Robustly assigning unstable items

Ananya Christman, Christine Chung, Nicholas Jaczko, Scott Westvold & David S. Yuen

Journal of Combinatorial Optimization

ISSN 1382-6905

J Comb Optim
DOI 10.1007/s10878-019-00515-w

Springer

Journal of Combinatorial Optimization
https://doi.org/10.1007/s10878-019-00515-w

Robustly assigning unstable items

Ananya Christman[1] · **Christine Chung**[2] · **Nicholas Jaczko**[1] · **Scott Westvold**[1] · **David S. Yuen**[3]

Abstract
We study the robust assignment problem where the goal is to assign items of various types to containers without exceeding container capacity. We seek an assignment that uses the fewest number of containers and is robust, that is, if any item of type t_i becomes corrupt causing the containers with type t_i to become unstable, every other item type $t_j \neq t_i$ is still assigned to a stable container. We begin by presenting an optimal polynomial-time algorithm that finds a robust assignment using the minimum number of containers for the case when the containers have infinite capacity. Then we consider the case where all containers have some fixed capacity and give an optimal polynomial-time algorithm for the special case where each type of item has the same size. When the sizes of the item types are nonuniform, we provide a polynomial-time 2-approximation for the problem. We also prove that the approximation ratio of our algorithm is no lower than 1.813. We conclude with an experimental evaluation of our algorithm.

Keywords Approximation algorithm · Robust · Assignment · Hosting · Distributed system · Combinatorial optimization · Bin packing

✉ Ananya Christman
achristman@middlebury.edu

Christine Chung
cchung@conncoll.edu

Nicholas Jaczko
njaczko@middlebury.edu

Scott Westvold
swestvold@middlebury.edu

David S. Yuen
yuen@math.hawaii.edu

[1] Department of Computer Science, Middlebury College, Middlebury, VT, USA

[2] Department of Computer Science, Connecticut College, New London, CT, USA

[3] Department of Mathematics, University of Hawaii, Honolulu, HI, USA

Published online: 06 January 2020

1 Introduction

We study the robust assignment problem (RAP) where we are given various types of items, each with a weight, where items of the same type have the same weight. We must assign the items to a set of containers with the constraint that if an item is found to be corrupt (we assume that there may be at most one such item), then every container containing an item of that type becomes unstable. Therefore, we would like at least one item of every other type to remain in at least one stable container. Such an assignment is considered *robust* and we would like a robust assignment that uses the fewest containers while satisfying their weight limit.

More formally, the input is n item types $t_1, \ldots, t_n$ with sizes (or weights) $w_1, \ldots, w_n$, respectively, and container capacity C. The output is an assignment of types to subsets of containers, which uses the fewest containers and satisfies the following constraints: (1) Each type is assigned to the same number of containers (2) Each container is assigned at most C total weight (3) The assignment is *robust*, that is, for any type t_i, if all containers having an item of type t_i become unstable, for all other types $t_j \neq t_i$, there is a stable container that contains t_j. Formally, let $S_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,k}\}$ for $1 \leq i \leq n$ denote the set of k containers to which an item of type t_i was assigned. Then for every type $t_j \neq t_i$ such that an item of type t_j is also assigned to any container of S_i, an item of type t_j will exist on some container that is not in S_i. The goal is to find a robust assignment that uses the fewest containers.

RAP has many practical applications. For example, in distributed systems, multiple applications, including instances of the same app, are hosted on a cluster of servers. If a failure occurs in an app (and may therefore possibly occur in the other instances of the faulty app), then the app, all of its hosting servers, and hence all other app instances on those servers, are temporarily suspended. Therefore, the system would like an assignment of app instances to the minimal number of servers such that if a failure occurs in an app and therefore all its hosting servers are temporarily suspended, there is still a running instance of every other app hosted on some unaffected server in the system. A solution for RAP can be used to find such an assignment—the apps correspond to the items and the servers correspond to the containers. The goals of our work were in fact motivated by a conversation with industry colleagues who encountered this problem in their company's hosting platforms.

Ad placement on webpages is another application of the Robust Assignment Problem. Ad companies often have ads from multiple clients that must be displayed throughout various webpages of a website. If an ad crashes or slows down, it may affect the entire webpage and hence, the other ads displayed on that webpage as well. Other webpages displaying the faulty ad may need to be temporarily suspended to repair or check the faulty ad. Therefore ad companies would like an assignment of ads to webpages such that if a faulty ad temporarily suspends all of the webpages it is displayed on, there is still a running instance of every other ad on some webpage on the website. Here, the ads and webpages correspond to the items and containers, respectively.

RAP can also be presented as an application to gardening/agriculture. Avid gardeners often grow multiple plants of different varieties in several garden beds. Suppose that during the growing season, it becomes known that a particular plant variety has become

disease prone. Therefore, all plants that are planted in the same bed as a disease-prone plant may become infected with the disease. Therefore, gardeners would like to find a way to plan their garden such that if a plant variety becomes prone to disease, then at least one plant of each variety still grows.

Our Results For RAP we first give an optimal polynomial-time algorithm for finding the minimum number of containers needed to robustly assign the given set of item types, ignoring capacity constraints on the containers (Sect. 3.1). We then introduce the constraint of capacitated containers and give an optimal polynomial-time algorithm for the special case where each type of item has the same size (Sect. 4). For the general case of nonuniform sizes, we provide a polynomial-time 2-approximation for the problem (Sect. 4.2). I.e., our algorithm uses no more than twice the number of containers of the optimal robust assignment. We also prove that the approximation ratio of our algorithm is at least 1.813. We conclude with an experimental evaluation of our algorithm (Sect. 5).

2 Related work

To the best of our knowledge, our specific model for a robust assignment has not been previously studied. However, our solution ideas draw on those used for the bin-packing problem and some assignment problems, so we first discuss literature related to both problems. As mentioned above, in the context of distributed computing, our work applies to the problem of assigning replicas of applications to servers on a hosting platform, so we also discuss some literature on variations of this problem.

Our problem model has similarities to the problem of bin-packing with conflicts (or constraints) (Epstein and Levin 2006; Jansen 1999; Jansen and Öhring 1997). In the most general form of this problem, there are conflicts among the items to be packed and these conflicts are captured by a *conflict graph*, where the nodes represent the items and an edge exists between two items that are in a conflict (Jansen 1999). The goal is to pack the items in the fewest number of bins while satisfying the capacity constraints on the bins and ensuring that no two items in a conflict are packed in the same bin. Jansen proposed an asymptotic FPTAS for this problem for d-inductive graphs (i.e., where the vertices can be assigned distinct numbers $1 \ldots n$ in such a way that each vertex is adjacent to at most d lower numbered vertices) including trees, grid graphs, planar graphs and graphs with constant treewidth (Jansen 1999). For all $\epsilon > 0$, Jansen and Öhring (1997) presented a $(2 + \epsilon)$-approximation algorithm for the problem on cographs and partial K-trees, and a 2-approximation algorithm for bipartite graphs. Epstein and Levin (2006) improved on the 2.7-approximation of Jansen and Öhring (1997) on perfect graphs by presenting a 2.5-approximation. They also presented a 7/3-approximation for a sub-class of perfect graphs and a 1.75-approximation for bipartite graphs.

Our problem differs from these previous problems in at least two important ways. First, the conflicts among our items cannot be easily captured by a conflict graph as they do not pertain to specific pairs of items, but rather to *all* pairs of items. Second,

for our problem, the total number of items that are packed into bins is not predefined, so an algorithm may create more or less if doing so yields fewer bins.

The wide variety of problems that address the task of assigning items to containers while satisfying constraints and minimizing or maximizing some optimization objective are typically classified as Generalized Assignment Problems (Chekuri and Khanna 2000; Shmoys and Tardos 1993). While (to our knowledge) no previous works have considered the requirement of a robust assignment as in our model, a few works have had some similarities to ours. Fleischer et al. (2006) studied a general class of maximizing assignment problems with packing constraints. In particular, they studied the Separable Assignment Problems (SAP), where the input is a set of n bins, a set of m items, values $f_{i,j}$ for assigning item j to bin i; and a separate packing constraint for each bin—i.e., for bin i, a family of subsets of items that fit in bin i; the goal is to find an assignment of items to bins with the maximum aggregate value. For all examples of SAP that admit an approximation scheme for the single-bin problem, they present an LP-based algorithm with approximation ratio $(1 - \frac{1}{e} - \epsilon)$ and a local search algorithm with ratio $(\frac{1}{2} - \epsilon)$. Korupolu et al. (2015) studied the Coupled Placement problem, in which jobs must be assigned to computation and storage nodes with capacity constraints. Each job may prefer some computation-storage node pairs more than others, and may also consume different resources at different nodes. The goal is to find an assignment of jobs to computation nodes and storage nodes that minimizes placement cost and incurs a minimum blowup in the capacity of the individual nodes. The authors present a 3-approximation algorithm for the problem.

One application of our work is the problem of assigning replicas of applications to servers on a hosting platform so that the system is fault-tolerant to a single application failure. There have been a wide variety of studies on related problems and here we discuss a few. Rahman et al. (2008) considered the related Replica Placement Problem where copies of data are stored in different locations on the grid such that if one instance at one location becomes unavailable due to failure, the data can be quickly recovered. They present extensive experimental results for this problem. Mills et al. (2017) also studied a variation of this problem in the setting where dependencies exist among the failures and the general goal is to find a placement of instances that does not induce a large number of failures. They give two exact algorithms for dependency models represented by trees. Urgaonkar et al. (2007) also studied the problem of placing apps on servers, but do not consider fault tolerance and focus instead on satisfying each application's resource requirement. The authors study the usefulness of traditional bin-packing heuristics such as First-Fit and present several approximation algorithms for variations of the problem.

More recently, Korupolu and Rajaraman (2016) studied the problem of placing tasks of a parallelizable job on servers with the goal of increasing availability under two models of failures: adversarial and probabilistic. In the adversarial model, each server has a weight and the adversary can remove any subset of servers of total weight at most a given bound; the goal is to find a placement that incurs the least disruption against such an adversary. For this problem they present a PTAS. In the probabilistic model, each node has a probability of failure and the goal is to find a placement that maximizes the probability that at least a certain minimum number of tasks survive at

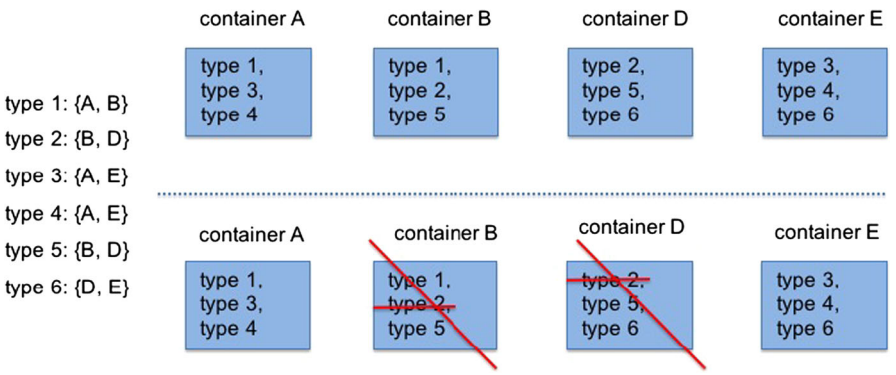

Fig. 1 Example of a non-robust assignment. If type 2 is corrupt, no items of type 5 exist

any time. For the most basic version of the problem they study they give an algorithm that achieves an additive ϵ-approximation. Stein and Zhong (2018) studied a related problem of processing jobs on machines to minimize makespan. The jobs must be grouped into sets before the number of machines is known and these sets must then be scheduled on machines without being separated. They present an algorithm that is guaranteed to return a schedule on any number of machines that is within a factor of $(\frac{5}{3} + \epsilon)$ of the optimal schedule, where the optimum is not subject to the restriction that the sets cannot be separated.

3 Preliminaries

We begin by presenting a small example instance of RAP to provide concreteness. Figures 1 and 2 show two assignments and the corresponding states of the containers for $n = 6$ item types. In Fig. 1, the assignment is not robust—if type 2 is found to be corrupt, then no items of type 5 will exist in any other containers since all items of type 5 are in the same set of containers as items of type 2 (a similar problem occurs with types 3 and 4). Figure 2 depicts one robust assignment using the optimal number of containers: 4. Note that if an item of type 2 fails, then all item types contained in B and D exist in some other container. Further note that in this assignment if *any* of the item types are found to be corrupt, this robustness property holds.

We note that the remainder of this section applies to the more general form of the problem where each item type may be assigned to a different number of containers.

A *robust* assignment is characterized by whether each type is assigned to a set of containers that is not a subset of the set of containers assigned to any other type. We present this characterization formally as our first Lemma.

Lemma 1 *Let $S_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,k_i}\}$ for $1 \leq i \leq n$ denote the set of k_i containers to which an item of type t_i was assigned. An assignment of item types to containers is robust if and only if there is no pair of item types t_i, t_j such that $S_i \subseteq S_j$.*[1]

[1] Note that $S_i \subseteq S_j$ is the condition for the general case; for the case where $k_i = k_j$ for all $i, j \in \{1, 2, \ldots, n\}$, the condition is $S_i = S_j$.

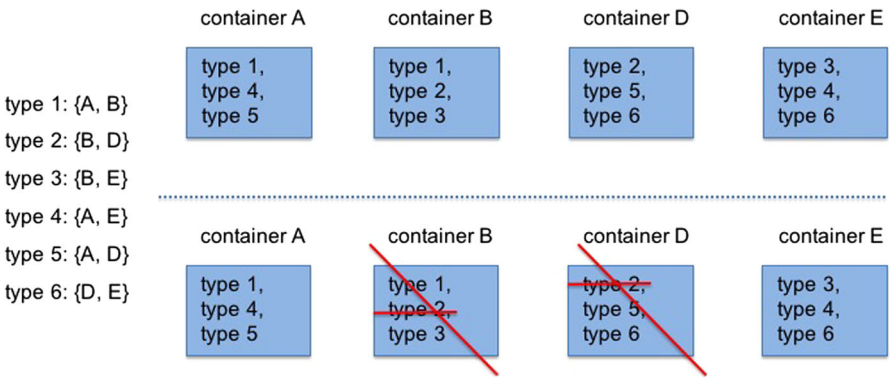

Fig. 2 An optimal robust assignment. If any type is corrupt, all other types still exist

Proof First we show that a robust assignment implies no pair of types t_i, t_j will be such that $S_i \subseteq S_j$. Suppose by way of contradiction that there is some pair of types t_i, t_j where $S_i \subseteq S_j$. This means if type t_j is found to be corrupt, and all the containers in S_j become unstable, then all the containers in S_i also become unstable. In this case there are no items of type t_i in stable containers so the assignment was not robust. We now prove the other direction of the lemma. Suppose for contradiction we have no pair of types t_i, t_j such that $S_i \subseteq S_j$, but the assignment is not robust. If it is not robust, there is some type t_h such that removing the containers in S_h will leave another type $S_{h'}$ in no remaining containers. But for this to be true, it must be that $S_{h'} \subseteq S_h$ which is a contradiction. □

3.1 Uncapacitated robust assignment problem

In this section we begin by considering the special case of the RAP where the containers have infinite capacity. To tackle the Uncapacitated robust assignment problem we first consider the inverse problem: given m containers, what is the maximum number of item types we can assign robustly? Due to Lemma 1 this problem can be modeled as the combinatorics problem of finding the maximum cardinality *antichain* of a set. Specifically, let P denote the set of subsets of m elements $\{1, 2, \ldots, m\}$. An antichain of P is a set $\bar{P} = \{s_1, s_2, \ldots, s_a\} \subseteq P$ such that for any pair of subsets s_i, s_j in $\bar{P}$, $s_i \nsubseteq s_j$. Table 1 shows all antichains for $m = 1, 2$, and 3.

3.2 Antichains

Table 1 shows all antichains for $m = 1, 2$, and 3.

Sperner's Theorem [1928] states that the maximum cardinality of an antichain $\bar{P}$ of an m-sized set is $\binom{m}{\lfloor m/2 \rfloor}$ and each subset of $\bar{P}$ has size $m/2$. (If m is odd then there will be two maximum cardinality antichains whose subsets will have size $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$, respectively.) For example, the last two rows of the rightmost column in Table 1 show the maximum cardinality antichains for $m = 3$ (both have cardinality

Table 1 Antichains for sets of subsets of m elements

m	P	Antichains
1	{∅, {1}}	{∅}, {{1}}
2	{∅, {1}, {2}, {1, 2}}	{∅}, {{1}}, {{2}}, {{1}, {2}}, {{1, 2}}
3	{∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}	{∅}, {{1}}, {{2}}, {{3}}, {{1, 2}}, {{1, 3}}, {{2, 3}}, {{1, 2, 3}}, {{1}, {2}}, {{1}, {3}}, {{2}, {3}}, {{1}, {2, 3}}, {{2}, {1, 3}}, {{3}, {1, 2}}, {{1, 2}, {1, 3}}, {{1, 2}, {2, 3}}, {{1, 3}, {2, 3}}, {{1}, {2}, {3}}, {{1, 2}, {1, 3}, {2, 3}}

Table 2 The maximum number of types that can be robustly assigned to $1 \leq m \leq 10$ containers

m (# of containers)	Maximum number of item types that can be robustly assigned
1	1
2	2
3	3
4	6
5	10
6	20
7	35
8	70
9	126
10	252

3). Therefore, Sperner's Theorem yields the maximum number of item types that can be assigned to m containers as well as the number of containers to which each type is assigned. The values in Table 2 were derived from Sperner's Theorem.

We can thus use this theorem in conjunction with Lemma 1 to solve our original assignment problem—that is, given n types, find the minimum number of containers required to assign these types. Specifically, given n types, we would like to find the smallest m such that $\binom{m}{\lfloor m/2 \rfloor} \geq n$. See Algorithm 4 for further details.

Theorem 1 *The uncapacitated robust assignment problem is solvable in time polynomial in n, the number of item types.*

Proof Due to Sperner's Theorem and Lemma 1, Algorithm 1 correctly returns the minimum number of containers required. Steps 1 and 2 take no more than time linear in the number of types as n serves as a trivial upperbound on the value of m that satisfies the Sperner's Theorem condition. Steps 3 and 4 of the algorithm require enumeration

Algorithm 1: Input is n item types.

1: Apply Sperner's Theorem: find the minimum integer m such that $\binom{m}{\lfloor m/2 \rfloor} \geq n$.
2: Set up m empty containers.
3: Generate all $\binom{m}{\lfloor m/2 \rfloor}$ of the $\lfloor m/2 \rfloor$-combinations of the m containers.
4: Assign each item type one of the $\lfloor m/2 \rfloor$-combinations, i.e., for each type, assign an item of that type to each of the $\lfloor m/2 \rfloor$ containers in the $\lfloor m/2 \rfloor$-combination that this type was assigned to.

of the $\binom{m}{\lfloor m/2 \rfloor}$ combinations; the number of combinations is exponential in m, but since the chosen m will be $O(\log(n))$, the composite run time is still polynomial in n. □

Note, we can potentially find the solution more quickly by computing upper and lower bounds on m using Stirling's approximation which states that $\binom{m}{\lfloor m/2 \rfloor} \approx m + \frac{1}{2} - \frac{1}{2}\log_2(m\pi)$ (Stirling 1730).

4 The robust assignment problem with capacity constraints

In Sect. 3.1, we implicitly assumed that an unlimited number of items can be assigned to any one container. However, in practical settings, constraints such as storage space, memory, or other demands will impose limits on the number of items a container may hold. We therefore consider a model where there is one such constraint. We will use the example of a storage constraint for expository purposes.

The problem now becomes: given n types $t_1, t_2, \ldots, t_n$ with integer-valued *sizes*, $w_1, w_2, \ldots, w_n$, respectively, where items of type t_i have size w_i; and an integer-valued container capacity C, where $1 \leq w_i \leq C$, find an assignment of items to containers such that each type is assigned the same number of containers, k, and the assignment is (1) robust, (2) minimizes the total number of containers, and (3) satisfies the following *capacity constraint*: if A_j is the set of items assigned to container s_j, then for all containers $j = 1 \ldots m$, where m is the number of containers used in the assignment, $\sum_{a \in A_j} w(a) \leq C$, where $w(a)$ is the size of item a. We refer to this variant as the *capacitated robust assignment problem*.

As a small example, suppose in Fig. 2, types $1, 2, \ldots, 6$ have sizes $1, 2, \ldots 6$, respectively. Then if $C = 12$, the assignment shown in the figure would not satisfy the capacity constraint since both containers D and E currently use 13 units of size. Figure 3 shows an altered assignment that satisfies both the robustness and capacity constraints.

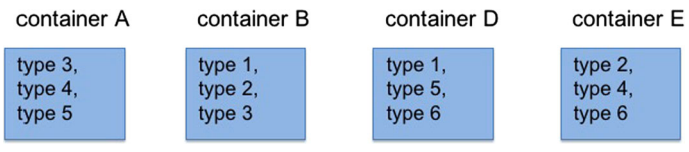

Fig. 3 An assignment that satisfies both the robustness and capacity constraints for capacity equal to 12

4.1 Uniform sizes

We first consider the special case where each type, and therefore each item, has the same size w. Given n such types and containers of capacity $C \geq w$, the problem is to find an assignment of types to the minimal number of containers such that the assignment is robust and also satisfies the capacity constraints. In this case the capacity constraint is that if $|A_j|$ denotes the number of items assigned to container s_j, and m is the number of containers in the assignment, then for all $j = 1 \ldots m$, $|A_j|w \leq C$.

Theorem 2 *The capacitated robust assignment problem with uniform sizes is solvable in time polynomial in n, the number of item types.*

Proof Algorithm 2 solves this problem optimally. Recall that k denotes the number of items of each type. The algorithm effectively performs an exhaustive search to find the minimum m over all possible k for which the robustness and capacity constraints are satisfied. Specifically, the algorithm starts with the lower bound for m (given by Sperner's Theorem) and searches every possible integral value of k given this m (i.e., starting from $k = \lfloor m/2 \rfloor$ down to $k = 1$) that will satisfy both the robustness and capacity constraints. Robustness is satisfied if $\binom{m}{k} \geq n$ and the capacity constraint is satisfied if $\lfloor \frac{C}{w} \rfloor \geq \frac{nk}{m}$. (Refer to Algorithm 3 and Lemmas 2 and 3 for why this capacity condition suffices.) If no value for k for the given m satisfies both constraints, the algorithm increments m and repeats the search for k.

The details for assigning the n types robustly to the m containers computed by Algorithm 2 are described in Sect. 4.1.1. We also note that while there are ways to optimize the run-time of our algorithm, the exhaustive-search version we present here is for the sake of simplicity and clarity.

Note that the algorithm will eventually terminate: if eventually m is incremented to n and k is decremented to 1 both conditions of the while loop will be true. There will be $O(n^2)$ iterations of the while loop. Each iteration takes constant time so the runtime of the loop is $O(n^2)$. The polynomial run-time and correctness of step 11 (Algorithm 3) is addressed in Lemmas 3 and 4in Sect. 4.1.1 So the overall run time of Algorithm 2 is polynomial in n. □

We note that if the problem is simply to find the minimum number of containers needed for the robust assignment, without also requiring the robust assignment itself, one can do so in polylog(n) time by formulating the problem as a fixed-dimension integer program. Namely, given inputs (n, C) where for simplicity we assume $w = 1$, then we want to solve the system $1 \leq k \leq m/2$, $kn \leq mC$, $\binom{m}{k} \geq n$ for k and m with m minimal. The key observation is that for any fixed m, the k satisfying the first two equations that yield the largest $\binom{m}{k}$ is $k = \lfloor \min(m/2, mC/n) \rfloor$. Thus whether or not an m has a corresponding k that satisfies the three equations is equivalent to whether or not $\binom{m}{\lfloor \min(m/2, mC/n) \rfloor} \geq n$. Because we can show $\binom{m}{\lfloor \min(m/2, mC/n) \rfloor}$ is an increasing function in m, then we can do a binary search for the minimal m that satisfies $\binom{m}{\lfloor \min(m/2, mC/n) \rfloor} \geq n$ in the interval $[1, n]$. This would be far more efficient than the brute force while-loop that we give in Algorithm 2.

Algorithm 2: Input is the container capacity C, n item types, and item size $w \leq C$.

1: Apply Sperner's Theorem: find the minimum integer m such that $\binom{m}{\lfloor m/2 \rfloor} \geq n$.
Note that m is a lower bound on the number of containers required to assign the types.
2: $k = \lfloor \frac{m}{2} \rfloor$
3: **while** not ($\binom{m}{k} \geq n$ and $\lfloor \frac{C}{w} \rfloor \geq \frac{nk}{m}$) **do**
4: **if** $k > 1$ **then**
5: $k - -$ //decrease the number of items per type
6: **else**
7: $m + +$ //add another container
8: $k = \lfloor \frac{m}{2} \rfloor$ //re-initialize k for the new m
9: **end if**
10: **end while**
11: For details on how to assign k items of each type so that the robustness and capacity constraints are satisfied using m containers of capacity C, refer to Algorithm 3 in Sect. 4.1.1.

4.1.1 Details for the case of uniform sizes

To assign the types to containers in the case of uniform sizes, we can use the subprocedure detailed in Algorithm 3. We begin by invoking Algorithm 2 from Sect. 4.1 to compute optimal values for m, the number of containers we will use, and k, the number of items per type.

In the algorithm, we let $M_0, M_1, \ldots, M_{m-1}$ refer to the m containers, and we use the term *contiguous set* to refer to a subset of k of these containers consecutively, allowing "wrap-around." So, a *contiguous set* of containers is a set $M_{i \bmod m}, M_{i+1 \bmod m}, M_{i+2 \bmod m}, \ldots, M_{i+k-1 \bmod m}$ for any integer $i \geq 1$.

We define the *translate*, $T_w(X)$, of a subset of containers

$$X = \{M_{x_1}, M_{x_2}, \ldots M_{x_k}\}$$

to be

$$T_w(X) = \{M_{(x_1+w) \bmod m}, M_{(x_2+w) \bmod m}, \ldots, M_{(x_k+w) \bmod m}\}.$$

Lemma 2 *In each iteration of the while loop in Algorithm 3, each container is used an equal number of times. I.e., in any iteration i of the while loop, if container j is assigned n_i types during iteration i, then all m containers are assigned n_i types in iteration i.*

Proof It is clear that if $T(S)$ has no repeated sets, then by the symmetry of the translates in $T(S)$, each container must appear an equal number of times among the sets of $T(S)$. Now we consider the case where $T(S)$ has repeated sets that are then removed in line 7. This case is handled by the following observation. Let $T(Y) = \{T_0(Y), T_1(Y), \ldots T_{m-1}(Y)\}$ be the set of all translates of any subset of containers, Y. Again, by the symmetry of the translates in $T(Y)$, if any set in $T(Y)$ is repeated r times, then all the sets are repeated r times. Hence the sets remaining in $T(S)$ on line 8 maintain the property that each container occurs an equal number of

Algorithm 3: Input is the container capacity C, n item types, and item size $w \leq C$.

1: Use Algorithm 2 from Sect. 4.1 to compute m, the minimum number of containers needed, along with k, the corresponding number of items per type for the given C and w.
2: Let the m containers be called $M_0, M_1, \ldots, M_{m-1}$.
3: Let $i = 0$
4: **while** $n - i > m$ **do**
5: Let S be a non-contiguous set of k containers that has not already been assigned to a type (in line 9 below).
6: Let $T(S) = \{T_0(S), T_1(S), T_2(S), \ldots, T_{m-1}(S)\}$ be the set of all m "translates" of S.
7: Ensure the sets in $T(S)$ are distinct by removing any repeated sets in $T(S)$.
8: **for** each set S' in $T(S)$ **do**
9: Assign one item of type t_i to each of the k containers in S'
10: i++
11: **end for**
12: **end while**
{now at most m types remain to be assigned}
13: $j' = 0$
14: $j = j'$
15: **for** each remaining type t_i still unassigned **do**
16: Let $C_j = \{M_{j \bmod m}, M_{j+1 \bmod m}, \ldots, M_{j+k-1 \bmod m}\}$
17: Assign one item of type t_i to each of the k containers in C_j
18: $j = j + k$
19: **if** $j \geq m$ **then**
20: j'++
21: $j = j'$
22: **end if**
23: **end for**

times. Specifically, this number will be between 1 and k since the at most m translates each contain k containers. □

The idea of the following proof of correctness of Algorithm 3 is that due to Lemma 2, the algorithm is always assigning types to containers in a completely load-balanced way, which maintains the availability of all $\binom{m}{k} \geq n$ combinations of the containers.

Lemma 3 *Algorithm 3 robustly assigns all n types to the m containers given by Algorithm 2.*

Proof To achieve a robust assignment the algorithm must assign each type to a distinct k-combination of containers without exceeding any container's capacity. Note that by Algorithm 2, $\binom{m}{k} \geq n$ so S (in Line 5) will always be non-empty and therefore the while loop will terminate with $i \geq n - m$. It is clear that all assignments made in the while loop are made to distinct k-combinations as S is chosen to be a k-combination that has not yet been assigned, and any sets in $T(S)$ that are repeated are removed before making assignments. The assignments made in the for loop that follows on line 15 are also distinct k-combinations from those already used because these are all contiguous sets of containers, while the while loop exclusively used non-contiguous k-combinations. What remains to show is that no container capacity is ever exceeded.

Assume for contradiction that the algorithm at some point assigns a type to a container M that is already at capacity (has been assigned $\lfloor C/w \rfloor$ types), thereby exceeding its capacity. Consider the first moment this occurs. We will reach a contradiction in

both of two separate cases based on whether this moment occurs during the while loop or during the later for loop on line 15.

Case 1 The container M is filled to capacity during iteration i of the while loop and later, during iteration $j \geq i$ of the while loop, the algorithm attempts to assign another type to M, exceeding its capacity. By Lemma 2, after each iteration of the while loop, all the containers must be equally loaded. Hence if M was filled to capacity in iteration i this must mean all containers were filled to capacity in that iteration. And since we have $nk \leq \lfloor C/w \rfloor m$, by the condition of the while loop in Algorithm 2, filling $\lfloor C/w \rfloor m$ space implies that all k items of each of the n types have already been assigned by that iteration. Contradiction.

Case 2 The container M that is already filled to capacity is assigned another type (therefore exceeding its capacity) during the for loop on line 15. Note that similar to the while loop, the for loop has the property of using the containers in a load balanced way, by assigning the remaining (at most m) types to "non-overlapping" contiguous subsets of the m containers. Note that there are m such distinct contiguous subsets that have not already been assigned so we are guaranteed to have enough for the remaining types. Further note that looping through them k at a time, "end-to-end," as the algorithm does, load-balances them while ensuring that none are used twice.

Since we have already established that the while loop always fills the containers in a perfectly load-balanced way, we can conclude that at any moment during the for loop on line 15, the loads of the containers are within one type of one another. That is, if a least-loaded container has been assigned s types, then each of the m containers has been assigned either s types or $s+1$ types. In this case, the container M that is at capacity is being assigned its $s+1$th type (in excess of its capacity), call it type t'. Let x be the number of containers that have $s+1$ types after type t' is assigned. Note $x \geq 1$ since M has $s+1$ types.

This implies $nk \geq s(m-x)+(s+1)x = ms+x$, which means $nk > ms$. However, $nk \leq \lfloor C/w \rfloor m$, which means $\lfloor C/w \rfloor \geq s+1$. Contradiction, with the assumption that the capacity of M is exceeded. □

Lemma 4 *Algorithm 3 is a polynomial time procedure.*

Proof The while loop on line 4 iterates $n-m$ times. Line 5 can be executed in time $O(n^2k)$ since we can choose potential sets for S in lexicographical order, and for each candidate, (at worst) loop through a list of k-combinations that we have already used to see if it is present. The remainder of the while loop body can clearly be executed in polynomial time. And the for loop on line 15 iterates at most m times, so the total run time of Algorithm 3 is at most polynomial. □

4.2 Nonuniform sizes

In this section we consider the variant of the problem where there may be a different size $1 \leq w_i \leq C$ for each type t_i. In this case, if we ignore the robustness constraint, the problem would be NP-hard due to its equivalence to bin-packing (Garey and Johnson 1979). We first present our algorithm, Robust First Fit (RFF), for this problem and

prove that its approximation ratio is at most 2. We then show that the approximation ratio of RFF is not lower than 1.813.

4.2.1 The robust first fit algorithm

RFF begins by sorting the types in descending order by size. When finding an assignment for type t_i the algorithm first finds the set S of all the containers that have enough empty space to fit an item of type t_i. It then assigns an item of type t_i to the (lexicographically) first container assignment that can be created from the containers in S that has not already been used by a previous type.

Whenever no suitable assignment can be created for type t_i with the existing containers, a new empty container is created. An assignment with a storage constraint will never require fewer containers than an assignment without a storage constraint, so the number of containers (m) and number of items of each type (k) are initialized to the values given by Sperner's Theorem. The **for**-loop in step 3 accounts for the fact that decreasing the number of items of each type might decrease the number of containers required. Therefore, we start with $k = \lfloor m/2 \rfloor$, as given by Sperner's Theorem, and try decreasing the number of items from there.

RFF runs in polynomial time. Since the initial value of m from step 2 can be at most n, the **for**-loop in Step 3 will run for at most $O(n)$ iterations. Each iteration of the loop finds an assignment for n types. To find an assignment for each type t_i, the algorithm searches for a k-combination that is *available* (i.e., has not already been assigned) whose containers have sufficient space for an item of type t_i. This can be done in poly-time as there will never be more than $O(n)$ k-combinations to check before finding one that is available. Since the number of containers will be no more than $n \cdot \frac{n}{2} = O(n^2)$ (i.e., if one item of each type was assigned to its own dedicated container), the algorithm may reach Step 10 (which causes a new iteration from Step 7) $O(n^2)$ times. Hence the overall run-time of RFF is polynomial.

We note that there are clearly ways to optimize the run-time of our algorithm if one wishes to implement it on a real-world system (for example, using binary search instead of linear search). The version we present here is for the sake of simplicity and clarity.

4.2.2 Upper bound

We now show that RFF has an approximation ratio of no worse than 2.

Theorem 3 RFF *is a 2-approximation for the capacitated robust assignment problem with nonuniform sizes. I.e.,* RFF *will use at most* $2m^*$ *containers to robustly assign all* n *types, where* m^* *is the number of containers that an optimal solution uses.*

Proof Consider any input instance. Let n denote the number of item types to be assigned, let OPT be an optimal robust assignment for them, and let k^* be the number of items assigned per type in OPT. It suffices to show that RFF uses at most $2m^*$ number of containers in iteration $k = k^*$ since RFF ultimately chooses the value of k that minimizes the number of containers required. Hence, if RFF uses no more than

Algorithm 4: ROBUST FIRST FIT (RFF). Input is the container capacity C and a set T of n types where all items of type t_i have size $w_i \leq C$ for $1 \leq i \leq n$.

1: Sort the types in descending size order.
2: Apply Sperner's Theorem: find the minimum integer m such that $\binom{m}{\lfloor m/2 \rfloor} \geq n$.
Note that m is a lower bound on the number of containers required to hold the items robustly.
3: **for** $k = \lfloor \frac{m}{2} \rfloor$ to 1 **do**
4: Set up $m' = m$ empty containers.
5: **for** each type t_i in T **do**
6: Let S denote the subset of the m' containers that still have sufficient space to fit an item of type t_i
7: **while** t_i not assigned **do**
8: **if** a k-combo of S is still available for assignment **then**
9: Let S' be the lexicographically-first such available k-combo of S
10: Assign t_i to S' (i.e., assign one item of type t_i to each of the k containers in S').
11: Mark/note that the k-combo S' is no longer available.
12: **else**
13: Add a new container to S, and increment m'. (Note that k does not change.)
14: **end if**
15: **end while**
16: **end for**
17: Store m' along with the corresponding assignment.
18: **end for**
19: Return the assignment among those stored in Step 20 that used the fewest containers (i.e., had the smallest value of m').

$2m^*$ containers when it assigns k^* items per type, it must ultimately not use more than $2m^*$ containers. We also may assume $k^* \geq 2$, since in the case of $k^* = 1$ RFF and OPT will both use a dedicated assignment (one item of each type per container) so RFF will return an optimal assignment, using $m = m^*$ containers.

Suppose for contradiction that OPT uses m^* containers while RFF uses strictly more than $2m^*$ containers when assigning the n types. Consider the moment during the execution of the RFF algorithm that container number $2m^* + 1$ was opened and added to S. Let t_i be the type that was being assigned when RFF opened this $(2m^* + 1)$th container. Let w_i be the size of type t_i. Let $S_{<i}$ be the set of $2m^*$ containers already in use by the algorithm when it tried to assign t_i, but before it added container number $2m^* + 1$. Note that RFF has sorted and re-indexed the types in descending size order.

Case 1 $2 \leq k^* \leq \lfloor m^*/2 \rfloor$, $w_i > C/3$. Let B denote the set of all types t_j for whom $w_j > C/3$. Note that type $t_i \in B$. Due to their size, no more than two items of types in B can fit on a single container, so there can be no more than $2m^*$ such items in total, i.e., $k^*|B| \leq 2m^*$. Note however, that RFF must be able to assign the $k^*|B| \leq 2m^*$ items to at most $2m^*$ containers because $2m^*$ containers would indeed be sufficient for even a dedicated assignment: one item of each type per container. This contradicts the assumption that type t_i required RFF to open a $(2m^* + 1)$th container.

Case 2 $2 \leq k^* \leq \lfloor m^*/2 \rfloor$, $w_i \leq C/3$. In this case, we consider two sub-cases. Subcase 1: there are at least m^* containers in $S_{<i}$ with available space at least w_i (i.e., enough space for an item of type t_i). In this case, we would then have a robust assignment from the set $S_{<i}$ for t_i because OPT needed only m^* containers total to assign all n

types, so having m^* containers must provide enough k^*-combinations to have at least one left for t_i.

Subcase 2 there are fewer than m^* containers in $S_{<i}$ with available space at least w_i. So, in this case there must be $m^* + x$ containers $\bar{S} \subseteq S_{<i}$, where $x > 0$, that have less than w_i available space. We can say that each of these containers in $\bar{S}$ already has filled capacity $\bar{C} > C - w_i$. So if $w(\bar{S})$ is the total size of all of the items in the containers in $\bar{S}$, then $w(\bar{S}) > (m^* + x)(C - w_i)$. Since OPT used m^* containers of capacity C to assign all k items of each of the n types robustly, we have $(m^* + x)(C - w_i) < m^*C$. Recalling that we are in the case where $w_i \leq C/3$, we then have

$$(m^* + x)\left(C - \frac{C}{3}\right) = (m^* + x)\left(\frac{2C}{3}\right) < m^*C.$$

This implies $2xC/3 < m^*C/3$, which implies

$$x < m^*/2. \tag{1}$$

Let $\hat{S} = S_{<i} - \bar{S}$ be the set of containers in $S_{<i}$ that still have enough remaining capacity to store an item of type t_i. For type t_i to be unable to be assigned to these $|\hat{S}| = |S_{<i}| - |\bar{S}| = m^* - x$ containers, it must be due to robustness: they must have no remaining available unique combinations of containers. We will show however, that if this were true, it would also lead to a contradiction.

If there are no unique combinations of containers remaining in $\hat{S}$ to assign t_i to, there must be at least $\binom{m^*-x}{k^*}$ distinct types that are already assigned to those containers. In other words, if

$$\hat{T} = \{t_j \in T : \text{ an item of type } t_j \text{ is assigned to some container in } \hat{S}\},$$

then $|\hat{T}| \geq \binom{m^*-x}{k^*}$. This is true because if $|\hat{T}| < \binom{m^*-x}{k^*}$ then there would be at least one remaining available k^*-combination of the containers in $\hat{S}$ on which to assign t_i.

RFF considers types in descending order by size so each item of the $|\hat{T}| \geq \binom{m^*-x}{k^*}$ types must take up at least as much space as w_i. Thus, $w(\hat{S}) \geq \binom{m^*-x}{k^*} k w_i$, where $w(\hat{S})$ is the total size of all the items on the $m^* - x$ containers of $\hat{S}$.

The size of all the items which are assigned to the $2m^*$ containers of $S_{<i}$ is $w(S_{<i}) = w(\bar{S}) + w(\hat{S}) \geq (m^* + x)(C - w_i) + \binom{m^*-x}{k^*} k^* w_i$. Again, OPT used m^* containers of capacity C so we know the total size of all the items cannot be more than m^*C. Thus,

$$(m^* + x)(C - w_i) + \binom{m^* - x}{k^*} k w_i \leq m^*C \tag{2}$$

By expanding the left hand side of (2) we get

$$m^*C - m^*w_i + xC - xw_i + \binom{m^* - x}{k^*} k^* w_i \leq m^*C$$

and rearranging terms gives us:

$$xC + \binom{m^* - x}{k^*} k^* w_i \leq m^* w_i + x w_i \tag{3}$$

By combining Eqs. 3 and 1with $w_i \leq C/3$ we get

$$3x w_i + \binom{m^* - x}{k^*} k^* w_i \leq m^* w_i + \frac{m^*}{2} w_i$$

which implies $\binom{m^*-x}{k^*} 2k^* + 6x \leq 3m^*$. Using the fact that $2 \leq k^* \leq \lfloor \frac{m^*}{2} \rfloor$ yields

$$\binom{m^* - x}{k^*} 4 + 6x \leq 3m^*. \tag{4}$$

It is a fact for any integers $a, b > 0$, where $b < a$, that $\binom{a}{b} \geq a$; and we know $m^* - x \geq k^*$ (since by Eq. (1) we know $x < m^*/2$ and we are currently in the case where $k^* \leq m^*/2$). Hence we can say from Eq. 4 that $4(m^* - x) + 6x \leq 3m^*$, which is a contradiction.

Both cases resulted in contradiction. So, RFF will never use more than $2m^*$ containers. □

4.2.3 Lower bound

We now provide a family of examples that give a lower bound on the approximation ratio of RFF. The family of examples is parameterized by a positive integer $d \geq 3$. We refer to the following instance as $I(d)$. There are $n = \binom{2d+3}{d}$ types, of which $\ell = 2d - 1$ are "large" types and $s = n - \ell$ are "small" types. The small types have size 1, while the large types have size $L = s$. Suppose the containers each have capacity $C = dL$. We first give an optimal assignment for this family.

Lemma 5 *For instance $I(d)$, an optimal assignment uses $m^* = 2d + 3$ containers.*

Proof First we note that since $d \geq 2$, we have $\frac{1}{d+1} < \frac{2d+3}{(d+3)(d+2)}$. Then

$$\frac{(2d+2)!}{(d+1)d!(d+1)!} < \frac{(2d+3)(2d+2)!}{d!(d+1)!(d+3)(d+2)},$$

from which we get

$$\binom{2d+2}{d+1} < \binom{2d+3}{d} = n.$$

By Sperner's Theorem, this says that instance $I(d)$ requires at least $m = 2d + 3$ containers and this number of containers is possible when $k = d$. Now, letting $k = d$, since

$$k\ell = d(2d-1) = 2d^2 - d \leq 2d^2 + d - 3 = (d-1)(2d+3) = (d-1)m,$$

we can store k items of each of the ℓ large types on the m containers with at most $d-1$ items on each container (by Lemma 2 in Sect. 4.1.1). Since the capacity of each container is $C = dL$, each container will have at least capacity L remaining. We will then use the remaining $\binom{2d+3}{d} - \ell$ combinations, which is exactly s, the number of small types, to assign the small items. By design, $Lm \geq ds$ and so there is enough remaining capacity to do this. Therefore $2d+3$ is the optimal number of containers for the instance $I(d)$. □

Given an instance $I(d)$, we now establish the number of containers returned by RFF.

Lemma 6 *For an instance $I(d)$, for each integer $1 \leq k \leq d+1$, we define*

$$y(k) = \min\left\{ j : \binom{j + 2d - 2k + 4}{j} \geq d - 1 \right\}$$
$$z(k) = \min\left\{ j : \binom{j}{k} \geq s \right\}$$

While using k items of each type, RFF *will return the number of containers equal to:*

$$m(k) = 2k - 1 - y(k) + z(k).$$

Then RFF *will return the number of containers such that $m(k)$ is minimal over $1 \leq k \leq d+1$.*

(Please refer to Table 3 for example values of $y(k)$ and $z(k)$).

Proof RFF begins by calculating that at least $2d+3$ containers are needed, and so RFF will loop from $k = d+1$ down to $k = 1$ in search of the minimum number of containers needed. In what follows, we index both the containers and item types starting from 0. Consider a fixed k for $1 \leq k \leq d+1$. RFF will assign each large item type t_i, for each $i = 0, \ldots, d-1$, to containers $\{0, \ldots, k-2, k-1+i\}$. Then the other remaining $d-1$ large types are assigned to containers $k-1, \ldots, 2k-2-j$ and some order j subset of $\{2k-1-j, \ldots, 2d+2\}$, which has cardinality $j+2d-2k+4$, for j large enough such that $\binom{j+2d-2k+4}{j} \geq d-1$. For any such j, the containers numbered 0 through $2k-2-j$ would be filled to capacity with large types. Thus calling $y(k)$ the minimum such j, we have that exactly the first $2k-1-y(k)$ containers are filled and the other containers have at least capacity L remaining.

Let $z(k)$ be the smallest positive integer such that $\binom{z(k)}{k} \geq s$. To assign the s small types, it is clear we need at least $z(k)$ containers beyond the $2k-1-y(k)$. For $d \geq 3$, we can prove by induction that

$$s = \binom{2d+3}{d} - (2d-1) > \binom{2d+2}{d+1}. \tag{5}$$

Table 3 The number of containers output by RFF and OPT for different values of d. RFF outputs the minimal $m(k)$ over $1 \le k \le d+1$ while OPT outputs $m^* = 2d+3$

d	k	$y(k)$	$z(k)$	RFF output	OPT output	RFF/OPT
5	5	1	13	21	13	1.615
8	8	2	19	32	19	1.684
9	8	2	22	35	21	1.666
15,000	10, 611	2	33, 185	54, 404	30, 003	1.81328
25,000	17, 663	2	55, 348	90, 671	50, 003	1.81331
35,000	24, 710	2	77, 521	12, 6938	70, 003	1.81332

This is true for $d = 3$, and assuming it is true for a particular d, then we multiply the lefthand side by $\frac{(2d+4)(2d+5)}{(d+1)(d+4)}$ and the righthand side by $\frac{(2d+3)(2d+4)}{(d+2)^2}$, the latter of which we can prove is smaller by cross-multiplying. We then get

$$\binom{2d+5}{d+1} - (2d-1)\frac{(2d+4)(2d+5)}{(d+1)(d+4)} > \binom{2d+4}{d+2}.$$

Now we can check by cross-multiplication that $(2d-1)\frac{(2d+4)(2d+5)}{(d+1)(d+4)} > (2d+1)$. Then $\binom{2d+5}{d+1} - (2d+1) > \binom{2d+4}{d+2}$, which is Eq. 5 with d replaced by $d+1$, completing the induction. Then $\binom{z(k)}{k} \ge \binom{2d+3}{d} - (2d-1) > \binom{2d+2}{d+1}$ implies $z(k) \ge 2d+3$.

Now note that by definition of $z(k)$ that $\binom{z(k)-1}{k} < s$. Because $z(k) \ge 2d+3$ and $k \le d+1 < z(k)/2$, then $\binom{z(k)-1}{k-1} < \binom{z(k)-1}{k} < s = L$. Thus $L \cdot z(k) \ge k\binom{z(k)}{k}$. By Lemma 2 (in Sect. 4.1.1), we can robustly assign each of $\binom{z(k)}{k}$ small types to k out of $z(k)$ containers each with capacity at least L and in particular RFF would naturally do this because every combination of k out of $z(k)$ containers is used. Since $s \le \binom{z(k)}{k}$, then RFF would successfully use $z(k)$ containers to robustly assign the s small types. Thus we have shown that RFF with $k \le d+1$ items of each type uses $2k-1-y(k)+z(k)$ containers. Thus RFF uses the number of containers equal to the minimum of $2k-1-y(k)+z(k)$ for $1 \le k \le d+1$. □

Theorem 4 *The approximation ratio of* RFF *is no better (lower) than 1.813.*

Proof Let $d = 15000$, and consider the instance $I(d)$ as defined above. By Lemma 6, RFF ends up using $k = 10611$ and $y = 2, z = 33185$ and $m = 54404$ for this instance, while (by Lemma 5) an optimal assignment requires only $m^* = 2d+3 = 30003$. (Please see Table 3.) □

5 Experimental results

As described in Sect. 1, RAP can be applied to assigning app instances to the minimal number of servers on a hosting platform while ensuring that if a failure occurs in

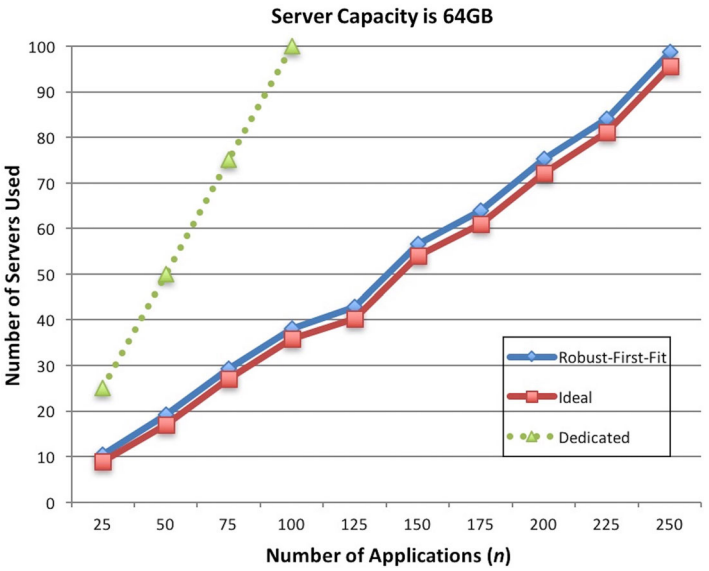

Fig. 4 Number of servers used when server capacity is 64 GB

an app and therefore all its hosting servers are temporarily suspended, there is still a running instance of every other app hosted on some unaffected server. Formally, we are given n apps where app i has size d_i and server capacity C. We would like to find an assignment of app instances to the minimal number of servers m, such that the assignment is robust and satisfies the capacity constraint. To evaluate the performance of the RFF algorithm, we simulated a hosting platform and measured the number of servers used by the algorithm. Specifically, we tested four values for server capacity C (64 GB, 128 GB, 256 GB, and 512 GB), varied the number of apps from $n = 25$ to $n = 250$ apps (at increments of 25) and set app sizes d_i to be normally distributed between 4 and 16 GB. We compared RFF to a dedicated system (i.e., where the number of servers is simply the number of apps) and an "ideal" assignment, which does not correspond to any feasible robust assignment, but serves as a lower bound on the minimally required number of servers. (Recall that it even without the robustness constraint, it is NP-hard to compute OPT so we did not compute it for the experiments.) We computed the "ideal" assignment by determining the minimum number of servers needed to satisfy robustness alone and the minimum number of servers to satisfy the storage constraints alone and taking the maximum of these two values. I.e., the "ideal" number of servers is defined as: $\min_k \max\{m_r, m_c\}$ where $m_c = \min\{m : mC \geq k\sum_{i=1}^{n} d_i\}$ and $m_r = \min\{m : \binom{m}{k} \geq n\}$. We tested each setting for 10 iterations and took the average of the results. Figures 4, 5, 6, 7 show the results. The graphs show that for all settings, RFF performs significantly better than the dedicated system and almost as well as the ideal assignment. Specifically, the worst (minimum) ratio (over all values of n) of servers used by the dedicated system and RFF is 2.40, 3.25, 3.57, and 3.57 for 64 GB, 128 GB, 256 GB, and 512 GB, so RFF always assigned apps more than twice as efficiently as a dedicated system. Note

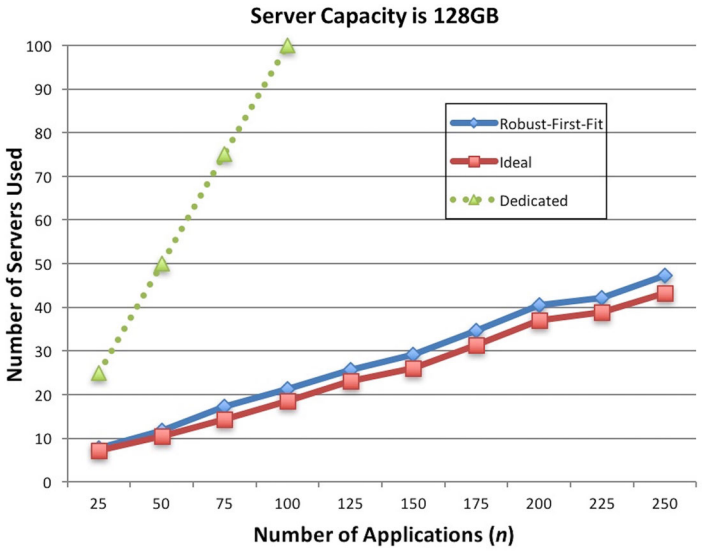

Fig. 5 Number of servers used when server capacity is 128 GB

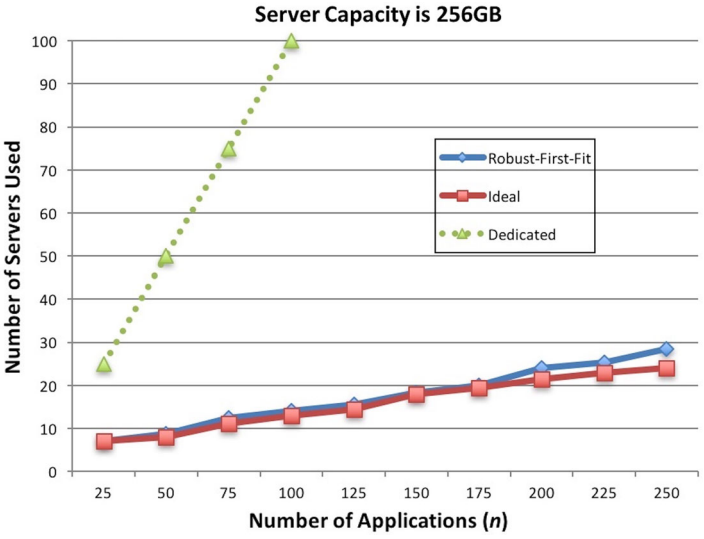

Fig. 6 Number of servers used when server capacity is 256 GB

that as the server capacity increases, these ratios either increase or stay the same. The average ratio of servers used by the dedicated system and RFF always increase: 2.64, 4.72, 7.34, and 9.89 for 64 GB, 128 GB, 256 GB, and 512 GB, respectively, so RFF on average performed as much as 9 times as efficiently as a dedicated hosting.

Comparing RFF with the lower bound on optimal, we find that the worst (maximum) ratio (over all values of n) of servers used by RFF and the ideal assignment is 1.17, 1.21,

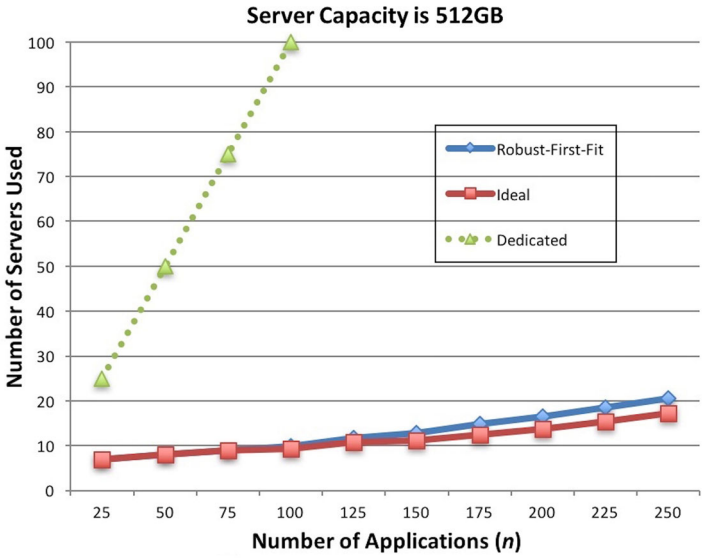

Fig. 7 Number of servers used when server capacity is 512 GB

1.13, and 1.20 for 64 GB, 128 GB, 256 GB, and 512 GB, respectively. So RFF always used close to the same number of servers as an optimal solution. (The average ratios are similar to these values.) The results indicate that when apps sizes are more realistic than those described in Theorem 4 of Sect. 4.2.3, RFF performs close to optimally.

6 Discussion and conclusions

We proposed a new model for assigning items of various types to containers such that the system is *robust*. We presented an optimal poly-time algorithm in the setting without capacity constraints on the containers. We also presented an optimal poly-time algorithm when item sizes are uniform. Our main algorithm RFF is a poly-time 2-approximation algorithm for the setting where item sizes are nonuniform. Our experimental results suggest that when run on a simulated hosting platform, RFF performs well not only in the worst-case, but even more so on average.

In the lower bound instance, as d increases, it is not clear whether the corresponding ratio is converging (very slowly) to 2 or to a number less than 2, or whether the ratio converges at all; if the ratio does not converge, one can still ask for the limit supremum of the sequence of ratios. If the limit supremum is 2, then the upper bound of 2 is tight.

One direction for future work is to determine whether there is an algorithm with an approximation ratio better than 2. Also, our problem model assumes that the number of items is uniform over all types. A natural extension of this work would be to consider the case where this number is not required to be uniform.

Acknowledgements The authors would like to thank Samuel Barnes and Nate Devine for their helpful corrections on an earlier version of this work.

References

Chekuri C, Khanna S (2000) A PTAS for the multiple knapsack problem. In: Symposium on discrete algorithms (SODA)

Epstein L, Levin A (2006) On bin packing with conflicts. In: Proceedings of the workshop on approximation and online algorithms (WAOA)

Fleischer L, Goemans MX, Mirrokni VS, Sviridenko M (2006) Tight approximation algorithms for maximum general assignment problems. In: Proceedings of the symposium on discrete algorithms

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York

Jansen K (1999) An approximation scheme for bin packing with conflicts. J Comb Optim 3(4):363–377

Jansen K, Öhring S (1997) Approximation algorithms for time constrained scheduling. Inf Comput 132(2):85–108

Korupolu M, Rajaraman R (2016) Robust and probabilistic failure-aware placement. In: Proceedings of the symposium on parallelism in algorithms and architectures (SPAA), pp 213–224

Korupolu M, Meyerson A, Rajaraman R, Tagiku B (2015) Robust and probabilistic failure-aware placement. Math Program 154(1–2):493–514

Mills K, Chandrasekaran R, Mittal N (2017) Algorithms for optimal replica placement under correlated failure in hierarchical failure domains. In: Theoretical computer science (pre-print)

Rahman R, Barker K, Alhajj R (2008) Replica placement strategies in data grid. J Grid Comput 6(1):103–123

Shmoys D, Tardos E (1993) An approximation algorithm for the generalized assignment problem. Math Program 62(3):461–474

Sperner E (1928) Ein Satz über Untermengen einer endlichen Menge. Math Z 27(1):544–548

Stein C, Zhong M (2018) Scheduling when you don't know the number of machines. In: Proceedings of the symposium on discrete algorithms (SODA)

Stirling J (1730) Methodus differentialis, sive tractatus de summation et interpolation serierum infinitarium. London

Urgaonkar B, Rosenberg A, Shenoy P (2007) Application placement on a cluster of servers. Int J Found Comput Sci 18(5):1023–1041