A Constant Factor Approximation Algorithm for the Storage Allocation Problem

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

Michael Beder*
bederml@cs.technion.ac.il

Dror Rawitz†
rawitz@eng.tau.ac.il

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, namely for 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 show that SAP is strongly NP-hard, even with uniform weights and even if assuming the *no bottleneck assumption*.

Keywords: approximation algorithms, bandwidth allocation, rectangle packing, storage allocation, unsplittable flow.

*Department of Computer Science, Technion, Haifa 32000, Israel.
†School of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel.

1 Introduction

The problems. 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 $S(e) = \{j \in J : e \in I_j\}$ 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 (i) $h(j) + d_j \leq c_e$, for every $e \in E$, and (ii) if $j, i \in S$ such that $I_j \cap I_i \neq \emptyset$ and $h(j) \geq h(i)$, then $h(j) \geq h(i) + d_i$. 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 Figure 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. The resource may be a banner, where each task is an advertisement that requires a contiguous portion of the banner.

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 UFPP instance is called δ-*small* (δ-*large*) if $d_j \leq \delta b(j)$ ($d_j > \delta b(j)$), for every $j \in J$. 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).

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. [6]. As done in [6] 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 from [6] are based on an approximation framework that provides an $(1+\varepsilon)\alpha$-approximation algorithm 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 [4] 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

[1] We note that Bonsma et al. [6] use tiny, medium, and large, since they consider both medium and tiny tasks as small tasks.

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 [4]. Finally a SAP solution is obtained by stacking the strips.

As for large tasks, Bonsma et al. [6] presented an approximation algorithm for large instances of UFPP that is based on (i) a reduction from UFPP to a special 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 BIN PACKING, we show that SAP is strongly NP-hard, even with uniform weights and even under the NBA.

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., [16]). 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 [14].

Bar-Noy et al. [3] 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 [15]. In DSA the goal is to find the minimum capacity c for all edges along with a SAP solution that contains all tasks. An extension of SAP-U in which each task j has a time window was studied in [3, 17]. Calinescu et al. [8] 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 [3].

Chen et al. [12] studied the special case of SAP-U where all demands are multiples of $1/K$, for some integer 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 $d_j = O(1)/K$ for every j. Bar-Yehuda et al. [4] 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. [7]. Bar-Yehuda et al. also presented a dynamic programming algorithm for large instances of SAP-U, and this lead 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. [1] described a deterministic quasi-polynomial time approximation scheme for instances of UFPP, where all capacities and demands are quasi-polynomial, thereby ruling out an APX-hardness result for such instances of UFPP, unless NP $\subseteq$ DTIME$(2^{\text{polylog}(n)})$. Chakrabarti et al. [9] presented a constant factor approximation algorithm for UFPP under the NBA by extending the approach of [8]. Chekuri et

al. [11] used an LP-based deterministic algorithm to obtain a $(2+\varepsilon)$-approximation algorithm for UFPP under the NBA. Bansal et al. [2] developed an $O(\log n)$-approximation algorithm for UFPP, beating the integrality gap of the natural LP-relaxation, which was shown to be $\Omega(n)$ [9]. This result has been generalized to trees and uniform weights by Chekuri et al. [10] who also provided an $O(\log^2 n)$-approximation algorithm for the weighted case. They also developed an LP formulation for UNSPLITTABLE FLOW in trees with general weights with an $O(\log^2 n)$ integrality gap.

Bonsma et al. [6] developed a $(7+\varepsilon)$ approximation algorithm for UFPP, being the first constant-factor approximation algorithm for the problem. They also showed that UFPP is strongly NP-hard even for instances with demands in $\{1,2,3\}$. Chrobak et al. [13] showed that UFPP-U is strongly NP-hard even for the case of uniform weights.

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 Section 3. Our algorithms for medium, small, and large SAP instances are given in Sections 4, 5, and 6, respectively. We show that SAP is strongly NP-hard in Appendix B. We conclude in Section 7.

2 Preliminaries

Given a task set $S \subseteq J$, the demand of S is denoted by $d(S)$, namely $d(S) \stackrel{\triangle}{=} \sum_{j \in S} d_j$. 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 $\mu_h(S(e)) \stackrel{\triangle}{=} \max_{j \in S(e)}(h(j)+d_j)$. 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 $e \in E$.

The following observation bounds the load of a UFPP solution on the edges in term of the maximum bottleneck. A similar observation was made in [6].

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 $\mu_h(S(e)) \leq \max_{j \in S} b(j)$, for every $e \in E$.*

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

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. (A proof is given in Appendix C.)

Lemma 1. *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.*

3 Statement of Results

In this section we provide a formal statement of out 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 $k \geq 1$.*

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

Theorem 4. *There is a polynomial time $(9+\varepsilon)$-approximation algorithm for* SAP.

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 1. □

We also provide a hardness result. The proof is given in Appendix B.

Theorem 5. SAP *is strongly NP-hard, even with uniform weights and even if assuming the NBA.*

4 Medium Tasks

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

4.1 Approximation Framework for SAP

Following [6], 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 [6] 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 3. *$c_e \geq 2^k$, 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 4. *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 $\min(c_e, 2^{k+\ell})$.

Let $q = \log \lceil 1/\beta \rceil$ and let ℓ 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 [6]. 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. *Let $\beta > 0$. A feasible* SAP *solution (S, h) where $S \subseteq J^{k,\ell}$ is called* β-elevated *if $h(j) \geq \beta 2^k$, for every $j \in S$.*

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

Algorithm 1 : **AlmostUniform** (J, ℓ)

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

Since ℓ is a constant there is a linear number of subsets $J^{k,\ell}$. 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 2. *The solution (S_r, h_r) computed by Algorithm* ***AlmostUniform*** *is a feasible* SAP *solution, for every $r \in \{0, \ldots, \ell + q - 1\}$.*

Proof. Given r, let $k_0 = \min K(r)$. Also given $i \in K(r)$, let $i^+ = \min\{k \in K(r) : k > i\}$. For $i \in K(r)$, let $S_i = \bigcup_{k \in K(r), k \leq i} S^{k,\ell}$ and let $h_i = \bigcup_{k \in K(r), k \leq i} h^{k,\ell}$. 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 4 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

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

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 [6] and applies here as well. We provide a proof in Appendix C for completeness.

Lemma 3. *If **Elevator** computes α-approximate solutions, then $w(S_{r^*}) \geq \frac{\ell}{\ell+q} \cdot \frac{1}{\alpha}\text{OPT}_{\text{SAP}}(J)$.*

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

Lemma 4. *Suppose we are given a polynomial time algorithm that computes an α-approximate β-elevated* SAP *solution for $J^{k,\ell}$, for every k and ℓ. Then, if $\ell = \frac{1}{\varepsilon}\log\lceil 1/\beta\rceil$, Algorithm **AlmostUniform** computes a $(1+\varepsilon)\alpha$-approximate solution in polynomial time.*

Proof. We know that the computed solution is feasible due to Lemma 2, and by Lemma 3 we have that

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

as required. □

4.2 Computing β-elevated 2-approximations

In this section we present an algorithm that computes a β-elevated solution for $J^{k,\ell}$, 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 [4] for SAP-U.

Observation 5. *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 given a solution (S,h), apply gravity on the tasks in S, until all tasks cannot fall any further. (See example in Figure 2.)

Using Observation 5 we are able to consider a specific type of optimal solutions.

Lemma 5. *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 $h^*(j) = d_{S^*}(H_j)$, for every task $j \in S^*$.*

Proof. Let (S^*, h^*) be an optimal SAP solution whose existence is implied in Observation 5. 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^*, if follows that each edge $e \in E$ is used by less than $B2^\ell/(\delta B) = 2^\ell/\delta$ tasks. (ii) follows from Observation 5 and (i). □

Lemma 5 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 [4]. (The proof is given in Appendix C.)

Lemma 6. *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.*

Lemma 6 implies that solving SAP on $J^{k,\ell}$ can be done in polynomial time. It remains to obtain a β-elevated solution.

Lemma 7. *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 $c_e \geq 2^k$ due to Observation 3, we have that

$$h(j) + d(j) < \beta 2^k + (1-2\beta)b(j) \leq \beta 2^k + (1-2\beta)c_e = c_e + \beta 2^k - 2\beta c_e \leq c_e - \beta 2^k \,, \qquad (1)$$

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 Figure 3.) (S_1,h_1) is β-elevated due to (1), while (S_2,h_2) is β-elevated by definition. 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 6 and 7.

Lemma 8. *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 6. (S^*,h^*) can be partitioned into two β-elevated solutions (S_1,h_1) and (S_2,h_2) due to Lemma 7. Since $w(S^*) = w(S_1) + w(S_2)$, one of the two solutions is 2-approximate. □

We conclude this section with the proof of Theorem 2.

Proof of Theorem 2. By Lemma 8 there is a polynomial time algorithm that computes β-elevated 2-approximate solutions for $J^{k,\ell}$, for every k and ℓ. Therefore, by Lemma 4, Algorithm **AlmostUniform** is a $(2+\varepsilon)$-approximation algorithm for δ-large and $(1-2\beta)$-small SAP instances. □

5 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 [4]. 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 bottleneck, 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.

5.1 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 $2B$. 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 Appendix D.

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:

$$\begin{aligned} &\max \textstyle\sum_{j\in J} w_j \cdot x_j && \\ &\text{s.t.} \textstyle\sum_{j\in S(e)} d_j x_j \leq c_e && \forall e \in E \\ &x_j \in \{0,1\} && \forall j \in J \end{aligned} \tag{2}$$

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 (2) and define $x' = \frac{1}{4}x^*$. The solution x' satisfies $\sum_{j\in S(e)} d_j x_j \leq \frac{1}{2}B$, and therefore it is feasible with respect to (2) with $c_e = \frac{1}{2}B$, for every e. Since this is a uniform capacity instance we may use the following result of Chekuri, Mydlarz, and Shepherd [11] to obtain an integral solution.

Theorem 6 ([11])**.** *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 $wx \geq \frac{1}{1+\varepsilon} wx^*$ can be found in polynomial time.*

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

Lemma 9 ([4])**.** *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 9 we obtain an approximate SAP solution.

Lemma 10. *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 $(4+\varepsilon)$-approximate solutions for δ-small* SAP *instances in which $b(j) \in [B, 2B)$, for every $j \in J$.*

Proof. Let $\delta_1 = \delta(\frac{\varepsilon}{5})$ be the constant required by Theorem 6. Set $\delta = \delta(\varepsilon)$ such that $\delta < \min\{\delta_1, \delta_0\}$ and $1-4\delta > (4+\frac{4}{5}\varepsilon)/(4+\varepsilon)$. Apply the algorithm from [11] to compute a $\frac{1}{2}B$-packable $4\cdot(1+\frac{\varepsilon}{5})$-approximate UFPP solution S. By Lemma 9, 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

$$w(S') \geq \tfrac{1-4\delta}{4\cdot(1+\varepsilon/5)} \cdot \textsc{opt}_{\textsc{ufpp}}(J) \geq \tfrac{1}{4+\varepsilon} \cdot \textsc{opt}_{\textsc{sap}}(J) \,,$$

as required. □

5.2 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 them combines the solutions. An example of a solution produced by Algorithm **Strip-Pack** is shown in Figure 4.

We conclude the section by showing that Algorithm **Strip-Pack** computes $(4+\varepsilon)$-approximate solutions.

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 Algorithm **Strip-Pack** and the algorithm from Lemma 9, both of which run in polynomial time.

Algorithm 2 : **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)

By Lemma 10 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. □

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. [6] 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 $\ell(j) \stackrel{\triangle}{=} b(j) - d_j$. 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 Figure 5.

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. [6] showed that the set of rectangles $\mathcal{R}(S)$ that correspond to a feasible UFPP schedule S, can be colored using $2k$ colors such that any color induces a pairwise non-intersecting subset of rectangles. Hence the total weight of the tasks that correspond to one of these subsets is at least $\frac{1}{2k} w(S)$. Bonsma et al. 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 ([6])**.** *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 [6] provides a UFPP schedule which 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 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), let $N_S(j) = \{j' \in S \setminus \{j\} : R(j') \cap R(j) \neq \emptyset\}$ and let $\deg_S(R(j))$ be the number of rectangles in $\mathcal{R}(S)$ that intersect $R(j)$, namely $\deg_S(R(j)) = |N_S(j)|$. 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.

Lemma 11. *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, $\sum_{j \in Q} d_j > \ell(j') + d_{j'} = b(j')$, in contradiction to Observation 2 since there exists an edge e such that $e \in I_j$, for every $j \in Q$. □

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

Lemma 12. *Let (S, h) be a $\frac{1}{k}$-large solution. Then there exists a task $j \in S$ such that* $\deg_S(R(j)) \leq 2k-2$.

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

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

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 11 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 11 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. Lemma 12 implies that a coloring using $2k - 1$ colors can be obtained in a greedy manner. The theorem follows due to Theorem 7. □

We note that Lemma 12 is tight for the case of $k = 2$. Figure 6 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 Conclusion

We presented a $(9 + \varepsilon)$-approximation algorithm for SAP. Our approximation ratios for medium and large instances match the ratios for UFPP from [6]. 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. Hence, it would be interesting to come up with algorithms 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 $\rho \cdot c$.

Acknowledgment. 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 Section 5.1.

References

[1] N. Bansal, A. Chakrabarti, A. Epstein, and B. Schieber. A quasi-ptas for unsplittable flow on line graphs. In *38th Annual ACM Symposium on the Theory of Computing*, pages 721–729, 2006.

[2] N. Bansal, Z. Friggstad, R. Khandekar, and M. R. Salavatipour. A logarithmic approximation for unsplittable flow on line graphs. In *20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 702–709, 2009.

[3] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Shieber. A unified approach to approximating resource allocation and schedualing. *Journal of the ACM*, 48(5):1069–1090, 2001.

[4] R. Bar-Yehuda, M. Beder, Y. Cohen, and D. Rawitz. Resource allocation in bounded degree trees. *Algorithmica*, 54(1):89–106, 2009.

[5] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[6] P. Bonsma, J. Schulz, and A. Wiese. A constant factor approximation algorithm for unsplittable flow on paths. In *52nd Annual IEEE Symposium on Foundations of Computer Science*, pages 47–56, 2011.

[7] A. L. Buchsbaum, H. Karloff, C. Kenyon, N. Reingold, and M. Thorup. OPT versus LOAD in dynamic storage allocation. *SIAM Journal on Computing*, 33(3):632–646, 2004.

[8] G. Calinescu, A. Chakrabarti, H. J. Karloff, and Y. Rabani. Improved approximation algorithms for resource allocation. In *9th International Integer Programming and Combinatorial Optimization Conference*, volume 2337 of *LNCS*, pages 401–414, 2002.

[9] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar. Approximation algorithms for the unsplittable flow problem. *Algorithmica*, 47(1):53–78, 2007.

[10] C. Chekuri, A. Ene, and N. Korula. Unsplittable flow in paths and trees and column-restricted packing integer programs. In *12th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, volume 5687 of *LNCS*, pages 42–55, 2009.

[11] C. Chekuri, M. Mydlarz, and F. B. Shepherd. Multicommodity demand flow in a tree and packing integer programs. *ACM Transactions on Algorithms*, 3(3), 2007.

[12] B. Chen, R. Hassin, and M. Tzur. Allocation of bandwidth and storage. *IIE Transactions*, 34:501–507, 2002.

[13] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching is hard - even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

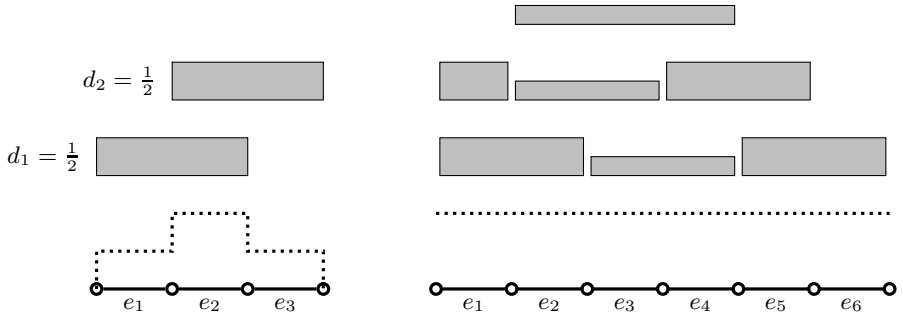
[14] A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional $0-1$ knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

[15] J. Gergov. Algorithms for compile-time memory optimization. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 907–908, 1999.

[16] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

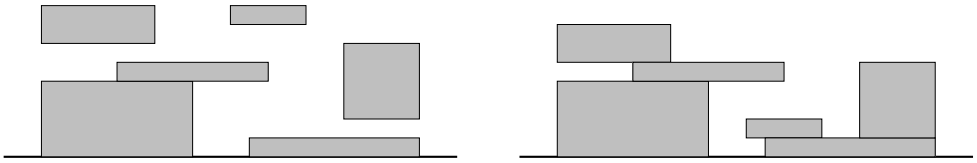
[17] S. Leonardi, A. Marchetti-Spaccamela, and A. Vitaletti. Approximation algorithms for bandwidth and storage allocation problems under real time constraints. In *20th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 409–420, 2000.

A Figures



(a) $c_{e_1} = c_{e_3} = 0.5$ and $c_{e_2} = 1$. (b) $c_e = 1$ for every e.

Figure 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 tasks 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 [12].)



(a) Original solution (b) Solution after application of gravity

Figure 2: Solution (b) is obtained by applying gravity on Solution (a).

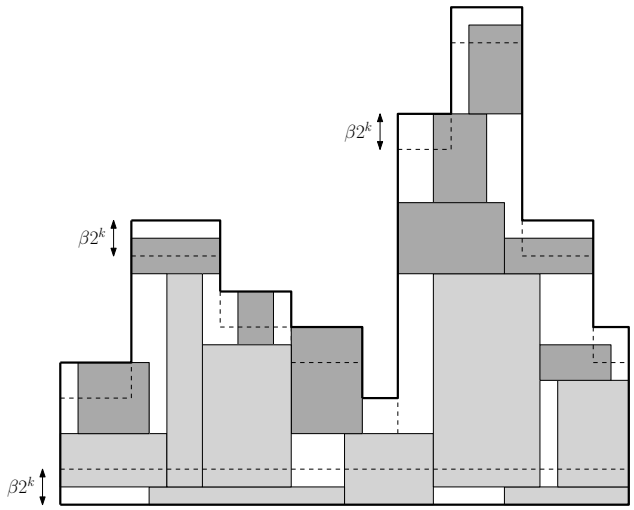


Figure 3: An example of partition of optimal solution into two β-elevated solutions. The light tasks belong to S_1, while the dark tasks belong to S_2.

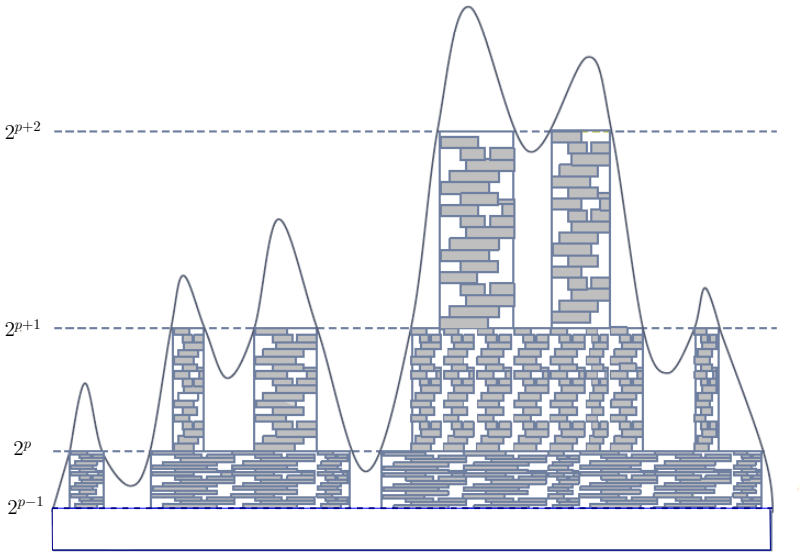


Figure 4: An example of a solution produced by Algorithm **Strip-Pack**.

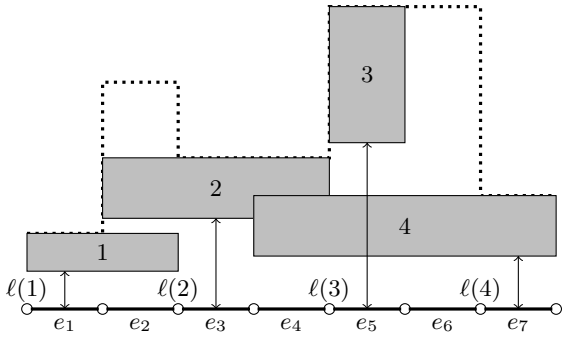
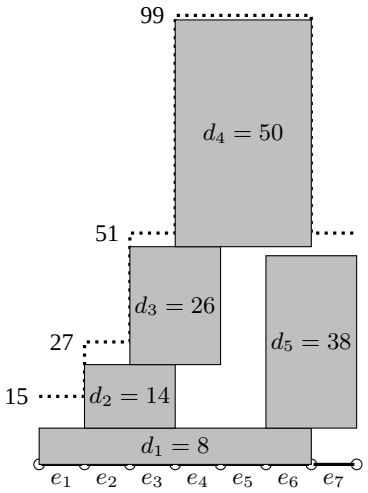
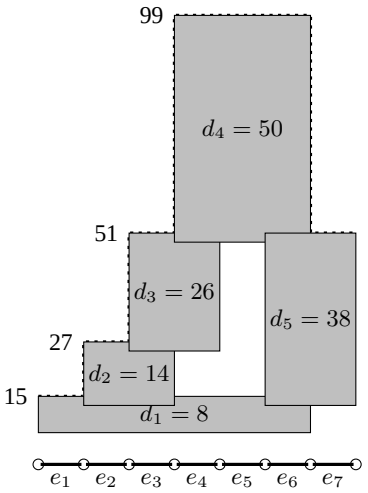


Figure 5: An example of four tasks that are placed at height $\ell(j) = b(j) - d_j$, for every j.



(a) $\frac{1}{2}$-large SAP solution



(b) Corresponding RECTANGLE PACKING instance.

Figure 6: An example of a SAP solution with five tasks whose corresponding rectangles form a cycle.

B Hardness Result

In this section we prove that SAP is strongly NP-hard using a reduction from BIN PACKING.

Given a BIN PACKING instance containing n items of sizes $s_1, \ldots, s_n$, a bin of size 1, and an integer k, we construct the following SAP instance. First the path P contains $2k+1$ edges, with capacities:

$$c_{e_i} = \begin{cases} 2i-1 & i \leq k, \\ 2k-1 & i \in \{k+1, k+2\}, \\ 2(2k+2-i) & i \geq k+3. \end{cases}$$

Also, there are $n+2k-1$ unit weight tasks with demands:

$$d_j = \begin{cases} s_j & j \leq n, \\ 1 & j > n. \end{cases}$$

and intervals:

$$I_j = \begin{cases} \{e_{k+2}\} & j \leq n, \\ \{e_{j-n}, \ldots, e_{k+1}\} & n < j \leq n+k \\ \{e_{k+1}, \ldots, e_{j-n+2}\} & j > n+k \end{cases}$$

The first n tasks represent the items and the remaining $2k-1$ tasks are used to construct a schedule that induces k bins. Such a solution containing tasks $\{n+1, \ldots, n+2k-1\}$ is given in Figure 7.

Lemma 13. *There exists a solution to the* BIN PACKING *instance with k bins if and only if there exists a solution to the* SAP *instance with weight $n+2k-1$.*

Proof. First, observe that since $e_{k+1} \in I_j$ for every $j \in \{n+1, \ldots, n+2k-1\}$, there is only one possible configuration to schedule the tasks $\{n+1, \ldots, n+2k-1\}$ together (see example in Figure 7 for $k=4$). Hence, in a SAP solution with weight $n+2k-1$, the first n tasks are placed in k bins that are formed by the remaining $2k-1$ tasks. The lemma follows. □

Theorem 5 follows from Lemma 13 since BIN PACKING is strongly NP-hard. Note that the reduction constructs instances with uniform weights that satisfy the NBA.

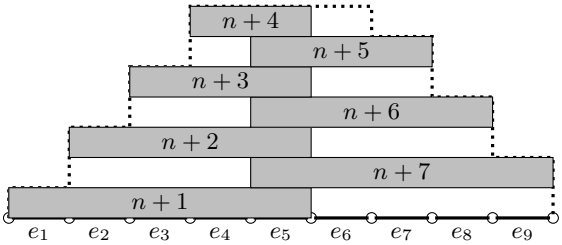


Figure 7: An example of a SAP instance with $k=4$. The dotted line represents the capacity of the edges, and the dark strips correspond to tasks $n+1, \ldots, n+2k-1$. Tasks $1, \ldots, n$ correspond to e_6.

C Omitted Proofs

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

Proof of Lemma 3. 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

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

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

Proof of Lemma 6. 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 $\Pi(e_i, S_i, h_i)$ is the maximum weight of a pair (S', h') such that $S' \subseteq J_i$ 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:

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

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.

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. □

D 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).

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 $w_1(j) = w(j^*) \cdot \begin{cases} 1 & j = j^*, \\ 2d_j/B & j \neq j^*, I_j \cap I_{j^*} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases}$ 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 polynomial.

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

Lemma 14. *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 Theorem [5, 3] 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 10.

Lemma 15. *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. Apply Algorithm **Strip** to compute a $\frac{1}{2}B$-packable $\frac{5}{1-4\delta}$-approximate UFPP solution S. By Lemma 9, 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 $\frac{5}{1-8\delta} \leq 5+\varepsilon$. □