for such instances of UFPP, unless NP $\subseteq$ DTIME$(2^{\text{polylog}(n)})$. Batra et al. [9] improved the above result by removing the assumption. They also present two PTASs, one for the case where weight by demand ratios lie in a constant range, and another for the case where one is allowed to shorten paths by an ϵ-fraction.

Chakrabarti et al. [14] presented a constant factor approximation algorithm for UFPP under the NBA and an $O(\log(d_{\max}/d_{\min}))$-approximation algorithm for UFPP by extending the approach of [13]. Chekuri et al. [17] used an LP-based deterministic algorithm to obtain a $(2 + \varepsilon)$-approximation algorithm for UFPP under the NBA. Bansal et al. [4] developed an $O(\log n)$-approximation algorithm for UFPP, beating the integrality gap of the natural LP-relaxation, which was shown to be $\Omega(n)$ [14]. Chekuri et al. [16] considered UNSPLITTABLE FLOW on trees. They generalized the above result by providing an $O(\log m)$-approximation algorithm for uniform weights and an $O(\log m \cdot \min\{\log m, \log n\})$-approximation algorithm for the weighted case, where m is the number of edges in the tree. They also developed an LP formulation for UFPP that has an $O(\log n \cdot \min\{\log m, \log n\})$ integrality gap and admits a polynomial time $O(1)$-approximation algorithm. (In a more recent version of their paper [15] they provide a polynomial-size LP with an integrality gap of $O(\log m)$.) A linear program for UFPP whose gap is $7 + \varepsilon$ was given by Anagnostopoulos et al. [1]. Friggstad and Gao [22] present a polynomial-size linear program for UNSPLITTABLE FLOW on trees whose integrality gap is $O(\log n \cdot \min\{\log m, \log n\})$.

Bonsma et al. [10] developed a $(7 + \varepsilon)$ approximation algorithm for UFPP and showed that UFPP is strongly NP-hard even for instances with demands in $\{1, 2, 3\}$. Chrobak et al. [19] showed that UFPP- U is strongly NP-hard even for the case of uniform weights. Recently, Anagnostopoulos et al. [2] presented a $(2+\varepsilon)$-approximation algorithm for UFPP.

Gergov [24] presented an $O(n \log n)$ time algorithm for DSA that computes a solution of cost at most $3\text{LOAD}(J)$, where $\text{LOAD}(J)$ is the maximum sum of demands on an edge. Buchsbaum et al. [12] presented a polynomial time algorithm that computes a solution of cost at most $(1+O((D/\text{LOAD}(J))^{1/7}))\cdot\text{LOAD}(J)$, where D is the maximal demand of a task. Garey and Johnson [23, Problem SR2] stated that Larry Stockmeyer showed that DSA is strongly NP-hard using a reduction from 3- PARTITION. The reduction is given in [11]. In fact, in the above reduction only two demand types are used. This hardness results implies that SAP- U is also strongly NP-hard.

Finally, we note that it can be shown that SAP- U is strongly NP-hard using a different reduction from 3- PARTITION that was used to show that CALL SCHEDULING with unit bandwidth and arbitrary duration is strongly NP-hard [20]. (The time dimension in call scheduling corresponds to the storage dimension in the storage allocation problem.) In the conference version of this paper [7] we used a relatively simple reduction from BIN PACKING, to show that SAP is strongly NP-hard, even with uniform weights and even under the NBA.

1.2 Our Contribution

We present a polynomial time $(9 + \varepsilon)$-approximation algorithm for SAP, for every constant $\varepsilon > 0$. Our algorithm is based on the recent constant factor approximation algorithm for UFPP by Bonsma et al. [10]. As done in [10] we partition the task set into three sets: *small* tasks, *medium* tasks, and *large* tasks. [1] Small tasks are δ-small for some $\delta > 0$, large tasks are δ'-large for some $\delta' > \delta$, and medium tasks are δ-large and δ'-small.

The algorithms for small and medium tasks by Bonsma et al. [10] are based on an approximation framework that provides $(1 + \varepsilon)\alpha$-approximate solutions given a certain type of α-approximation algorithm for UFPP with "almost uniform" capacities ($c_e \in [2^k, 2^{k+\ell})$, for some k and a constant ℓ). Our algorithm for medium tasks uses a variation of this framework for SAP. The main difference is that in SAP we also need to worry about the height assignments. Additionally, we provide a 2-approximation algorithm for "almost uniform" instances. We do this by extending the dynamic programming algorithm for SAP with uniform capacities from [6] to "almost uniform" capacities. A factor 2 is lost due to the framework's requirement from the α-approximation algorithm for "almost uniform" instances. Hence, combined with the above framework we obtain a $(2+\varepsilon)$-approximation algorithm for medium tasks.

Our $(4 + \varepsilon)$-approximation algorithm for small tasks is based on partitioning the instance into instances in which bottlenecks are within factor 2 of each other. We show how to compute an approximate solution for each instance and then explain how to adjust the heights in order to combine them. This can be seen as a variant of the framework for medium tasks, in which the α-approximation algorithm should satisfy an additional requirement: the tasks must be packed in a strip. We use an LP-rounding $(4 + \varepsilon)$-approximation algorithm for the UFPP version of each such instance that computes approximate solutions in which tasks are packed in strips. (We also provide an alternative local ratio $(5+\varepsilon)$-approximation algorithm.) A $(1+\varepsilon)$ factor is incurred by a reduction from SAP to UFPP in strips [6]. Finally a SAP solution is obtained by stacking the strips.

As for large tasks, Bonsma et al. [10] presented an approximation algorithm for large instances of UFPP that is based on (i) a reduction from UFPP to a special

[1] Bonsma et al. [10] use tiny, medium, and large, since they consider both medium and tiny tasks as small tasks.

case of the Rectangle Packing problem, and (ii) an algorithm that solves this special case that correspond to instances that are obtained by the reduction. Their algorithm provides a schedule that is induced by a subset of pairwise non-intersecting rectangles, and therefore it is also a SAP schedule. It follows that this algorithm is also an approximation algorithm for large instances of SAP. In this paper, we give a tighter analysis and provide a better upper bound on the approximation ratio for large instances of SAP.

Finally, using a reduction from rings to paths we provide a $(10+\varepsilon)$-approximation algorithm for SAP on ring networks, where each task has two possible paths.

1.3 Paper Organization

The remainder of the paper is organized as follows. Section 2 contains definitions and a few preliminary observations. A formal description of our results is given in Sect. 3. Our algorithms for medium, small, and large SAP instances are given in Sects. 5, 4, and 6, respectively. We consider ring networks in Sect. 7. We conclude in Sect. 8.

2 Preliminaries

Given a task set $S \subseteq J$, the demand of S is denoted by $d(S)$, namely $d(S) \triangleq \sum_{j \in S} d_j$. The *load* of S on an edge e is defined as $d(S(e)) = \sum_{j \in S(e)} d_j$. A feasible UFPP solution is a set of tasks $S \subseteq J$ such that $d(S(e)) \leq c_e$, for every $e \in E$. A UFPP solution S is called B-*packable* if $d(S(e)) \leq B$, for every $e \in E$. Given a SAP solution (S, h), the *makespan* of an edge e is defined as $\mu_h(S(e)) \triangleq \max_{j \in S(e)}(h(j) + d_j)$. Observe that $d(S(e)) \leq \mu_h(S(e))$, for every $e \in E$. A SAP solution (S, h) is called B-packable if $\mu_h(S(e)) \leq B$, for every $e \in E$.

Given two disjoint subsets $S_1, S_2 \subseteq J$ and two height functions $h_1 : S_1 \to \mathbb{R}^+$ and $h_2 : S_2 \to \mathbb{R}$, let $h_1 \cup h_2 : S_1 \cup S_2 \to \mathbb{R}$ be the following function:

$$(h_1 \cup h_2)(j) = \begin{cases} h_1(j) & j \in S_1, \\ h_2(j) & j \in S_2. \end{cases}$$

The following observation bounds the load of a UFPP solution on the edges in terms of the maximum bottleneck. A similar observation was made in [10].

Observation 1 *Let S be a feasible* UFPP *solution. Then $d(S(e)) \leq 2\max_{j \in S} b(j)$, for every $e \in E$.*

Proof Let e be an edge. Any task $j \in S(e)$ must use an edge with capacity at most $B = \max_{j \in S} b(j)$. Let e_L and e_R be the closest such edges to the left and to the right, respectively. (It may be that $e_L = e_R = e$.) Hence, $d(S(e)) \leq d(S(e_L)) + d(S(e_R)) \leq 2B$. □

The next observation is the analogous observation for SAP.

Observation 2 *Let (S, h) be a feasible* SAP *solution. Then $\mu_h(S(e)) \le \max_{j\in S} b(j)$, for every $e \in E$.*

Proof Let $e \in E$ be an edge and let $S(e) = \{j_1, \dots, j_p\}$ such that $h(j_i) + d_{j_i} \le h(j_{i+1})$, for every i. The observation follows, since $\mu_h(S(e)) = h(j_p) + d_{j_p} \le b(j_p) \le \max_{j\in S} b(j)$. □

Finally, we need the following standard result that is used when one partitions the input into small and large instances. Given a SAP instance, let J_S and J_L be the subset of δ-small tasks and the subset of δ-large tasks, respectively. (The proof is given for completeness.)

Lemma 3 *Let S_1 and S_2 be an r_1-approximate solution with respect to J_S and an r_2-approximate solution with respect to J_L, respectively. Then, the solution of greater weight is an $(r_1 + r_2)$-approximation for the original instance.*

Proof Let S^* be an optimal solution for the original instance. Either $w(S^* \cap J_S) \ge \frac{r_1}{r_1+r_2} w(S^*)$ or $w(S^* \cap J_L) \ge \frac{r_2}{r_1+r_2} \cdot w(S^*)$. Hence, either $w(S_1) \ge \frac{1}{r_1} \cdot \frac{r_1}{r_1+r_2} \cdot w(S^*) = \frac{1}{r_1+r_2} \cdot w(S^*)$ or $w(S_2) \ge \frac{1}{r_2} \cdot \frac{r_2}{r_1+r_2} \cdot w(S^*) = \frac{1}{r_1+r_2} \cdot w(S^*)$. The lemma follows. □

3 Statement of Results

In this section we provide a formal statement of our results. We start with our results regarding small, medium, and large instances.

Theorem 1 *There exists a polynomial time algorithm such that for every constant $\varepsilon > 0$, there exists a constant $\delta > 0$, such that the algorithm computes $(4 + \varepsilon)$-approximate solutions for δ-small* SAP *instances.*

Theorem 2 *There exists a polynomial time $(2 + \varepsilon)$-approximation algorithm for δ-large and $(1 - 2\beta)$-small* SAP *instances for every constants $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1 - 2\beta)$.*

Theorem 3 *There exists a polynomial time $(2k - 1)$-approximation algorithm for $\frac{1}{k}$-large* SAP *instances for every integer $k \ge 1$.*

The proofs of Theorems 1, 2, and 3 are given in Sects. 4, 5, and 6, respectively. Our result for general SAP instances follows.

Theorem 4 *There exists a polynomial time $(9+\varepsilon)$-approximation algorithm for* SAP, *for any constant $\varepsilon > 0$.*

Proof Set $k = 2$ and $\beta = \frac{1}{4}$. By Theorem 1 there exists a constant $\delta > 0$ for which there is a polynomial time $(4+\varepsilon)$-approximation algorithm for δ-small SAP instances. By Theorem 2 there is a $(2+\varepsilon)$-approximation algorithm for δ-large and $\frac{1}{2}$-small SAP instances. Also, there a polynomial time 3-approximation algorithm for $\frac{1}{2}$-large SAP instances by Theorem 3. The theorem follows from Lemma 3. □

In Sect. 7 we consider SAP on ring networks.

Theorem 5 *There exists a polynomial time $(10 + \varepsilon)$-approximation algorithm for* SAP *on ring networks, for any constant $\varepsilon > 0$.*

4 Small Tasks

In this section we prove Theorem 1, namely we present a polynomial time algorithm that, for every $\varepsilon > 0$, computes $(4+\varepsilon)$-approximate solutions for δ-small instance of SAP, for some constant $\delta > 0$ (depending on ε).

We first present an LP-rounding algorithm for UFPP instances in which bottlenecks are within factor 2 of each other. A $(1+\varepsilon)$ factor is incurred by a reduction from SAP to UFPP in strips [6]. Then, we show how to use this algorithm to design an algorithm for small instances. We partition the instance into instances in which tasks have similar bottlenecks, and then use the above algorithm to compute an approximate solution that resides in a strip. A SAP solution is obtained by stacking the strips.

4.1 Packing Small Tasks in Strips

As a first step we consider the following special case of SAP. Let $B > 0$, and assume we are given a δ-small SAP instance in which $b(j) \in [B, 2B)$, for every $j \in J$. Note that due to Observation 2, without loss of generality we may assume that all edge capacities are between B and $2B$ (see example in Fig. 3). We present a LP-rounding algorithm that computes a $\frac{1}{2}B$-packable $(4+\varepsilon)$-approximate SAP solution. An alternative local ratio $(5+\varepsilon)$-approximation algorithm is also provided in an appendix.

The first step is an LP-rounding algorithm that computes $\frac{1}{2}B$-packable UFPP solutions. The algorithm is based on the following integer linear formulation of UFPP:

$$\begin{aligned} \max \ & \sum_{j\in J} w_j \cdot x_j \\ \text{s.t.} \ & \sum_{j\in S(e)} d_j x_j \le c_e \quad && \forall e \in E \\ & x_j \in \{0,1\} && \forall j \in J \end{aligned} \tag{1}$$

where $x_j = 1$ represents that j is in the solution. An LP-relaxation is obtained by replacing the integrality constraints with $x_j \in [0, 1]$, for every $j \in J$.

Let x^* be an optimal fractional solution of (1) and define $x' = \frac{1}{4}x^*$. The solution x' satisfies $\sum_{j\in S(e)} d_j x_j \le \frac{1}{2}B$, and therefore it is feasible with respect to (1) with

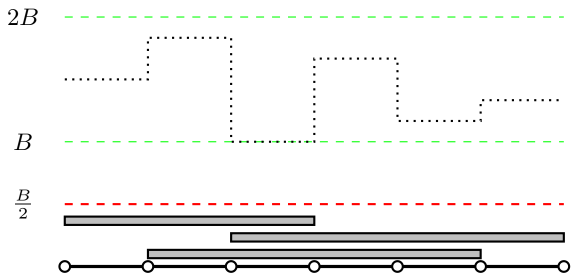


Fig. 3 Example of a $\frac{1}{2}B$-packable solution for a δ-small SAP instance in which $b(j) \in [B, 2B)$, for every $j \in J$

$c_e = \frac{1}{2}B$, for every e. Since this is a uniform capacity instance we may use the following result of Chekuri et al. [17] to obtain an integral solution.

Theorem 6 ([17]). *For every constant $\varepsilon > 0$, there exists a constant $\delta > 0$, such that given a δ-small instance of* UFPP- U*, an integral solution x such that $\sum_j w_j x_j \geq \frac{1}{1+\varepsilon}\sum_j w_j x_j^*$ can be found in polynomial time, where x^* is an optimal fractional solution of* (1) .

We now transform our UFPP- U solution into a SAP solution using the following result:

Lemma 4 ([6]). *There exists a constant $\delta_0 > 0$, such that if S is a B-packable* UFPP *solution to some δ-small instance, where $\delta \in (0, \delta_0)$, then S can be transformed into a B-packable* SAP *solution (S', h') such that $w(S') \geq (1-4\delta)w(S)$ in polynomial time.*

Using Lemma 4 we obtain an approximate SAP solution.

Lemma 5 *For every constant $\varepsilon > 0$, there exists a constant $\delta > 0$ and a polynomial time algorithm that computes $\frac{1}{2}B$-packable $(4+\varepsilon)$-approximate solutions for δ-small* SAP *instances in which $b(j) \in [B, 2B)$, for every $j \in J$.*

Proof Given $\varepsilon > 0$, set δ such that (i) $\delta \leq \delta_1$, where δ_1 is the constant that is required by Theorem 6 for $\varepsilon/5$, (ii) $\delta < \delta_0$, and (iii) $1-4\delta > (4+\frac{4}{5}\varepsilon)/(4+\varepsilon)$ (e.g., $\delta \leq \varepsilon/100$ would suffice). Apply the algorithm from [17] to compute a $\frac{1}{2}B$-packable $4\cdot(1+\frac{\varepsilon}{5})$-approximate UFPP solution S. By Lemma 4, S can be transformed into a $\frac{1}{2}B$-packable SAP solution (S', h') such that $w(S') \geq (1-4\delta)w(S)$ in polynomial time. It follows that

$$w(S') \geq \frac{1-4\delta}{4\cdot(1+\varepsilon/5)} \cdot \text{OPT}_{\text{UFPP}}(J) \geq \frac{1}{4+\varepsilon} \cdot \text{OPT}_{\text{SAP}}(J) \ ,$$

as required. □

4.2 Stacking Strips

The next step is to partition the instance. Let $J_t = \left\{j \in J : 2^t \leq b(j) < 2^{t+1}\right\}$, for every t. Algorithm **Strip-Pack** computes an approximate solution for J_t, for each t, and then combines the solutions. An example of a solution produced by Algorithm **Strip-Pack** is shown in Fig. 4.

We conclude the section by showing that Algorithm **Strip-Pack** computes $(4+\varepsilon)$-approximate solutions, provided that it uses the algorithm whose existence is shown in Lemma 5 to compute the 2^{t-1}-packable solutions (Line 2).

Proof of Theorem 1 First, the running time of Algorithm **Strip-Pack** is polynomial, since there are at most $O(n)$ nonempty subsets J_t, and for each such subset we call the algorithm from Lemma 5 which runs in polynomial time.

Algorithm 1 : Strip-Pack (J, w)

```
1: for each $t$ do
2:     Compute a $2^{t-1}$-packable SAP solution $(S_t, h_t)$ for $J_t$
3:     $h'_t(j) = h_t(j) + 2^{t-1}$, for every $j \in J_t$
4: end for
5: $S \leftarrow \bigcup_t S_t, h \leftarrow \bigcup_t h'_t$
6: Return $(S, h)$
```

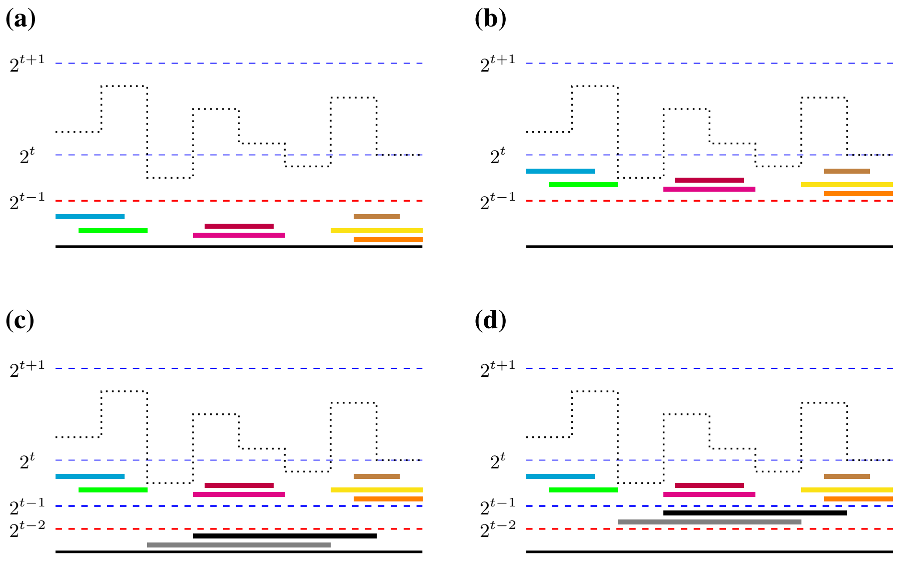


Fig. 4 An example of a solution produced by Algorithm **Strip-Pack**. **a** Computing solution for J_t. **b** Lifting solution for J_t. **c** Computing solution for J_{t-1}. **d** Lifting solution for J_{t-1}

By Lemma 5 we have that Algorithm **Strip-Pack** computes a 2^{t-1}-packable $(4+\varepsilon)$-approximate solution (S_t, h_t) for J_t, for every t. By lifting the solution (S_t, h_t) by 2^{t-1}, Algorithm **Strip-Pack** ensures that a feasible SAP solution is obtained. Also, let (S^*, h^*) be an optimal solution for J. Then,

$$w(S) = \sum_t w(S_t) \geq \frac{1}{4+\varepsilon} \sum_t \text{OPT}_{\text{SAP}}(J_t) \geq \frac{1}{4+\varepsilon} \sum_t w(S^* \cap J_t) = \frac{1}{4+\varepsilon} \cdot w(S^*) ,$$

as required. □

5 Medium Tasks

In this section we prove Theorem 2. We present a polynomial time algorithm that computes $(2+\varepsilon)$-approximate solutions for instances of SAP that are both δ-large and $(1-2\beta)$-small, for any constants $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1-2\beta)$.

5.1 Approximation Framework for SAP

Following [10], we present a framework that acts as a reduction from a SAP instance to multiple "almost uniform" SAP instances. Given an α-approximation algorithm for almost uniform instances, the framework provides a $(1+\varepsilon)\alpha$-approximation algorithm. As opposed to the framework from [10] that was designed for UFPP, our framework has an additional difficulty which is taking care of height assignments.

Let $k \in \mathbb{Z}$ and $\ell \in \mathbb{N}$. Given a SAP instance, let $J^{k,\ell} = \left\{ j \in J : 2^k \leq b(j) < 2^{k+\ell} \right\}$ and let $E^{k,\ell} = \cup_{j \in J^{k,\ell}} I_j$. We observe that without loss of generality, we may assume that for each $J^{k,\ell}$, edge capacities are between 2^k and $2^{k+\ell}$.

Observation 6 *Given a* SAP *instance, $k \in \mathbb{Z}$, and $\ell \in \mathbb{N}$, we have that $c_e \geq 2^k$, for every $e \in E^{k,\ell}$.*

Proof If $j \in J^{k,\ell}$, then $c_e \geq b(j) \geq 2^k$, for every $e \in I_j$. □

Observation 7 *Let (S, h) be a feasible* SAP *solution such that $S \subseteq J^{k,\ell}$. Then $\mu_h(S(e)) \leq \min(c_e, 2^{k+\ell})$, for every edge $e \in E$.*

Proof Observation 2 implies that any feasible SAP solution $S \subseteq J^{k,\ell}$ is $2^{k+\ell}$-packable. □

Thus, from the view point of tasks in $J^{k,\ell}$, the capacity of $e \in E^{k,\ell}$ is $\min(c_e, 2^{k+\ell})$.

Define $q = \lceil \log_2(1/\beta) \rceil$, and let $\ell \in \mathbb{N}$ be a constant that will be determined later. Algorithm **AlmostUniform** is our framework for computing SAP solutions, and it is based on the framework for UFPP that was given in [10]. The main difference is that with SAP one cannot simply combine sub-solutions. A height function for the tasks should also be computed. This motivates the following definition.

Definition 1 Fix k and ℓ and let $\beta > 0$. A feasible SAP solution (S, h) where $S \subseteq J^{k,\ell}$ is called *β-elevated* (with respect to k) if $h(j) \geq \beta 2^k$, for every $j \in S$.

Note that it may be the case that $S \subseteq J^{k_1,\ell_1}, J^{k_2,\ell_2}$, where $k_1 < k_2$. Moreover, it may be that a feasible solution (S, h) is β-elevated with respect to k_1, but not with respect to k_2.

Algorithm **AlmostUniform** uses an algorithm called **Elevator** that computes an α-approximate SAP solution for $J^{k,\ell}$ which is also β-elevated (with respect to k). Notice that a necessary condition for the existence of such a nonempty SAP solution is that there are $(1-\beta)$-small tasks in $J^{k,\ell}$.

Since ℓ is a constant, the number of subsets $J^{k,\ell}$ is linear. Hence, if the running time of Algorithm **Elevator** is polynomial, then the running time of Algorithm **AlmostUniform** is also polynomial. It remains to show that the computed solution is indeed $(1+\varepsilon)\alpha$-approximate, for an appropriate choice of ℓ.

Lemma 8 *Let J be a set of δ-large and $(1-2\beta)$-small tasks, where $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1-2\beta)$ are constants. The solution (S_r, h_r) computed by Algorithm **AlmostUniform** is a feasible* SAP *solution, for every $r \in \{0, \ldots, \ell + q - 1\}$, where $q = \lceil \log_2(1/\beta) \rceil$.*

Algorithm 2 : **AlmostUniform** (J, ℓ)

```
1: $\mathcal{K} \leftarrow \left\{k \in \mathbb{Z} : J^{k,\ell} \neq \emptyset\right\}$
2: for each $k \in \mathcal{K}$ do
3:     $(S^{k,\ell}, h^{k,\ell}) \leftarrow$ Elevator$(J^{k,\ell}, \beta)$
4: end for
5: for each $r \in \{0, \ldots, \ell + q - 1\}$ do
6:     Let $\mathcal{K}(r) = \mathcal{K} \cap \{r + i \cdot (\ell + q) : i \in \mathbb{Z}\}$
7:     $S_r \leftarrow \bigcup_{k\in\mathcal{K}(r)} S^{k,\ell}$, $h_r \leftarrow \bigcup_{k\in\mathcal{K}(r)} h^{k,\ell}$
8: end for
9: $r^* \leftarrow \text{argmax}_{r\in\{0,\ldots,\ell+q-1\}}\, w(S_r)$
10: Return $(S_{r^*}, h_{r^*})$
```

Proof Given r, let $k_0 = \min \mathcal{K}(r)$. Also given $i \in \mathcal{K}(r)$, let $i^+ = \min\{k \in \mathcal{K}(r) : k > i\}$. For $i \in \mathcal{K}(r)$, let $S_i = \bigcup_{k\in\mathcal{K}(r),k\leq i} S^{k,\ell}$ and let $h_i = \bigcup_{k\in\mathcal{K}(r),k\leq i} h^{k,\ell}$. We prove that (S_i, h_i) is feasible by induction on i. In the base case we have $i = k_0$, and we have that $(S_i, h_i) = (S^{k_0,\ell}, h^{k_0,\ell})$ is feasible due to our assumption on Algorithm **Elevator**. For the inductive step, we assume that the claim holds for i and prove that it holds for i^+. We know that (S_i, h_i) is feasible due to the inductive hypothesis, and by Observation 7 we know that $\mu_{h_i}(S_i(e)) \leq \min(c_e, 2^{i+\ell})$, for every edge $e \in E$. Since **Elevator** computes a β-elevated SAP solution for $J^{i^+,\ell}$, it follows that

$$h_{i^+}(j) \geq \beta \cdot 2^{i^+} \geq 2^{-q} \cdot 2^{i^+} \geq 2^{-q} \cdot 2^{i+\ell+q} \geq 2^{i+\ell} ,$$

for every $j \in S_{i^+}$. Hence (S_{i^+}, h_{i^+}) is a feasible SAP schedule. □

The UFPP version of the following lemma appeared in [10] and applies here as well. We provide a proof for completeness.

Lemma 9 *Let J be a set of δ-large and $(1-2\beta)$-small tasks, where $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1 - 2\beta)$ are constants. If Algorithm **Elevator** computes α-approximate solutions, then $w(S_{r^*}) \geq \frac{\ell}{\ell+q} \cdot \frac{1}{\alpha}\text{OPT}_{\text{SAP}}(J)$, where $q = \lceil \log_2(1/\beta) \rceil$.*

Proof Let (S, h) be an optimal SAP solution for J. Since each $S^{k,\ell}$ is a β-elevated α-approximation for $J^{k,\ell}$ and every task $j \in J$ belongs to exactly ℓ sets $J^{k,\ell}$, it follows that

$$\begin{aligned}
\sum_{r=0}^{\ell+q-1} w(S_r) &= \sum_{r=0}^{\ell+q-1} \sum_{k\in\mathcal{K}(r)} w(S^{k,\ell}) \\
&\geq \sum_{r=0}^{\ell+q-1} \sum_{k\in\mathcal{K}(r)} \frac{1}{\alpha} \cdot \text{OPT}_{\text{SAP}}(J^{k,\ell}) \\
&= \frac{1}{\alpha} \cdot \sum_{k\in\mathcal{K}} \text{OPT}_{\text{SAP}}(J^{k,\ell}) \\
&\geq \frac{1}{\alpha} \cdot \sum_{k\in\mathcal{K}} w(S \cap J^{k,\ell})
\end{aligned}$$

$$= \frac{\ell}{\alpha} \cdot \mathrm{OPT}_{\mathrm{SAP}}(J) .$$

Therefore, $w(S_{r^*}) \geq \frac{1}{\alpha} \cdot \frac{\ell}{\ell+q} \cdot \mathrm{OPT}_{\mathrm{SAP}}(J)$. □

By choosing the right value of ℓ we obtain a $(1+\varepsilon)\alpha$-approximation algorithm.

Lemma 10 *Let J be a set of δ-large and $(1-2\beta)$-small tasks, where $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1-2\beta)$ are constants. Suppose we are given a polynomial time algorithm that computes a β-elevated α-approximate* SAP *solution for $J^{k,\ell}$, for every k and ℓ. Then, if $\ell = \frac{1}{\varepsilon}\lceil \log_2(1/\beta) \rceil$, Algorithm **AlmostUniform** computes a $(1+\varepsilon)\alpha$-approximate solution in polynomial time.*

Proof We know that the computed solution is feasible due to Lemma 8, and by Lemma 9 we have that

$$w(S_{r^*}) \geq \frac{1}{\alpha} \cdot \frac{\ell}{\ell + \lceil \log_2(1/\beta) \rceil} \cdot \mathrm{OPT}_{\mathrm{SAP}}(J) = \frac{1}{\alpha} \cdot \frac{1}{1+\varepsilon} \cdot \mathrm{OPT}_{\mathrm{SAP}}(J) ,$$

as required. □

5.2 Computing 2-Approximations that are β-Elevated

In this section we present an algorithm that computes a 2-approximation solution for $J^{k,\ell}$ which is also β-elevated, for any k and ℓ. Throughout the section we consider medium tasks, namely we assume that every task $j \in J^{k,\ell}$ is δ-large and $(1-2\beta)$-small, for constants $\varepsilon > 0$, $\beta \in (0, \frac{1}{2})$, and $\delta \in (0, 1-2\beta)$.

Our algorithm is based on the following simple observation that was given in [6] for SAP- U.

Observation 11 *Given a* SAP *instance, there exists an optimal solution (S, h) such that, for every task j, either $h(j) = 0$ or there exists a task $j' \neq j$ such that $I_j \cap I_{j'} \neq \emptyset$ and $h(j) = h(j') + d_{j'}$.*

The proof of the observation uses a "gravity" argument, namely as long as there is a task whose height can be decreased, pick one such task and decrease its height as much as possible. (Alternatively, one could consider a height function h' that minimizes $\sum_{j \in S} h(j)$.) See example in Fig. 5.

Using Observation 11 we are able to consider a specific type of optimal solutions.

Lemma 12 *Suppose we are given a δ-large* SAP *instance, where $c_e \in [B, B2^\ell)$, for every $e \in E$, for some B. Then there exists an optimal solution (S^*, h^*) such that:*

(i) *$|S^*(e)| < 2^\ell/\delta$, for every e, and*
(ii) *there exists a subset $H_j \subseteq S^* \setminus \{j\}$ of size at most $2^\ell/\delta$ such that $h^*(j) = d(H_j)$, for every task $j \in S^*$.*

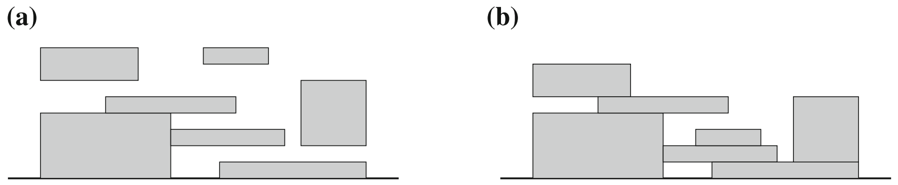


Fig. 5 Solution (**b**) is obtained by applying gravity on Solution (**a**). **a** Original solution. **b** Solution after application of gravity

Proof Let (S^*, h^*) be an optimal SAP solution whose existence is implied in Observation 11. To prove (i) observe that $d_j \geq \delta b(j) \geq \delta B$, for every $j \in S^*$ and that $c_e < B2^\ell$, for every $e \in E$. Thus from the feasibility of S^*, it follows that each edge $e \in E$ is used by less than $B2^\ell/(\delta B) = 2^\ell/\delta$ tasks. (ii) follows from Observation 11 and (i), by induction over the height. □

Lemma 12 implies an upper bound on the number of possibilities for the height of a task $j \in J$, given a δ-large SAP instance, where $c_e \in [B, B2^\ell)$, for every $e \in E$, for some B. Since the maximal number of tasks assigned to an edge is at most $L = 2^\ell/\delta$, the number of possible heights is bounded by $\sum_{i=0}^{L} \binom{n}{i} = O(n^L)$. It follows that there are at most $O(n^{O(L^2)})$ possibilities for assigning a task set and its corresponding heights to a given edge $e \in E$. Therefore, an optimal SAP solution for J can be computed using a dynamic programming algorithm similar to the one described in [6].

Lemma 13 *There is a polynomial time algorithm that computes an optimal solution for a δ-large* SAP *instance, where $c_e \in [B, B2^\ell)$, for every $e \in E$, for some B.*

Proof Let $V = \{v_0, \ldots, v_m\}$ and $E = \{e_1, \ldots, e_m\}$. Given a vertex $v_i \in V$, let P_i be the path that is induced by $V_i = \{v_i, \ldots, v_m\}$. Let J_i be the tasks that are fully contained in P_i. A feasible solution (S_i, h_i) is called *proper with respect to e_i* if $e_i \in I_j$, for every $j \in S_i$. Recall that there are $O(n^L)$ possibilities for choosing S_i, and that given S_i there are $O(n^{L^2})$ possibilities for choosing h_i. A solution (S_{i+1}, h_{i+1}) is *compatible* with the proper pair (S_i, h_i) if (i) it is proper with respect to e_{i+1}, (ii) Either $j \in S_i \cap S_{i+1}$ or $j \notin S_i \cap S_{i+1}$ for every j such that $e_i, e_{i+1} \in I_j$, and (iii) $h_i(j) = h_{i+1}(j)$ for every $j \in S_i \cap S_{i+1}$.

We define a dynamic programming table of size $O(n^{L+L^2})$ as follows. For an edge e_i and a pair (S_i, h_i) that is proper with respect to e_i the state $\Pi(e_i, S_i, h_i)$ stands for the maximum weight of a pair (S', h') such that $S' \subseteq J_{i+1}$ and $(S_i \cup S', h_i \cup h')$ is feasible. We initialize the table Π by setting $\Pi(e_m, S_m, h_m) = 0$ for every proper pair (S_m, h_m) with respect to e_m. We compute the rest of the entries by using:

$$\Pi(e_i, S_i, h_i) = \max_{\substack{(S_{i+1}, h_{i+1}) \\ \text{is compatible with} \\ (S_i, h_i)}} \{w(S_{i+1} \setminus S_i) + \Pi(e_{i+1}, S_{i+1}, h_{i+1})\}$$

The weight of an optimal solution is $\Pi(e_0, \emptyset, h_\emptyset)$, where e_0 is a dummy edge and $h_\emptyset$ is a function whose domain is the empty set.

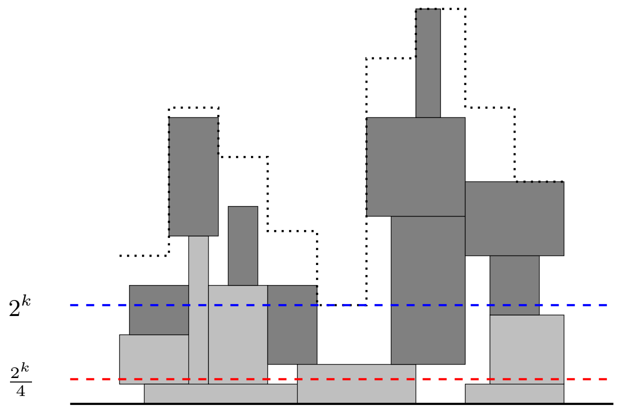


Fig. 6 An example of partition of optimal solution into two $\frac{1}{4}$-elevated solutions. The *light* tasks belong to S_1, while the *dark* tasks belong to S_2

To compute each entry $\Pi(e_i, S_i, h_i)$ we need to go through all the possibilities for a solution (S_{i+1}, h_{i+1}) that is compatible with (S_i, h_i). There are no more than $O(n^{L+L^2})$ such possibilities. Hence, the total running time is $O(m \cdot n^{L+L^2} \cdot n^{(L+L^2)}) = O(m \cdot n^{O(L^2)})$. In order to compute a corresponding solution, one needs to keep track of which option was taken in the recursive computation. An optimal solution can be reconstructed in a top down manner. □

Lemma 13 implies that solving SAP on $J^{k,\ell}$ can be done in polynomial time. It remains to obtain a β-elevated solution.

Lemma 14 *Suppose we are given a* $(1-2\beta)$*-small* SAP *instance. A* SAP *solution* (S, h) *for* $J^{k,\ell}$ *can be partitioned into two* β*-elevated* SAP *solutions* (S_1, h_1) *and* (S_2, h_2) *in linear time.*

Proof Consider a task $j \in S$ such that $h(j) < \beta 2^k$. Since j is $(1-2\beta)$-small and $c_e \geq 2^k$, for every $e \in E^{k,\ell}$, due to Observation 6, we have that

$$\begin{aligned} h(j) + d(j) &< \beta 2^k + (1-2\beta) b(j) \\ &\leq \beta 2^k + (1-2\beta) c_e \\ &= c_e + \beta 2^k - 2\beta c_e \\ &\leq c_e - \beta 2^k \,, \end{aligned} \tag{2}$$

for every $e \in I_j$. Define $S_1 = \left\{ j \in S : h(j) < \beta 2^k \right\}$ and $S_2 = S \setminus S_1$. Also, define $h_1(j) = h(j) + \beta 2^k$, for all $j \in S_1$, and $h_2(j) = h(j)$, for all $j \in S_2$ (see example in Fig. 6) (S_1, h_1) and (S_2, h_2) are both β-elevated by definition, and (S_1, h_1) is feasible due to (2). Finally, it is not hard to verify that the described partition can be done in linear time. □

The 2-approximation algorithm follows due to Lemmas 13 and 14.

Lemma 15 *There is a polynomial time algorithm that computes β-elevated 2-approximations for $J^{k,\ell}$, given a δ-large and $(1-2\beta)$-small* SAP *instance.*

Proof An optimal solution (S^*, h^*) for $J^{k,\ell}$ can be computed in polynomial time due to Lemma 13. (S^*, h^*) can be partitioned into two β-elevated solutions (S_1, h_1) and (S_2, h_2) due to Lemma 14. Since $w(S^*) = w(S_1) + w(S_2)$, one of the two solutions is 2-approximate. □

We note that it is also possible to use dynamic programming to find the optimal β-elevated solution for $J^{k,\ell}$ directly. Such a solution would be 2-approximate due to Lemma 14.

We conclude this section with the proof of Theorem 2.

Proof of Theorem 2 By Lemma 15 there exists a polynomial time algorithm that computes β-elevated 2-approximate solutions for $J^{k,\ell}$, for every k and ℓ. Therefore, by Lemma 10, Algorithm **AlmostUniform** is a $(2+\varepsilon)$-approximation algorithm for δ-large and $(1-2\beta)$-small SAP instances. □

6 Large Tasks

In this section we consider $\frac{1}{k}$-large instances of SAP, for an integer $k \geq 1$. Recall that in such instances $d_j > \frac{1}{k} b(j)$, for every j. We present a $(2k-1)$-approximation algorithm for $\frac{1}{k}$-large instances of SAP.

Bonsma et al. [10] presented a $2k$-approximation algorithm for $\frac{1}{k}$-large UFPP instances, for any $k \geq 2$, that is based on a reduction from UFPP to a special case of Rectangle Packing (or Maximum Independent Set in rectangle intersection graphs). The reduction is as follows. Let $j \in J$ be a task. The *residual capacity* of j is defined as $\ell(j) \triangleq b(j) - d_j$. Task j is *associated* with the rectangle $R(j) = [s_j, t_j) \times [\ell(j), b(j))$. In SAP terms, it is the rectangle that is induced by assigning height $\ell(j)$ to j. See example in Fig. 7.

Let $\mathcal{R}(S) = \{R(j) : j \in S\}$ be the family of rectangles that is obtained from a subset $S \subseteq J$. Bonsma et al. [10] showed that the set of rectangles $\mathcal{R}(S)$ that correspond to a feasible UFPP solution S, can be colored using $2k$ colors such that any color induces

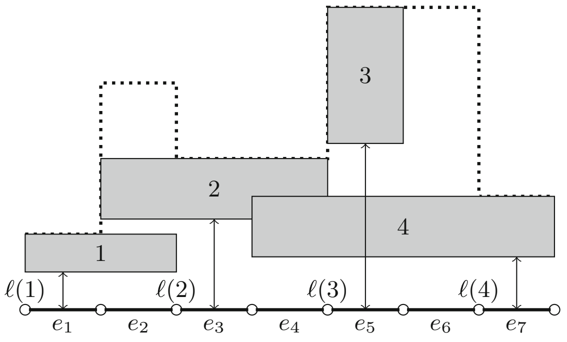


Fig. 7 An example of four tasks that are placed at height $\ell(j) = b(j) - d_j$, for every j

a pairwise non-intersecting subset of rectangles. Hence there exists at least one subset for which the total weight of the tasks is at least $\frac{1}{2k}w(S)$. Bonsma et al. also presented a polynomial time algorithm that solves the special case of Rectangle Packing that correspond to instances that are obtained by the above reduction.

Theorem 7 ([10]). *There is an $O(n^4)$ algorithm that computes an optimal rectangle packing of $\mathcal{R}(J)$, for every* UFPP *instance J.*

We note that the algorithm from [10] provides a UFPP solution which is induced by a subset of pairwise non-intersecting rectangles, and therefore it is also a SAP solution. It follows that this algorithm is also a $2k$-approximation algorithm for $\frac{1}{k}$-large instances of SAP. In what follows we use the geometric properties of SAP to show that $\mathcal{R}(S)$ can be colored using only $2k-1$ colors for any $\frac{1}{k}$-large SAP solution (S, h). This implies a $(2k-1)$-approximation algorithm for $\frac{1}{k}$-large instances of SAP, for any integer $k \geq 1$.

Given a feasible SAP solution (S, h) and a task $j \in S$, let

$$N_S(j) = \left\{j' \in S : R(j') \cap R(j) \neq \emptyset\right\} .$$

Notice that $j \in N_S(j)$. Let $\deg_S(R(j))$ be the number of rectangles in $\mathcal{R}(S)$ that intersect $R(j)$ (not including $R(j)$), namely $\deg_S(R(j)) = |N_S(j)| - 1$. We show that there exists a rectangle $R(j)$ whose degree is at most $2k-2$. This implies that a $(2k-1)$-coloring can be obtained in a greedy manner.

In the next lemma we show that there may be at most k $\frac{1}{k}$-large tasks that share the same an edge. Notice that it relies on Observation 2 that does not apply to UFPP.

Lemma 16 *Let (Q, h) be a $\frac{1}{k}$-large* SAP *solution that contains a task j' such that $\ell(j') < b(j) \leq b(j')$, for every $j \in Q$. If there exists an edge e such that $e \in I_j$, for every $j \in Q$, then $|Q| \leq k$.*

Proof Suppose that $|Q| > k$. Since $\ell(j') < b(j) \leq b(j')$, for every $j \in Q$ we have that

$$\sum_{j \in Q \setminus \{j'\}} d_j > \frac{1}{k} \sum_{j \in Q \setminus \{j'\}} b(j) > \frac{1}{k} \sum_{j \in Q \setminus \{j'\}} \ell(j') = \frac{1}{k}(|Q| - 1) \cdot \ell(j') \geq \ell(j') .$$

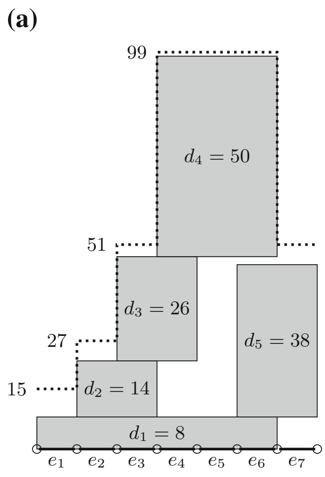
Therefore,

$$\mu_h(Q(e)) = \sum_{j \in Q} d_j = \sum_{j \in Q \setminus \{j'\}} d_j + d_{j'} > \ell(j') + d_{j'} = b(j') .$$

Since $b(j') \geq b(j)$, for every $j \in Q$, we get a contradiction to Observation 2. □

We are now ready to show that there exists a task whose rectangle has at most $2k-2$ neighbors.

Lemma 17 *Let (S, h) be a $\frac{1}{k}$-large* SAP *solution. Then there exists a task $j \in S$ such that* $\deg_S(R(j)) \leq 2k - 2$.



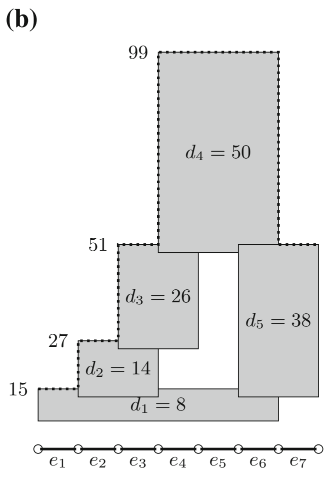


Fig. 8 An example of a SAP solution with five tasks whose corresponding *rectangles* form a cycle. **a** $\frac{1}{2}$-large SAP solution. **b** Corresponding Rectangle Packing instance

Proof Let j_0 be the task with minimal right endpoint, and let e_0 be the right most edge in I_{j_0}. Define

$$Q^- = \{j \in S : b(j) \leq b(j_0)\} \cap N_S(j_0),$$
$$Q^+ = \{j \in S : b(j) \geq b(j_0)\} \cap N_S(j_0) .$$

Observe that $j_0 \in Q^- \cap Q^+$. Consider $j \in Q^-$. Since $R(j) \cap R(j_0) \neq \emptyset$, it follows that $b(j) > \ell(j_0)$. Hence Q^- satisfies the conditions of Lemma 16 with $j' = j_0$ and $e = e_0$, and we have that $|Q^-| \leq k$. Furthermore, observe that $\ell(j) < b(j_0)$, for every $j \in Q^+$, and thus $\cap_{j \in Q^+} R(j) \neq \emptyset$. It follows that $\ell(j') < b(j_0) \leq b(j) \leq b(j')$, for every $j \in Q^+$, for a task j' such that $b(j') = \max_{i \in Q^+} b(i)$. Hence Q^+ satisfies the conditions of Lemma 16 with $e = e_0$. The lemma follows since $\deg_S(R(j)) \leq |Q^-| + |Q^+| - 2 = 2k - 2$. □

We are now ready to prove Theorem 3.

Proof of Theorem 3 Given a SAP solution S, let $G(S) = (S, E)$ be the intersection graph of $\mathcal{R}(S)$, where $E = \{(j, j') : R(j) \cap R(j') \neq \emptyset\}$. Lemma 17 shows that $G(S)$ is $(2k-2)$-degenerate, for any optimal SAP solution S. Therefore $G(S)$ can be colored using $2k - 1$ colors by removing a vertex with minimum degree, recursively coloring the remaining graph, and then coloring the vertex with an available color [27]. The theorem follows due to Theorem 7. □

We note that Lemma 17 is tight for the case of $k = 2$. Figure 8 shows a $\frac{1}{2}$-large SAP solution and the resulting Rectangle Packing instance. Since the instance is a 5-cycle, it is not 2-colorable.

7 SAP on Ring Networks

In this section we present a constant factor approximation algorithm for SAP on ring networks. Our algorithm is based on a simple reduction to the line network that was used by Chakrabarti et al. [14] (and later in [10]) for the UNSPLITTABLE FLOW PROBLEM on rings.

SAP on rings is defined similarly to SAP (on paths). The main difference is that in the former we are given a cycle $C = (V, E)$ and not a path. In this case there are two possible paths from s_j to t_j for each task j, a clockwise path and a counter-clockwise path. A feasible solution for SAP on rings is a triple (S, h, I) where S and h are defined as in SAP, and $I(j)$ is the path chosen for j, for any $j \in S$.

In the next lemma we show that an approximation algorithm for SAP may be used to approximate SAP on rings.

Lemma 18 *A polynomial time α-approximation algorithm for* SAP *implies a polynomial time $(1 + \alpha + \varepsilon)$-approximation algorithm on rings, for any $\varepsilon > 0$.*

Proof The algorithm first chooses any minimum capacity edge e, namely $c(e) = \min_f c_f$. Then, it removes e from the ring and finds a SAP solution (S_1, h_1) on the resulting line network. Observe that the tasks in S_1 are not routed through e. I_1 is defined accordingly. The next step is to compute a solution S_2 by calling an FPTAS for KNAPSACK. Observe that all tasks may be routed through e, either the clockwise path or the counter clockwise path contain e, so all tasks should be considered by the FPTAS. Hence, the input is the (d, w), namely the jth item has size d_j and weight w_j. A height function h_2 is defined as follows: $h_2(j) = \sum_{\ell \in S_2 : \ell < j} d_\ell$, for every $j \in S_2$. We assume that all tasks in S_2 are routed through e, and I_2 is defined accordingly. Finally the algorithm returns the solution with maximum weight.

Clearly, (S_1, h_1, I_1) is feasible. Since e is a minimum capacity edge, (S_2, h_2, I_2) does not violate the capacity of any edge, and therefore it is feasible. It remains to prove that the computed solution is $(1 + \alpha + \varepsilon)$-approximate. We use an argument similar to the one used to prove Lemma 3. Let S^* be the set of tasks in an optimal solution for the original instance. Also, let S_2^* be the tasks in S^* that are routed through e, and let $S_1^* = S^* \setminus S_2^*$. Either $w(S_1^*) \geq \frac{\alpha}{\alpha+1+\varepsilon} \cdot w(S^*)$ and $w(S_1) \geq \frac{1}{\alpha} \cdot \frac{\alpha}{\alpha+1+\varepsilon} \cdot w(S^*) = \frac{1}{\alpha+1+\varepsilon} \cdot w(S^*)$ or $w(S_2^*) \geq \frac{1+\varepsilon}{\alpha+1+\varepsilon} \cdot w(S^*)$ and $w(S_2) \geq \frac{1}{1+\varepsilon} \cdot \frac{1+\varepsilon}{\alpha+1+\varepsilon} \cdot w(S^*) = \frac{1}{\alpha+1+\varepsilon} \cdot w(S^*)$. The lemma follows. □

Theorem 5 follows from Theorem 4 and Lemma 18.

8 Conclusion

We presented a $(9 + \varepsilon)$-approximation algorithm for SAP. Our approximation ratios for medium and large instances match the ratios for UFPP from [10]. In fact our ratio for large tasks is even better (3 instead of 4). However, our approximation ratio for small instances is larger ($4 + \varepsilon$ vs. $1 + \varepsilon$). This larger ratio stem from our need to pack small tasks in strips in order to use the transformation from a UFPP solution to a SAP solution. The ratio for small instances may have been smaller, if we had such a transformation that works on non-uniform instances. It would be interesting to come up with an algorithm for an extended version of DSA in which one is given a path $P = (V, E)$ with a non-uniform capacity vector $c \in \mathbb{R}_+^{|E|}$ and a set of (small) tasks, and the goal is to find the minimum coefficient ρ such that all tasks can be packed within the capacity vector $\rho \cdot c$.

Finally, we note that recently Mömke and Wiese [28] improved our main result by presenting a $(2 + \varepsilon)$-approximation algorithm for SAP.

Acknowledgements We thank an anonymous referee for pointing out that we can use LP-rounding instead of local ratio for computing a $\frac{1}{2}B$-packable SAP solution in Sect. 4.1.

Appendix: A Local Ratio Algorithm for Packing Small Tasks in a Strip

In this section we provide a local ratio algorithm that computes $\frac{1}{2}B$-packable $(5+\varepsilon)$-approximate solutions for δ-small SAP instances in which edge capacities are between B and $2B$, for some constant $\delta > 0$ (depending on ε).

The local ratio technique [5,8] is based on the Local Ratio Lemma. We use the maximization version of the lemma which applies to optimization problems in which the input is a profit vector $w \in (\mathbb{Q}^+)^n$ and a set of feasibility constraints $\mathcal{F}$. The problem is to find a solution vector $x \in \mathbb{Q}^n$ that maximizes the inner product $p \cdot x$ subject to the constraints $\mathcal{F}$.

Lemma 19 (Local Ratio [5]). *Let $\mathcal{F}$ be a set of constraints and let w, w_1, and w_2 be profit functions such that $w = w_1 + w_2$. Then, if x is r-approximate both with respect to $(\mathcal{F}, w_1)$ and with respect to $(\mathcal{F}, w_2)$, for some r, then x is also an r-approximate solution with respect to $(\mathcal{F}, w)$.*

We are now ready to present our local ratio algorithm.

Algorithm 3 : **Strip**(J, w)

```
1: if $J = \emptyset$ then return $\emptyset$
2: Let $j^* \in J$ be a task such that $t^* = \min_{j \in J} t_j$
3: Define $w_1(j) = w(j^*) \cdot \begin{cases} 1 & j = j^*, \\ \frac{2d_j}{B} & j \neq j^*, I_j \cap I_{j^*} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases}$
   and $w_2 = w - w_1$
4: Let $J^+$ be the set of positive weighted tasks
5: $S' \leftarrow$ Strip$(J^+, w_2)$
6: Let $e^*$ be the right-most edge of $j^*$
7: if $d(S'(e^*)) \leq \frac{1}{2}B - d_{j^*}$ then $S \leftarrow S' \cup \{j^*\}$
   else $S \leftarrow S'$
8: Return $S$
```

Sorting the tasks according to their right end-point can be done in $O(n \log n)$. There are $O(n)$ recursive calls, each requiring linear time. Hence the running time of Algorithm **Strip** is $O(n^2)$.

We show that Algorithm **Strip** computes approximate solutions whose load on any edge is at most $\frac{1}{2}B$.

Lemma 20 *Given a δ-small* SAP *instance in which $b(j) \in [B, 2B)$, for every $j \in J$, Algorithm **Strip** computes a $\frac{1}{2}B$-packable* UFPP *solution S. Furthermore, $w(S) \geq \frac{5}{1-4\delta} \cdot \text{OPT}_{\text{SAP}}(J)$.*

Proof We first prove that S is $\frac{1}{2}B$-packable, for every e, by induction on the number of recursive calls. In the base case $S = \emptyset$ and we are done. For the inductive step, assume that $d(S'(e)) \leq \frac{1}{2}B$, for every $e \in E$. First, $d(S(e)) = d(S'(e)) \leq \frac{1}{2}B$, for every $e \notin I_{j^*}$. For $e \in I_{j^*}$, observe that $d(S(e)) \leq d(S(e^*)) \leq \frac{1}{2}B$.

We prove that S is $\frac{5}{1-4\delta}$-approximate also by induction on the number of recursive calls. In the base case $S = \emptyset$ is optimal. For the inductive step, assume that S' is $\frac{5}{1-4\delta}$-approximate with respect to J^+ and w_2. Since $w_2(j^*) = 0$, S is also $\frac{5}{1-4\delta}$-approximate with respect to J and w_2, We show that S is also $\frac{5}{1-4\delta}$-approximate with respect to J and w_1. This completes the proof, since by the Local Ratio Lemma we get that S is $\frac{5}{1-4\delta}$-approximate with respect to J and w as well.

It remains to show that S is $\frac{5}{1-4\delta}$-approximate with respect to J and w_1. Notice that either $j^* \in S$ or $d(S(e^*)) + d_{j^*} > \frac{1}{2}B$. If $j^* \in S$, then $w_1(S) \geq w(j^*)$. Otherwise,

$$w_1(S) > w(j^*) \cdot 2 \cdot \frac{B/2 - d_{j^*}}{B} \geq w(j^*) \cdot 2 \cdot \frac{B/2 - 2\delta B}{B} = w(j^*) \cdot (1 - 4\delta)\ .$$

On the other hand, for a feasible SAP solution T we have that

$$w_1(T) = w_1(T(e^*)) \leq w(j^*) + w(j^*) \cdot 2 \cdot 2B/B = 5w(j^*),$$

due to Observation 1. Therefore $w(S)$ is $\frac{5}{1-4\delta}$-approximate with respect to J and w_1. □

The following lemma replaces Lemma 5.

Lemma 21 *There exists a polynomial time algorithm such that for every constant $\varepsilon > 0$, there exists a constant $\delta > 0$, such that the algorithm computes $\frac{1}{2}B$-packable $(5 + \varepsilon)$-approximate solutions for δ-small* SAP *instances in which $b(j) \in [B, 2B)$, for every $j \in J$.*

Proof First, execute Algorithm **Strip** to compute a $\frac{1}{2}B$-packable $\frac{5}{1-4\delta}$-approximate UFPP solution S. By Lemma 4, S can be transformed into a $\frac{1}{2}B$-packable SAP solution (S', h') such that $w(S') \geq (1 - 4\delta)w(S)$ in polynomial time. It follows that

$$w(S') \geq \frac{(1 - 4\delta)^2}{5} \cdot \text{OPT}_{\text{SAP}}(J) \geq \frac{1 - 8\delta}{5} \cdot \text{OPT}_{\text{SAP}}(J)\ .$$

The lemma follows by setting δ such that $\delta < \delta_0$ and $\frac{5}{1-8\delta} \leq 5 + \varepsilon$. □

References

1. Anagnostopoulos, A., Grandoni, F., Leonardi, S., Wiese, A.: Constant integrality gap LP formulations of unsplittable flow on a path. In: 16th International Integer Programming and Combinatorial Optimization Conference, volume 7801 of LNCS, pp. 25–36 (2013)
2. Anagnostopoulos, A., Grandoni, F., Leonardi, S., Wiese, A.: A mazing $(2 + \varepsilon)$-approximation for unsplittable flow on a path. In: 25th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 26–41 (2014)
3. Bansal, N., Chakrabarti, A., Epstein, A., Schieber, B.: A quasi-ptas for unsplittable flow on line graphs. In: 38th Annual ACM Symposium on the Theory of Computing, pp. 721–729 (2006)
4. Bansal, N., Friggstad, Z., Khandekar, R., Salavatipour, M.R.: A logarithmic approximation for unsplittable flow on line graphs. ACM Trans. Algorithms **10**(1), 1 (2014)
5. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Shieber, B.: A unified approach to approximating resource allocation and scheduling. J. ACM **48**(5), 1069–1090 (2001)
6. Bar-Yehuda, R., Beder, M., Cohen, Y., Rawitz, D.: Resource allocation in bounded degree trees. Algorithmica **54**(1), 89–106 (2009)
7. Bar-Yehuda, R., Beder, M., Rawitz, D.: A constant factor approximation algorithm for the storage allocation problem. In: 25th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 204–213 (2013)
8. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Ann. Discret. Math. **25**, 27–46 (1985)
9. Batra, J., Garg, N., Kumar, A., Mömke, T., Wiese, A.: New approximation schemes for unsplittable flow on a path. In: 26th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 47–58 (2015)
10. Bonsma, P.S., Schulz, J., Wiese, A.: A constant-factor approximation algorithm for unsplittable flow on paths. SIAM J. Comput. **43**(2), 767–799 (2014)
11. Buchsbaum, A.L., Efrat, A., Jain, S., Venkatasubramanian, S., Yi, K.: Restricted strip covering and the sensor cover problem. Technical report. arXiv:cs/0605102v1. CoRR (2008)
12. Buchsbaum, A.L., Karloff, H., Kenyon, C., Reingold, N., Thorup, M.: OPT versus LOAD in dynamic storage allocation. SIAM J. Comput. **33**(3), 632–646 (2004)
13. Calinescu, G., Chakrabarti, A., Karloff, H.J., Rabani, Y.: Improved approximation algorithms for resource allocation. In: 9th International Integer Programming and Combinatorial Optimization Conference, vol. 2337 of LNCS, pp. 401–414 (2002)
14. Chakrabarti, A., Chekuri, C., Gupta, A., Kumar, A.: Approximation algorithms for the unsplittable flow problem. Algorithmica **47**(1), 53–78 (2007)
15. Chekuri, C., Ene, A., Korula, N.: Unsplittable flow in paths and trees and column-restricted packing integer programs. http://www.cs.princeton.edu/aene/papers/ufp-full.pdf
16. Chekuri, C., Ene, A., Korula, N.: Unsplittable flow in paths and trees and column-restricted packing integer programs. In: 12th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, vol. 5687 of LNCS, pp. 42–55 (2009)
17. Chekuri, C., Mydlarz, M., Shepherd, F.B.: Multicommodity demand flow in a tree and packing integer programs. ACM Trans. Algorithms **3**(3), 1–23 (2007)
18. Chen, B., Hassin, R., Tzur, M.: Allocation of bandwidth and storage. IIE Trans. **34**, 501–507 (2002)
19. Chrobak, M., Woeginger, G.J., Makino, K., Xu, H.: Caching is hard—even in the fault model. Algorithmica **63**(4), 781–794 (2012)
20. Erlebach, T., Jansen, K.: The complexity of path coloring and call scheduling. Theor. Comput. Sci. **255**(1–2), 33–50 (2001)
21. Frieze, A.M., Clarke, M.R.B.: Approximation algorithms for the m-dimensional $0-1$ knapsack problem: worst-case and probabilistic analyses. Eur. J. Oper. Res. **15**, 100–109 (1984)
22. Friggstad, Z., Gao, Z.: On linear programming relaxations for unsplittable flow in trees. In: 18th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, vol. 40 of LIPIcs, pp. 265–283 (2015)
23. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
24. Gergov, J.: Algorithms for compile-time memory optimization. In: 10th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 907–908 (1999)
25. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, Cambridge (1980)

26. Leonardi, S., Marchetti-Spaccamela, A., Vitaletti, A.: Approximation algorithms for bandwidth and storage allocation problems under real time constraints. In: 20th Conference on Foundations of Software Technology and Theoretical Computer Science, vol. 1974 of LNCS, pp. 409–420 (2000)
27. Matula, D.W., Beck, L.L.: Smallest-last ordering and clustering and graph coloring algorithms. J. ACM **30**(3), 417–427 (1983)
28. Mömke T., Wiese A., A $(2+\epsilon)$-approximation algorithm for the storage allocation problem. In: 42nd Annual International Colloquium on Automata, Languages, and Programming, vol. 9134 of LNCS, pp. 973–984 (2015)