Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 352 (2006) 250–265

Theoretical Computer Science

www.elsevier.com/locate/tcs

Efficient algorithms for robustness in resource allocation and scheduling problems[☆]

Greg N. Frederickson[a,1], Roberto Solis-Oba[b,*,2]

[a]Department of Computer Science, Purdue University, West Lafayette IN 47907, USA
[b]Department of Computer Science, The University of Western Ontario, London, ON, N6A5B7, Canada

Received 12 August 2004; accepted 18 November 2005

Communicated by G.F. Italiano

Abstract

The *robustness function* of an optimization (minimization) problem measures the maximum increase in the value of its optimal solution that can be produced by spending a given amount of resources increasing the values of the elements in its input. We present efficient algorithms for computing the robustness function of resource allocation and scheduling problems that can be modeled with partition and scheduling matroids. For the case of scheduling matroids, we give an $O(m^2n^2)$ time algorithm for computing a complete description of the robustness function, where m is the number of elements in the matroid and n is its rank. For partition matroids, we give two algorithms: one that computes the complete robustness function in $O(m \log m)$ time, and other that optimally evaluates the robustness function at only a specified point.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Robustness; Optimization; Scheduling; Partition matroid

1. Introduction

The *robustness function* of an optimization (minimization) problem measures the maximum increase in the value of its optimal solution that can be produced by spending a given amount of resources increasing the values of the elements in its input. The robustness function of a problem can be used to assess the quality of its solution when the values of the elements in its input are not known exactly, and hence, estimates have to be used for them [18,24,25]. In this paper we present efficient algorithms for computing the robustness function of resource allocation and scheduling problems that can be modeled with partition and scheduling matroids. We assume a cost model that for any element e of a matroid, charges $\delta * c(e)$ to increase the value of e by δ, where $c(e)$ is a non-negative value.

☆ A preliminary version of this paper appeared in the Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 659–668.
* Corresponding author. Tel.: 519 6612111x86974; fax: 519 6613515.
E-mail addresses: gnf@cs.purdue.edu (G.N. Frederickson), solis@csd.uwo.ca (R. Solis-Oba).
[1] Research of this author partially supported by the National Science Foundation under Grant CCR-9322501.
[2] Part of this research was done while this author was at Purdue University.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.11.023

In [14] we presented an algorithm for computing the robustness function of an arbitrary matroid optimization problem. In this paper we take advantage of the special structure of scheduling and partition matroids to design algorithms which are much more efficient than the general algorithm in [14]. Our algorithm for scheduling matroids is also faster that the more general algorithm for transversal matroids that we give in [14]. A very brief description of the algorithms that we present here was given in [12].

The concept of robustness function has been considered before in the context of minimum spanning trees [13], shortest paths [15], maximum flows in planar graphs [25], and intersections of matroids [21]. Related problems have also been considered. Drangmeister et al. [9] give a constant-approximation algorithm for the problem of spending a fixed budget reducing the weights of the edges of a given graph to minimize the weight of its minimum spanning trees. Berman et al. [4] study the problem of shortening the weights of the edges of a given rooted tree to minimize the sum of the distances from the root to all of the other vertices in the tree. Ahuja and Orlin [2] consider the problem of spending a fixed budget increasing the capacities of the edges in a flow network so as to maximize the value of a maximum flow. Burkard et al. [6] study the bottleneck capacity expansion problem which calls for increasing the values of the elements in the input of a bottleneck problem so maximize the value of its optimum solution.

We showed in [14] that the robustness function of any matroid is piecewise-linear and that it has $O(mn)$ breakpoints, where m is the number of elements in the matroid and n is its rank. Our algorithm for computing the robustness function of a scheduling matroid finds all the breakpoints in $O(m^2n^2)$ time. Each breakpoint is computed by solving a series of scheduling-with-preemption subproblems in which some subset of elements of the matroid must be scheduled to completion. We present a formulation for these scheduling subproblems that corresponds to a generalization of the *off-line min* problem [1]. This reformulation allows us to solve optimally each subproblem.

For partition matroids, we give two algorithms. The first algorithm computes all the breakpoints of the robustness function in $O(m \log m)$ time. As the algorithm computes the breakpoints, it identifies increasingly larger clusters of elements that must undergo the same weight increases in the determination of the remaining breakpoints. These clusters have a certain "convexity" property that we exploit to compute each breakpoint in $O(\log m)$ time. The increasing size of the clusters allows us to prove that the robustness function of a partition matroid has only $O(m)$ points.

Our second algorithm for partition matroids does not compute all the breakpoints of the robustness function, but evaluates the robustness function only at a required point, and does so in $O(m)$ time. This new formulation for the problem is a version of the optimal distribution of effort problem (see e.g. [22]). All known methods of solution for the latter problem assume explicit representations for the gain functions that make it possible to evaluate them efficiently [11,22]. Such representations are not available in our robustness problem, and thus, we cannot use those methods. Instead, we present a new approach that optimally solves the problem by a sophisticated application of a linear-time selection algorithm [5], that interleaves searches on weights with searches on costs.

The rest of the paper is organized as follows. In Section 2 we review the general algorithm in [14] to compute the robustness function of a matroid. In Section 3 we present our algorithm for computing the robustness function of a scheduling matroid. In Section 4 we describe the algorithm for computing all the breakpoints of the robustness function of a partition matroid. In Section 5 we present an optimal algorithm for evaluating the robustness function of a partition matroid at only a given point.

2. Preliminaries

A *matroid* $M = (E, \mathcal{I}, w, c)$ consists of a finite set E of *elements* and a collection $\mathcal{I}$ of subsets of E satisfying well-known axioms (see e.g. [27]). Function w assigns a non-negative weight to each element in E. We assume that the weights are not fixed, but they can be changed, and c indicates the cost of each unit increase in the weight of an element. If an element $e \in E$ increases its weight $w(e)$ by some amount $\delta > 0$, a total cost $\delta c(e)$ is incurred. Set E is called the *ground set* of the matroid M, and the subsets in $\mathcal{I}$ are called the *independent sets* of the matroid. An independent set of maximum size is a *base* of M. The number of elements in any base is the *rank* of the matroid. We denote by n the rank of a matroid and by m the size of its ground set.

The robustness function $F_M(b)$ of matroid M measures the maximum increase in the weight of the minimum weight bases of M that can be obtained by increases of total cost b on the weights of its elements. In this section we briefly review the algorithm in [14] for computing the robustness function of an arbitrary matroid.

Given a set $S \subseteq E$, we define *coverage* (S, M) as the minimum number of elements that any minimum weight base of M shares with S. Let *tolerance* (S, M) be the minimum amount by which the weight of each element in S has to be increased to reduce *coverage* (S, M). The *rate* of S in M, denoted as *rate* (S, M), is defined as $c(S)/coverage\,(S, M)$, where $c(S)$ is the sum of the costs of the elements in S. Note that if the weight of each element in set S is increased by some value $\delta \leqslant tolerance\,(S, M)$, then the value of *rate* $(S, M) * \delta$ gives the cost for increasing the weight of every minimum weight base of M by at least δ.

The following algorithm [14] outputs all the breakpoints of the robustness function of a given matroid M.

```
**Algorithm** *uplift* $(M)$
*value* $\leftarrow$ weight of a minimum weight base of $M$
*cost* $\leftarrow 0$
Output (*cost*, *value*).
Find a set $S \subseteq E$ of smallest rate $r$ in $M$.
**while** (*tolerance* $(S, M) \neq \infty$) **do**
  Increase the weights of the elements in $S$ by *tolerance* $(S, M)$.
  *cost* $\leftarrow$ *cost* $+$ *tolerance* $(S, M) * c(S)$
  Find a set $S' \subseteq E$ of smallest rate in $M$.
  **if** *rate* $(S', M) < r$ **then** Output (*cost*, *value*) **end if**
  $S \leftarrow S'$
**end while**
```

The most crucial part of algorithm *uplift* is how to find a set of smallest rate in the matroid. Both of our algorithms for computing all the breakpoints of the robustness functions of scheduling and partition matroids use algorithm *uplift*. However, we take advantage of the special structure of these matroids to design efficient algorithms for finding a set of smallest rate.

3. Scheduling matroids

Consider the following scheduling problem. Let $J = \{j_1, j_2, \ldots, j_m\}$, be a set of jobs. Each job j_i requires one unit of processing time and has a weight $w(j_i)$. Job j_i has integer release time r_i and deadline d_i, with $d_i > r_i$. Without loss of generality, we assume that the largest deadline has value at most m. The problem is to select a largest subset of jobs of minimum total weight that can be executed on a single processor.

This problem can be reduced to a weighted matching problem on a *convex* bipartite graph $G = (J \cup T, A)$ (see e.g. [16]). We think of the ith vertex of J as job j_i, and the ith vertex of T as the time interval from $i - 1$ to i. For each job j_i, graph G has edges of weight $w(j_i)$ from j_i to each time interval between r_i and d_i. It can be proved that a maximum cardinality matching of minimum weight in G corresponds to a scheduling of a largest subset of J of minimum total weight [16].

The set of matchings of a bipartite graph can be modeled with a *transversal matroid* (see e.g. [16]). For the special case of a convex bipartite graph, the corresponding transversal matroid is called a *scheduling matroid*. A scheduling matroid $M = (J, \mathcal{I}, w, c)$ has a set J of jobs as its ground set and a subset of jobs is independent if and only if there is a feasible schedule for them.

In [14] it is shown that a set of smallest rate in a transversal matroid can be computed by performing a series of minimum-cut computations over bipartite graphs. We briefly review the approach in [14]. A bipartite graph $G = (J \cup J', E)$ defines a transversal matroid $M_T = (J, E, w, c)$ with ground set J, the set of vertices on one side of G, and whose independent sets are the subsets of J which can be covered by a matching in G. Let $B \subseteq J$ be a maximum independent set of minimum weight in M_T. Let $w_1, w_2, \ldots, w_p$ be the different weights of elements in M_T. Given a set $S \subseteq J$, let $S_{=w_k}$ be the set formed by the elements of weight w_k in S, and let $S_{\neq w_k}$ be the set of elements in S of weight different from w_k.

For $k = 1, 2, \ldots, p$, build auxiliary graphs $G_k = (\{s_k, t_k\} \cup J_{=w_k} \cup B_{\neq w_k} \cup J', E_k)$, with E_k as follows. Given a parameter λ, E_k has an edge of capacity $\min\{1, c(v)/\lambda\}$ from s_k to each vertex $v \in J_{=w_k}$, and an edge of capacity 1 from s to each vertex in $B_{\neq w_k}$. There is an edge of infinite capacity from each vertex $v \in J_{=w_k} \cup B_{\neq w_k}$ to a vertex $w \in J'$ whenever the edge (v, w) is in G. Finally, E_k has an edge of capacity 1 from each vertex in J' to t_k (see Fig. 1).

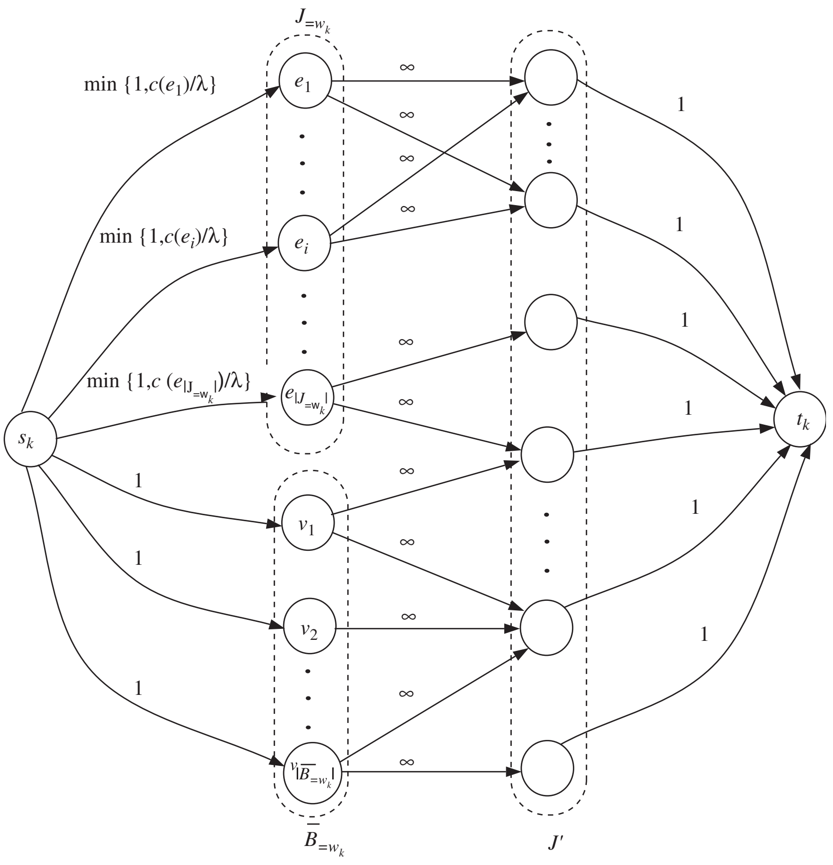
$J_{=w_k}$
min $\{1, c(e_1)/\lambda\}$
min $\{1, c(e_i)/\lambda\}$
min $\{1, c(e_{|J_{=w_k}|})/\lambda\}$
e_1
e_i
$e_{|J_{=w_k}|}$
s_k
t_k
v_1
v_2
$v_{|\overline{B}_{=w_k}|}$
∞
1
$\overline{B}_{=w_k}$
J'

Fig. 1. Auxiliary graph G_k.

Lemma 3.1 (*Frederickson and Solis-Oba [14]*). *A set of smallest rate in a transversal matroid $M_T = (J, E, w, c)$ has rate λ^* equal to the largest parameter λ such that the maximum flow in each auxiliary graph G_k has value $|B|$.*

Lemma 3.2 (*Frederickson and Solis-Oba [14]*). *Let G_k be an auxiliary graph such that for every parameter $\lambda > \lambda^*$ a maximum flow of G_k has value smaller than $|B|$. For parameter λ^*, let z_k^* be a maximum flow of G_k that saturates every edge from s_k to $B_{\neq w_k}$. Set $S \subseteq J_{=w_k}$ formed by all elements $v \in J_{=w_k}$ for which $z_k^*(s_k, v) = c(v)/\lambda^*$ is a set of smallest rate in M_T.*

These lemmas can be used to design an efficient algorithm for computing a set of smallest rate in a scheduling matroid $M = (J, \mathcal{I}, w, c)$. Consider one of the auxiliary graphs G_k. It is known [10,26] that in an s-t graph in which the capacities of the edges leaving the source are linear functions of a parameter $1/\lambda$, the value of a maximum flow is a piecewise-linear concave function of $1/\lambda$. Let $f_k(1/\lambda)$ denote this flow function for auxiliary graph G_k. By definition of B, there is a matching of maximum cardinality in G that covers the vertices in B. Therefore, for any value of the parameter $1/\lambda$, there is a maximum flow of G_k that saturates at least $|B_{\neq w_k}|$ edges incident to t_k. Let us consider this maximum flow. Note that if λ is decreased, the value of the flow increases linearly with $1/\lambda$ until one more edge incident to t_k is saturated. Since a maximum matching of G has size $|B|$, then for any value of $1/\lambda$ at most $|B|$ of the edges incident to t_k can be saturated by a flow. This implies that $f_k(1/\lambda)$ has at most $|B_{=w_k}|$ breakpoints.

It is easy to see that a set of smallest rate in $M = (J, \mathcal{I}, w, c)$ has rate $\lambda^* \leqslant c(J)/n$. To compute the value of λ^*, we use Newton's method (see e.g. [23]) on each function $f_k(1/\lambda)$ to find the largest value λ_k for which $f_k(1/\lambda_k) = |B|$. By the above discussion, the value of λ_k can be found by performing at most $|B_{=w_k}|$ maximum flow computations on G_k. The value of λ^* is equal to the smallest λ_k, for $k = 1, 2, \ldots, p$.

We show that a maximum flow of G_k that saturates all edges from s_k to $B_{\neq w_k}$ can be computed in $O(|J_{=w_k} \cup B_{\neq w_k}|)$ time, whereas the previous best algorithm for finding a maximum flow in a bipartite graph G_k needs $O(|J_{=w_k} \cup B_{\neq w_k}||E_k| \log(|J_{=w_k} \cup B_{\neq w_k}|^2/|E_k|+2)$ time [3]. We describe first how to find in linear time a maximum flow for G_k, and then we show how to modify it so that it saturates all edges from s_k to $B_{\neq w_k}$ as required by Lemma 3.2.

Fix one of the auxiliary graphs $G_k = (\{s_k, t_k\} \cup J_{=w_k} \cup B_{\neq w_k} \cup J', E_k)$. The problem of computing a maximum flow of G_k can be interpreted as a scheduling-with-preemption problem on a single processor. Let the vertices $\tau_k \in J'$ be time slots and the vertices $j_i \in J_{=w_k} \cup B_{\neq w_k}$ be jobs. The capacity of edge (s_k, j_i) is the processing time required by job j_i, and each infinite capacity edge (j_i, τ_k) identifies a time slot τ_k where job j_i can be scheduled. Under this interpretation, any flow function for G_k defines a feasible preemptive schedule for various portions of the jobs in $J_{=w_k} \cup B_{\neq w_k}$: the value of the flow on edge (s_k, j_i) determines the total time that job j_i is scheduled for execution, and the flow on edge (j_i, τ_k) defines the portion of job j_i that is scheduled in time interval τ_k. The converse of this statement is also true, i.e., a preemptive schedule that specifies the execution order of portions of the jobs in $J_{=w_k} \cup B_{\neq w_k}$ on a single processor defines a valid flow for G_k.

By the previous discussion, any algorithm that finds a preemptive schedule for the jobs in $B_{\neq w_k} \cup J_{=w_k}$ that maximizes the amount of time that the jobs are executed on a single processor, also computes a maximum flow for G_k. We present below an efficient implementation of the *preemptive earliest deadline first rule* [20], which finds in linear time an optimal preemptive schedule for the jobs $B_{\neq w_k} \cup J_{=w_k}$, and thus, a maximum flow for G_k.

We assume that the largest deadline among the jobs in $J_{=w_k} \cup B_{\neq w_k}$ is at most $|J_{=w_k} \cup B_{\neq w_k}|$. If this condition does not hold, we can modify the deadlines so that this condition is satisfied [12]. This preprocessing step requires $O(|J_{=w_k} \cup B_{\neq w_k}|)$ time. The extra time required by this step does not affect the overall time complexity of our algorithm.

A straightforward implementation of the preemptive earliest deadline first rule uses an initially empty set T to store all those jobs that can be scheduled at a given time. Starting at the earliest release time, and moving forward in time, a schedule is generated as follows. A job is added into T whenever it becomes available for scheduling, and in each time interval (a fraction of) the job with smallest deadline in T is scheduled.

Our scheduling problem can be seen as an extension of the *off-line min problem* defined in [1]. In this latter problem we are given a set of integers $\{1, 2, \ldots, i\}$, an initially empty set T, and two operations: *insert*, that adds an integer to T, and *extract_min*, that removes the smallest integer from T. It is desired to maintain T under a given sequence of *insert* and *extract_min* operations, assuming that each integer is inserted in T only once.

As in an off-line min problem, we want to maintain in our scheduling problem a set of jobs T under some sequence $\mathcal{S}$ of two operations: insertion of all the jobs with a particular release time, and extraction of a job with smallest deadline. However, contrary to the off-line min problem, we might have several jobs with the same deadline, and we do not know in advance how many jobs will be extracted between two consecutive insertion operations. We define the *generalized off-line min problem* over a set of elements $\{e_1, e_2, \ldots, e_m\}$ as follows. Each element has two attributes, an integral *value* from the domain $\{1, 2, \ldots, m\}$, and a positive real *magnitude* no larger than 1. There is an initially empty set T over which two operations are defined, *insert* and *retrieve*. Operation *insert* adds an element to T. Operation *retrieve* extracts from T elements of smallest value until either T is empty or total magnitude 1 is achieved, splitting the magnitude of the last element before extraction, as necessary. Given a sequence $\mathcal{S}$ of *insert* and *retrieve* operations, the generalized off-line min problem consists in finding which portions of what elements are removed by each *retrieve* operation.

Consider the following instance of the generalized off-line min problem defined over the set $\{a, b, c, d, e\}$. The values of the elements are 1, 1, 3, 4, and 2, respectively, and their magnitudes are 0.3, 0.5, 0.4, 0.8, and 1.0. The sequence $\mathcal{S}$ is: *insert* a, *insert* b, *insert* c, *retrieve*, *insert* d, *insert* e, *retrieve*, *retrieve*. The first *retrieve* operation extracts from T elements a and b, and half of element c (so it leaves in T the rest of element c with magnitude 0.2). The second *retrieve* operation extracts element e, while the third *retrieve* extracts the remaining of c plus element d.

To solve the generalized off-line min problem, write the sequence $\mathcal{S}$ as $I_1, R, I_2, R, \ldots, I_q, R$, where R is a *retrieve* operation and each I_i is a sequence of *insert* operations. Let A_i be the set of elements inserted by I_i. As in [1], we use disjoint-set data structures to represent each set A_i; these sets are stored in a doubly linked list with $succ(A_i)$ the successor of A_i. We use an array Q, and store in $Q(i)$ a list with the elements extracted by the ith *retrieve* operation. Let the elements be indexed non-decreasingly by value. Function FIND(x) returns the set to which element x belongs and function UNION(x, y, y) on sets x and y, makes $y \leftarrow x \cup y$. The algorithm to solve the generalized off-line min problem is the following.

```
Algorithm generalized_off-line($\mathcal{S}$, $\{e_1, e_2, \ldots, e_m\}$)
Initialize each $Q(i)$ to $\emptyset$, and set $j \leftarrow 1$.
while $j < m$ do
  $A_i \leftarrow$ FIND($e_j$).
  Add $e_j$ to the end of list $Q(i)$.
  Find $S_m$, the sum of magnitudes of the elements in $Q(i)$.
  if $S_m \geqslant 1$ then
    UNION($A_i$, $succ(A_i)$, $succ(A_i)$)
    $magnitude\,(e_j) \leftarrow S_m - 1$
  end if
  if $S_m \leqslant 1$ then $j \leftarrow j + 1$ end if
end while
return ($Q$)
```

Lemma 3.3. *Algorithm generalized_off-line runs in* O(m) *time.*

Proof. We can use *counting sort* (see e.g. [7]) to index the elements non-decreasingly by value, as required by *generalized_off-line*, in O(m) time. Note that the only UNION operations that the algorithm performs are of the form UNION(A_i, $succ\,(A_i)$, $succ\,(A_i)$). This special order of the UNION operations allows us to use the algorithm of Gabow and Tarjan [17] to perform all the UNION and FIND operations in O(m) time. □

We use algorithm *generalized_off-line* to find an optimal scheduling for the jobs in $J_{=w_k} \cup B_{\neq w_k}$ as follows. We initialize the data structures of *generalized_off-line* by storing in set i all jobs with the ith smallest release time. The attribute magnitude of each job is set equal to the processing time for the job and the attribute value is equal to the deadline of the job. The jobs are indexed non-decreasingly by deadline. Run algorithm *generalized_off-line* with the following slight change: in each iteration of the while-loop, after finding the set i containing job e_j, *generalized_off-line* inserts e_j in $Q(i)$ only if its deadline is at least i (otherwise job e_j cannot be scheduled in the ith time slot). The schedule can easily be obtained from the information that *generalized_off-line* stores in Q.

Lemma 3.4. *An optimal scheduling for the jobs $J_{=w_k} \cup B_{\neq w_k}$ can be computed in* O($|J_{=w_k} \cup B_{\neq w_k}|$) *time and* O($|J_{=w_k} \cup B_{\neq w_k}|$) *space.*

The maximum flow of G_k corresponding to the schedule generated by *generalized_off-line* might not saturate all edges from s_k to $B_{\neq w_k}$. We show below how to find a flow that complies with this condition.

First, modify the deadlines of the jobs in $B_{\neq w_k}$ so that no two of them have the same deadline. To do this, find an optimal schedule for the jobs in $B_{\neq w_k}$ using a time-reversed version of the preemptive earliest deadline first rule. This new rule schedules jobs by starting at the largest deadline and moving backwards in time towards the smallest release time. The rule schedules in each interval the job with latest release time that is available. The correctness of this rule can be easily established. Let S' be the schedule for the jobs in $B_{\neq w_k}$ obtained with this rule. Modify the deadline of each job $j_i \in B_{\neq w_k}$ by making it equal to the completion time of job j_i in S'. Let d'_i be the modified deadline for job j_i.

Then, modify algorithm *generalized_off-line* so that whenever two or more jobs have the same modified deadline, it chooses to schedule first the jobs from $B_{\neq w_k}$. (The only change that we actually need to make is to index the jobs so that if several jobs have the same modified deadline, the jobs from $B_{\neq w_k}$ are indexed first.) Use this modified *generalized_off-line* algorithm to find a schedule S^* for the jobs in $J_{=w_k} \cup B_{\neq w_k}$.

Lemma 3.5. *The schedule S^* maximizes the amount of time that the jobs in $J_{=w_k} \cup B_{\neq w_k}$ are executed on a single processor, and it schedules to completion all jobs from $B_{\neq w_k}$.*

Proof. We need only prove that there is an optimal schedule $\hat{S}$ for the jobs in $J_{=w_k} \cup B_{\neq w_k}$ in which all jobs from $B_{\neq w_k}$ are scheduled to completion, and such that no job $j_i \in B_{\neq w_k}$ is scheduled after its modified deadline d'_i. The proof is by contradiction.

Let S' be the schedule used to determine the modified deadlines d'_i. Suppose there is no optimal schedule $\hat{S}$ for the modified deadlines, but there is one for the original deadlines. Choose S to be an optimal schedule for the original deadlines that schedules to completion all jobs from $B_{\neq w_k}$, such that the first job $j_i \in B_{\neq w_k}$ that is not completed by its modified deadline has d'_i as large as possible. Since jobs are scheduled in S' as late as possible, there must be at least one job $j_k \in B_{\neq w_k}$ with $d'_k > d'_i$ and $r_k \geqslant r_i$ that is scheduled in S to be completed at a time $\ell_k < d'_i$. Clearly, we can modify S so that j_i is scheduled within the portions of time alloted to j_k, and j_k is scheduled in the time intervals assigned to j_i. This modification to S produces a new optimal scheduling in which the completion time for j_i does not exceed d'_i. This is a contradiction. □

Theorem 3.1. *The robustness function of a scheduling matroid $M = (J, \mathcal{I}, w, c)$ can be computed in* $O(m^2n^2)$ *time, where $m = |J|$ and $n = |B|$ is the size of a maximum independent set in $\mathcal{I}$.*

Proof. In each auxiliary graph G_k, the largest value λ_k for which a maximum flow has value $|B|$ can be computed in $O(|B_{=w_k}||J_{=w_k} \cup B_{\neq w_k}|)$ time. Hence, the total time used to find a set of smallest rate in the scheduling matroid M is $O(\sum_{i=1}^{p} |B_{=w_k}||J_{=w_k} \cup B_{\neq w_k}|) = O(|B||D| + |B|^2) = O(mn)$. Since algorithm *uplift* computes $O(mn)$ sets of smallest rate, the total time needed to compute all the breakpoints in the robustness function in $O(m^2n^2)$. □

4. Partition matroids

A *partition matroid* $P = (E, \mathcal{I}, w, c)$ is defined over a finite set E of m elements, partitioned into ℓ disjoint *blocks* $E_1, E_2, \ldots, E_\ell$. Given a set of ℓ positive integers $n_i \leqslant |E_i|, i = 1, \ldots, \ell$, a set $S \subseteq E$ is independent in P if and only if $|S \cap E_i| \leqslant n_i$, for all $1 \leqslant i \leqslant \ell$. A minimum weight base of P is a minimum weight subset B of E such that $|B \cap E_i| = n_i$, for all $1 \leqslant i \leqslant \ell$.

We use algorithm *uplift* to compute the robustness function of a partition matroid. However, we take advantage of the special structure of these matroids to design a very efficient algorithm for finding a set of smallest rate. Consider a partition matroid $P = (E, \mathcal{I}, w, c)$. The blocks E_i are independent in the sense that any change in the weights of the elements in some block E_i does not affect the set of elements of another block E_j that might belong to a minimum weight base. Hence, there must be at least one block E_i that contains a subset of elements having the smallest rate in P. This observation simplifies the problem. We need only show how to compute a set of smallest rate in one of the blocks E_i, or, equivalently, how to find a set of smallest rate in a *uniform matroid*. A uniform matroid is a partition matroid in which $\ell = 1$. In a uniform matroid $P = (E, \mathcal{I}, w, c)$ of rank n a set $S \subseteq E$ is independent if and only if $|S| \leqslant n$.

Fix a uniform matroid $U = (E, \mathcal{I}, w, c)$ with rank n. Let $m = |E|$. Let w_n be the weight of the nth smallest element in E. Note that the only elements of E that can belong to a minimum weight base of U are those elements of weight at most w_n. Let $E_{<w_n}$, $E_{=w_n}$, and $E_{>w_n}$ denote the subsets of E formed by all elements of weight smaller than w_n, equal to w_n, and larger than w_n, respectively. Let $\Delta = |E_{<w_n}| + |E_{=w_n}| - n$.

Lemma 4.1. *If $S \subseteq E$ is a set of smallest rate in U then*

$$\mathit{rate}(S, U) = \min\{\min\{c(e) | e \in E_{<w_n}\}, \min\{c(T)/(|T| - \Delta) | T \subseteq E_{=w_n} \text{ and } |T| > \Delta\}\}$$

Proof. For any set $T \subseteq E_{<w_n}$, *coverage* $(T, U) = |T|$ since all elements in $E_{<w_n}$ are in every minimum weight base of U. Hence, if $S \subseteq E_{<w_n}$, then S must contain only the element of smallest cost in $E_{<w_n}$. Also, for any set $T \subseteq E_{=w_n}$, *coverage* $(T, U) = \max\{0, |T| - \Delta\}$ because there is a minimum weight base of U that contains $\min\{n - |E_{<w_n}|, |E_{=w_n}| - |T|\}$ elements from $E_{=w_n} - T$. Thus, if $S \subseteq E_{=w_n}$, then S minimizes $c(S)/(|S| - \Delta)$.

Finally, note that for every set $T \subseteq E_{<w_n} \cup E_{=w_n}$, *rate* $(T, U) \geqslant \min$ {*rate* $(T \cap E_{<w_n}, U)$, *rate* $(T \cap E_{=w_n}, U)$}. □

By Lemma 4.1, there is a set of smallest rate in U that either consists of a single element of smallest cost in $E_{<w_n}$, or that is formed by the κ elements of smallest cost in $E_{=w_n}$, for some $\Delta < \kappa \leqslant |E_{=w_n}|$. The following observation allows us to compute efficiently the value of κ. Let $\{e_1, e_2, \ldots, e_{|E_{=w_n}|}\}$ be the elements of $E_{=w_n}$ indexed in non-decreasing order of cost.

Lemma 4.2. *The discrete function $f(i) = \sum_{j=1}^{i} c(e_j)/(i - \Delta)$ with integer argument i is strictly decreasing in the interval $\Delta + 1 \leqslant i \leqslant \kappa$, and non-decreasing in the interval $\kappa \leqslant i \leqslant |E_{=w_n}|$, where κ is the smallest integer where f reaches its minimum.*

Proof. We show that if $f(i) \geqslant f(i-1)$ for any $\Delta + 2 \leqslant i \leqslant |E_{=w_n}| - 1$, then $f(i+1) \geqslant f(i)$. This proves the lemma. If $f(i) \geqslant f(i-1)$, then

$$(i - 1 - \Delta)\sum_{j=1}^{i} c(e_j) \geqslant (i - \Delta)\sum_{j=1}^{i-1} c(e_j), \text{ hence,}$$

$$(i - \Delta)\sum_{j=1}^{i} c(e_j) \geqslant (i - \Delta + 1)\sum_{j=1}^{i-1} c(e_j), \text{ and since } c(e_{i+1}) \geqslant c(e_i)$$

$$(i - \Delta)\sum_{j=1}^{i+1} c(e_j) \geqslant (i - \Delta + 1)\sum_{j=1}^{i} c(e_j).$$

We use a balanced binary search tree $T_{=w_n}$ to compute efficiently the value of κ. The elements of $E_{=w_n}$ are stored in the leaves of $T_{=w_n}$, maintaining a dictionary order on the costs. Given a node x of $T_{=w_n}$, let $L(x)$ be the set of all elements stored in the leaves of $T_{=w_n}$ that appear before x in an in-order traversal of the tree. Let *next*(x) denote the first leaf of $T_{=w_n}$ that appears after x in an in-order traversal of the tree. The largest set of smallest rate in $E_{=w_n}$ can be found by a search in $T_{=w_n}$ from the root to the leaves as follows:

```
**Algorithm** *descend* ($T_{=w_n}$, $U$)
Let $x$ be the root of $T_{=w_n}$.
**while** $x$ is not a leaf **do**
    **if** $|L(x)| \leqslant \Delta$, or *rate* $(L(x), U) >$ *rate* $(L(next(x)), U)$ **then**
        $x \leftarrow$ right child of $x$
    **else** $x \leftarrow$ left child of $x$ **end if**
**end while**
**if** (cost of the element stored in node $x$) $=$ *rate* $(L(x), U)$ **then**
    Find the rightmost leaf $y$ that stores an element of cost equal to *rate* $(L(x), U)$.
    **return** (set of elements in $L(y)$)
**else return** (set of elements in $L(x)$) **end if**
```

The correctness of this algorithm follows from Lemma 4.2.

Lemma 4.3. *Algorithm descend runs in* $O(\log |T_{=w_n}|)$ *time.*

Proof. Since *rate* $(L(x), U) = c(L(x))/(|L(x)| - \Delta)$, to compute *rate* $(L(x), U)$ we need to know in each iteration of *descend* the number of elements in $L(x)$ and their total cost. This can easily be computed as the algorithm traverses the tree if each internal node x of $T_{=w_n}$ stores the number and total cost of all elements in the subtree rooted at x, and the smallest cost of any element in that subtree. This information is also used by the algorithm to compute *rate* $(L(next(x)), U)$. Since $T_{=w_n}$ is a balanced tree, the algorithm runs in $O(\log |T_{=w_n}|)$ time. □

Let $H_{<w_n}$ be a min-heap that stores the elements of $E_{<w_n}$ maintaining heap order on their costs. With this heap and algorithm *descend* a set of smallest rate in U can be found in $O(\log |T_{=w_n}|)$ time.

We describe now how our data structures have to be updated so that a set of smallest rate in U can be computed efficiently in each iteration of algorithm *uplift*. We use an additional data structure, a list $L_{>w_n}$ containing the elements of $E_{>w_n}$ in non-decreasing order of weight. There are two cases that have to be considered. (1) If a singleton $\{e_s\} \subseteq E_{<w_n}$ is chosen as the subset of smallest rate in U, then *uplift* increases the weight of e_s up to w_n and moves e_s from $H_{<w_n}$ into $T_{=w_n}$. (2) If a set $S \subseteq E_{=w_n}$ is the set of smallest rate in U, then algorithm *uplift* increases the weights of the elements in S to w_g, the smallest element weight larger than w_n. Since then w_g becomes the weight of the nth smallest element in E, all elements in $E_{=w_n} - S$ are moved from $T_{=w_n}$ to $H_{<w_n}$. Also each element of weight w_g in $L_{>w_n}$ is removed form $L_{>w_n}$ and inserted in $T_{=w_n}$.

Lemma 4.4. *The robustness function of a uniform matroid* $U = (E, \mathcal{I}, w, c)$ *can be computed in* $\mathrm{O}(|E| \log |E|)$ *time and using* $\mathrm{O}(|E|)$ *space.*

Proof. We first show that *uplift* performs at most $2|E| - n$ iterations. In each iteration in which *uplift* selects some set $S \subseteq E_{=w_n}$ as the set of smallest rate in U, at least one element is removed from $L_{>w_n}$. This can happen at most $|E| - n$ times, since no element is ever inserted into $L_{>w_n}$ while updating the data structures. In each iteration in which *uplift* chooses a singleton $\{e_s\}$ as the set of smallest *rate*, the element e_s is moved from $H_{<w_n}$ to $T_{=w_n}$, and it remains in $T_{=w_n}$ until the algorithm ends. To show this, suppose that in a later iteration j, element e_s is moved back to $H_{<w_n}$. This means that in iteration j a set $T \subseteq E_{=w_n}$, with $e_s \notin T$, has the smallest rate in U. Since *descend* finds the largest set of smallest rate in $E_{=w_n}$, it follows that $c(e_s) > \mathit{rate}\,(T, U)$. But, this is not possible since the robustness function is non-decreasing. Therefore, the number of iterations in which a singleton is selected as a set of smallest rate is at most $|E|$, and hence the total number of iterations that *uplift* performs is at most $2|E| - n$.

A set S of smallest rate in U can be found in $\mathrm{O}(\log |E|)$ time, and *tolerance* (S, U) can be computed in constant time assuming that the weights of the elements are initially sorted non-decreasingly by weight.

It only remains to bound the time required to update the data structures. Removing the smallest element from $H_{<w_n}$ takes $\mathrm{O}(\log |H_{<w_n}|)$ time, and removing the first element from $L_{>w_n}$ can be done in constant time. Inserting an element into $T_{=w_n}$ takes $\mathrm{O}(\log |T_{=w_n}|)$ time. Since, by the above discussion, any element is removed from $H_{<w_n}$ or $L_{>w_n}$ at most once, and an element can be inserted into $T_{=w_n}$ at most twice, the total time required to update the data structures is $\mathrm{O}(|E| \log |E|)$. $\square$

We now consider the complexity for computing the robustness function of a partition matroid $P = (E, \mathcal{I}, w, c)$ with $\ell \geqslant 1$.

Theorem 4.1. *The robustness function of a partition matroid* $P = (E, \mathcal{I}, w, c)$ *can be computed in* $\mathrm{O}(|E| \log |E|)$ *time and using* $\mathrm{O}(|E|)$ *space.*

Proof. Let $E_1, E_2, \ldots, E_\ell$ be the blocks of E. We have shown above how to compute efficiently a set of smallest rate in each set E_i. To find the set with overall smallest rate in P, we use a heap H in which we store the rate of a set of smallest rate in each E_i, for each $i = 1, 2, \ldots, \ell$.

The arguments used in the proof of Lemma 4.4 can be extended to show that *uplift* performs only $\mathrm{O}(|E|)$ iterations. Hence, *uplift* computes the robustness function of P in $\mathrm{O}(|E| \log |E|)$ time.

The overall space used by our data structures is $\mathrm{O}(|E|)$. $\square$

5. Evaluating the robustness function at one point

In the previous section we presented an algorithm that computes all the breakpoints of the robustness function F_P for a partition matroid P. If we do not want to compute all the breakpoints of F_P, but only wish to evaluate F_P at a specific point b, then we can design an algorithm that requires only linear time.

Consider a partition matroid $P = (E, \mathcal{I}, w, c)$ with blocks $E_1, E_2, \ldots, E_\ell$. Let $P_i = (E_i, \mathcal{I}_i, w, c)$ be the uniform matroid induced by E_i, for $i = 1, 2, \ldots, \ell$. For some given budget b, the value of $F_P(b)$ can be computed by determining the optimal way of distributing the budget among the matroids P_i, and by optimally spending the fractional budget b_i assigned to each P_i increasing the weights of its elements. Therefore, we can write

$$F_P(b) = \max \left\{ \sum_{i=1}^{\ell} F_{P_i}(b_i) | b_i \geqslant 0 \text{ for all } 1 \leqslant i \leqslant \ell \text{ and } \sum_{i=1}^{\ell} b_i = b \right\}. \tag{1}$$

For arbitrary functions F_{P_i}, problem (1) is known as the *optimal distribution of effort problem* [22] or as the *convex knapsack problem* [11]. There are several efficient algorithms to solve these problems [19,11], but only under the assumption that the functions F_{P_i} are given in an explicit form that make it possible to compute in constant time the value of $F_{P_i}(x)$ for any $x \geqslant 0$ and $1 \leqslant i \leqslant \ell$. Since we do not have an explicit representation of the robustness functions F_{P_i}, we do not know how to compute the value of $F_{P_i}(x)$ in constant time, and thus, we have not found efficient implementations of these algorithms for our problem. Instead, we present here a prune-and-search algorithm to compute $F_P(b)$ in $O(m)$ time.

Let $\delta = \delta_1, \delta_2, \ldots, \delta_k$ be a sequence of increases on the weights of the elements of partition matroid P. We say that δ is a *canonical sequence of increases* if each δ_i increases only the weights of the elements in the largest set S of smallest rate in P, and each element in S has its weight increased by the same amount $d_i \leqslant \mathit{tolerance}\,(S, P)$. Algorithm *uplift* can be implemented to determine a canonical sequence of increases, and therefore, a canonical sequence of increases can be used to compute $F_P(b)$ for any $b \geqslant 0$.

The following property of the largest sets of smallest rate in P plays a key role in our algorithm.

Property 5.1. *Let $\delta = \delta_1, \delta_2, \ldots, \delta_k$ be a canonical sequence of increases and S_i be the set chosen by some δ_i, $i < k$. In all subsequent increases δ_j, $j > i$, all elements of S_i will undergo the same weight changes.*

Proof. Let us consider a set S_i with at least 2 elements, e_1 and e_2. Note that the cost of e_1 (e_2) cannot be larger than the rate r_{S_i} of S_i, because otherwise $S_i - \{e_1\}$ ($S_i - \{e_2\}$) would have a smaller rate than S_i. To see this, assume that

$$c(e_1) > \mathit{rate}\,(S_i) \geqslant \frac{c(S_i - \{e_1\}) + c(e_1)}{\mathit{coverage}\,(S_i - \{e_1\}) + 1},$$

then,

$$c(e_1)\, \mathit{coverage}\,(S_i - \{e_1\}) > c(S_i - \{e_1\}).$$

Adding $c(S_i - \{e_1\})\, \mathit{coverage}\,(S_i - \{e_1\})$ to both sides, we get

$$(c(e_1)+c(S_i-\{e_1\}))\mathit{coverage}\,(S_i-\{e_1\}) > c(S_i - \{e_1\})(1+\mathit{coverage}\,(S_i-\{e_1\})).$$

Re-arranging terms we finally get,

$$\mathit{rate}\,(S_i) \geqslant \frac{c(S_i - \{e_1\}) + c(e_1)}{\mathit{coverage}\,(S_i - \{e_1\}) + 1} > \frac{c((S_i - \{e_1\}))}{\mathit{coverage}\,(S_i - \{e_1\})} = \mathit{rate}\,(S_i - \{e_1\}).$$

Suppose that δ_j, $j > i$, selects set $S_j \subseteq E$, with rate r_{S_j}, and that $e_2 \in S_j$ but $e_1 \notin S_j$. This means that $r_{S_j} < c(e_1)$, because otherwise $S_j \cup \{e_1\}$ would be a larger set with rate no larger than r_{S_j}. But, then $r_{S_j} < r_{S_i}$ since $c(e_1) \leqslant r_{S_i}$. This contradicts that the robustness function of P is non-decreasing. □

Property 5.1 can be used to design a slightly more efficient version of *uplift* than that presented in the previous section. The idea is that if in some iteration of *uplift*, set $S \subseteq E$ with $|S| > 1$ is selected as the set of smallest rate, then we can replace S by a single meta-element e_S. We can do this, since in the succeeding iterations *uplift* does not need to keep track of the individual weight changes of the elements in S. Although this modified algorithm is more efficient than the original one, its time complexity is still $O(|E| \log |E|)$. We give below a different approach that exploits Property 5.1 to yield a linear time algorithm for computing $F_P(b)$, for a given $b \geqslant 0$. To introduce our basic strategy, we show first how to compute $F_P(b)$ for a uniform matroid.

5.1. Evaluating the robustness function of a uniform matroid

Let $\delta(w_i)$, for any value $w_i \geqslant 0$ be a canonical sequence of increases for the elements of a partition matroid P such that the weights of the elements are increased as much as possible without increasing the weight of any element above w_i. Let $\bar{\delta}(r_i)$, for any value $r_i \geqslant 0$ be a canonical sequence of increases such that the weights of the elements in P are increased as much as possible without selecting a set of rate at least r_i.

Given a uniform matroid $U = (E, \mathcal{I}, w, c)$, our algorithm for evaluating $F_U(b)$, for some $b \geqslant 0$, does not perform a linear search over the curve F_U, as *uplift* does, but evaluates F_U at some sequence of probe values that converge to the desired one. The algorithm performs two different types of probes, each one implemented by a prune-and-search linear-time routine. The first routine, called *up_to_weight*, takes a weight w_i and computes the weight increases corresponding to $\delta(w_i)$. The routine determines the cost of the weight increases and the rate of a set of smallest rate according to the increased weights. The second routine, called *up_to_rate*, takes a rate r_i and computes the weight increases corresponding to $\bar{\delta}(r_i)$. This routine determines the total cost of the weight increases and the weight of the nth smallest element according to the increased weights.

We present below implementations for these routines. Routine *up_to_weight* receives as arguments a weight w_i, the set of elements E, and the weight and cost functions w and c.

```
Algorithm up_to_weight ($w_i$, $E$, $w$, $c$)
Find the largest set $S$ of smallest rate in $E_{\leqslant w_i}$ assuming that all elements in $E_{\leqslant w_i}$
   have the same weight.
return (rate of $S$, cost of increasing the weights of the elements in $S$ to $w_i$)
```

Lemma 5.1. *Algorithm up_to_weight finds in linear time the cost of the weight increases made by $\delta(w_i)$ and the rate of a set of smallest rate according to the increased weights.*

Proof. We first describe how to compute in linear time the largest set of smallest rate in $E_{\leqslant w_i}$ assuming that all elements in $E_{\leqslant w_i}$ have the same weight. Perform a binary search on the costs of the elements to find the smallest cost c' for which $c(T)/(|T| - |E_{\leqslant w_i}| + n) \leqslant (c(T) + c')/(|T| + 1 - |E_{\leqslant w_i}| + n)$, where $T \subseteq E_{\leqslant w_i}$ is formed by all elements in $E_{\leqslant w_i}$ of cost smaller than c' and $|T| > |E_{\leqslant w_i}| - n$. If $c' = c(T)/(|T| - |E_{\leqslant w_i}| + n)$, then add to T all elements from $E_{\leqslant w_i}$ of cost c'. By Lemma 4.2, T is the desired set. Using the linear-time algorithm of Blum et al. [5], the above binary search can be performed in $O|E_{\leqslant w_i}|)$ time.

To show that algorithm *up_to_weight* is correct we have to consider two different cases. Let w_n be the weight of the nth smallest element in E. If $w_i < w_n$, then only the elements of smallest cost in $E_{<w_n}$ have their weights increased by $\delta(w_i)$. Note that the set S computed by *up_to_weight* contains exactly those elements. For the case when $w_i \geqslant w_n$, suppose that the weights of the elements have been increased by $\delta(w_i)$. Let R be the largest set of smallest rate according to the increased weights, and let r_R be its rate. Observe that all elements in $E_{\leqslant w_i} - R$ have cost no smaller than r_R. Hence, if we assume the same initial weight for all elements in $E_{\leqslant w_i}$, set R would be the largest set of smallest rate in it. □

Algorithm *up_to_rate*, described below, receives as input a rate r, the set of elements E, and the weight and cost functions w and c.

```
Algorithm up_to_rate ($r$, $E$, $w$, $c$)
$T \leftarrow \emptyset$;    $w^* \leftarrow \infty$
repeat
   Find the weight $\bar{w}$ of the $\lceil \frac{1}{2}|E| \rceil$th smallest element in $E$.
   $(r_1, c_1) \leftarrow$ up_to_weight($\bar{w}$, $E$, $w$, $c$)
   if $r_1 \geqslant r$ then
      $E \leftarrow E_{<\bar{w}}$
      $w^* \leftarrow \bar{w}$
   else
      $T \leftarrow T \cup \{e | w(e) \leqslant \bar{w}$ and $c(e) < r\}$
      $E \leftarrow E_{>\bar{w}}$
   end if
until $|E| = 0$
$c_T \leftarrow$ cost of increasing the weights of the elements in $T$ to $w^*$.
$w_n \leftarrow$ weight of the $n$th smallest element in $E$ according to the modified weights
return $(c_T, w_n)$
```

The algorithm performs a binary search on the weights, invoking *up_to_weight* on each probe weight $\bar{w}$. If $\bar{w}$ is too large, *up_to_rate* discards all elements of weight $\bar{w}$ or larger since their weights are not increased by $\bar{\delta}(r)$. If $\bar{w}$ is too small, *up_to_rate* tries a larger probe value, but first it reduces the size of E. Since all elements of cost at least r and weight at most $\bar{w}$ do not have their weights increased by $\bar{\delta}(r)$, they can be ignored. However, all elements of cost smaller than r and weight at most $\bar{w}$ will have their weights increased by $\bar{\delta}(r)$. Instead of keeping track of all the individual weight increases of these elements, *up_to_rate* stores them in a set T. When the algorithm determines the maximum weight increase that those elements should have, it performs the increases in a single step.

The algorithm maintains the invariant that the value of w^* is an upper bound on the maximum weight w_r that $\bar{\bar{\delta}}(r)$ can assign to an element of E without selecting a set of rate at least r. In each iteration of the repeat-loop, either the value of w^* is decreased to $\bar{w}$, or it is discovered that $\bar{w}$ is a lower bound for w_r. The gap between upper and lower bounds for w_r decreases in each iteration of the repeat-loop. When the loop ends, the value of w^* is equal to w_r.

Lemma 5.2. *Algorithm up_to_rate finds in linear time the cost of the weight increases determined by* $\bar{\bar{\delta}}(r)$ *and the weight of the nth smallest element according to the increased weights.*

Proof. By the above discussion, *up_to_rate* correctly computes the weight increases determined by $\bar{\bar{\delta}}(r)$. Each iteration of the repeat-loop takes linear time, and in each iteration the size of E is reduced by at least one half. Therefore, the total time needed by the algorithm is $O(|E|)$. □

We describe now a recursive algorithm for evaluating the robustness function F_U of a uniform matroid $U = (E, \mathcal{I}, w, c)$ at a given budget value $b \geqslant 0$. In each recursive call the algorithm reduces in linear time the number of elements in E by a fraction of at least one third, hence, the overall time complexity of the algorithm is linear in the number of elements. The essential component of each iteration is a pair of tests that allow the algorithm either to find at least one third of the elements in E that have weights or costs that are too large (and, thus, that can be discarded), or to identify at least one third of the elements in E that will end up having the same final weight (and, thus, that can be contracted to a single meta-element). Let w_B be the weight of a minimum weight base of U according to the initial weights. The algorithm receives as arguments the set of elements E, the weight and cost functions, and the budget value b.

```
Algorithm robustness_uniform($E, w, c, b$)
if $|E| = 1$ then
    Let $E = \{e\}$. Set $w(e) \leftarrow w(e) + b/c(e)$.
    Let $w_B^*$ be the weight of a minimum weight base of $U$ according to the increased weights.
    return $(w_B^* - w_B)$
else
    Compute $w_t$, the weight of the $\lceil \frac{2}{3}|E| \rceil$th smallest element in $E$.
    $(c_t, r_t) \leftarrow$ up_to_weight($w_t, E, w, c$)
    if $c_t \geqslant b$ then
        $E \leftarrow E - \{e | w(e) \geqslant w_t\}$
        return (robustness_uniform($E, w, c, b$))
    else
        Compute $\bar{c}$, the upper median cost among the elements $e \in E$ of weight $w(e) \leqslant w_t$.
        $(\tilde{c}, \tilde{w}_n) \leftarrow$ up_to_rate ($\bar{c}, E, w, c$)
        if $\tilde{c} \geqslant b$ then
            $E \leftarrow E - \{e | w(e) \leqslant \tilde{w}_n$ and $c(e) \geqslant \bar{c}\}$
            return (robustness_uniform($E, w, c, b$))
        else
            $(E, w, b) \leftarrow$ contract_uniform ($E, w, c, b, \tilde{c}, \tilde{w}_n, \bar{c}$)
            if $b = 0$ then
                Let $w_B^*$ be the weight of a minimum weight base of $U$ according to the increased
                weights.
                return $(w_B^* - w_B)$
            else return (robustness_uniform($E, w, c, b$)) end if
        end if
    end if
end if
```

Function *contract_uniform* identifies a set $S \cup T \subseteq E$ of size at least $\lceil |E|/3 \rceil$ formed by elements that will experience exactly the same weight increases in the computation of $F_U(b)$. Instead of keeping track of the individual weight changes of these elements, they are contracted to a single meta-element $\hat{e}$ and the algorithm computes only the weight increases for $\hat{e}$. After the algorithm has computed the final weight increases, it is easy to expand the meta-elements to determine the final weight for each element in E.

```
Algorithm contract_uniform($E, w, c, b, \tilde{c}, \tilde{w}_n, \bar{c}$)
Let $S = \{e | w(e) \leqslant \tilde{w}_n$ and $c(e) < \bar{c}\}$, and $T = \{e | w(e) \leqslant \tilde{w}_n$ and $c(e) = \bar{c}\}$.
for each $e \in S$ do $w(e) \leftarrow \tilde{w}_n$ end for
$b \leftarrow b - \tilde{c}$
for each $e \in T$ do
    $(w(e),\ b) \leftarrow (\min\{\tilde{w}_n, w(e) + b/c(e)\},\ \max\{0, b - (\tilde{w}_n - w(e)) * c(e)\})$
    if $b = 0$ then exit the for-loop end if
end for
if $b > 0$ then
    $E \leftarrow (E - S - T) \cup \{\hat{e}\}$, where $\hat{e}$ is meta-element with $w(\hat{e}) = \tilde{w}_n$ and $c(\hat{e}) = c(S \cup T)$.
end if
return $(E, w, b)$
```

Observe that every call that algorithm *robustness_uniform* makes to *contract_uniform* is preceded by a call to *up_to_rate*, which determines the cost $\tilde{c}$ of optimally increasing the weights of the elements in E to the point at which the following weight increase would be over a set of rate at least $\bar{c}$. Algorithm *contract_uniform* continues these optimal weight increases by lifting to $\tilde{w}_n$ the weights of the elements in S and T. If the budget is exhausted while performing these increases, then the algorithm stops since it has computed the weight function needed to determine $F_U(b)$. Otherwise, the budget is decreased to account for the new weight increases.

If any budget remains after increasing the weights of the elements in $S \cup T$, then all elements of weight at most $\tilde{w}_n$ and cost at most $\bar{c}$ are contracted. Note that these elements belong to the first set formed by elements of weight $\tilde{w}_n$ that a canonical sequence of increases would select, and thus, by Property 5.1, they can be replaced by a meta-element.

To illustrate how algorithm *robustness_uniform* works, consider the following example. Let $U = (E, \mathcal{I}, w, c)$ be a uniform matroid of rank 4 with set $E = \{a, b, c, d, e, f, g\}$. The initial weights of the elements are 1, 1, 2, 3, 4, 5, and 6, respectively, and their costs are 1, 3, 5, 1, 3, 1, and 3. The value of w_B is $1 + 1 + 2 + 3 = 7$. We wish to evaluate $F_U(9)$. The algorithm first computes $w_t = 4$, and invokes routine *up_to_weight*. This routine returns $(c_t, r_t) = (4, 2)$. The value of c_t is the cost of increasing the weights of the elements to 4, 1, 2, 4, 4, 5, and 6, respectively; with these weights the next set of smallest rate is $\{a, d\}$ and it has rate $r_t = 2$. Since $c_t < 9$, *robustness_uniform* computes $\bar{c} = 3$, and invokes routine *up_to_rate*. This routine returns $(\tilde{c}, \tilde{w}_n) = (6, 5)$. The value of $\tilde{c}$ is the cost of increasing the weights of the elements to 5, 1, 2, 5, 4, 5, and 6. Since $\tilde{c} < 9$, then *contract_uniform* is invoked, and it exhausts the budget by increasing the weights of the elements to 5, 2, 2, 5, 4, 5, and 6. Since upon return from *contract_uniform* b is zero, *robustness_uniform* outputs $F_U(9) = (2 + 2 + 4 + 5) - 7 = 6$.

Lemma 5.3. *Algorithm robustness_uniform computes $F_U(b)$, for any given $b \geqslant 0$, in $O(|E|)$ time.*

Proof. Each call to algorithm *robustness_uniform* takes linear time, and, as we show below, reduces the size of E by at least a fraction of one third. Therefore, the overall time complexity is linear on the number of elements.

In each recursive call, *robustness_uniform* invokes routine *up_to_weight*, which computes the weight increases of canonical sequence of increases $\delta(w_t)$. If the cost of these weight increases exceeds b, then all elements with weight at least w_t are discarded from E. There are at least $\lceil \frac{1}{3}|E| \rceil$ of these elements, and so, in this case the size of E is reduced to at most $\lfloor \frac{2}{3}|E| \rfloor$.

If the cost of the weight increases is smaller than b, then *robustness_uniform* invokes *up_to_rate* to find the weight increases defined by $\bar{\delta}(\bar{c})$. If the cost of these new increases surpasses the budget, then all elements of weight at most $\tilde{w}_n$ and cost at least $\bar{c}$ are discarded. Since $\tilde{w}_n \geqslant w_t$, this step discards at least $\lceil \frac{1}{3}|E| \rceil$ elements from E. However, if the cost of the last weight increases is smaller than b, then *robustness_uniform* invokes routine *contract_uniform*. This routine either makes optimal weight increases that exhaust the budget, or it replaces all elements of weight at most $\tilde{w}_n$ and cost at most $\bar{c}$ by a meta-element. These elements represent at least one third of E. Note that if $|E| < 4$ the size of E is still reduced by a fraction of at least one third, even when a meta-element is added to E. The reason for this is that either $\lceil \frac{2}{3}|E| \rceil$ rounds up (in the computation of w_t), or $\lceil \frac{1}{2} \lceil \frac{2}{3}|E| \rceil \rceil$ rounds up (in the computation of $\bar{c}$). $\square$

5.2. Evaluating the robustness function of a partition matroid

We turn our attention now to the problem of evaluating in linear time $F_P(b)$ for a partition matroid P with ℓ blocks, and $\ell > 1$. This problem is more difficult than for the case when $\ell = 1$ since we have to determine simultaneously how the weights of the elements change in all the blocks E_i. As for the case of $\ell = 1$, we compute $F_P(b)$ by a recursive prune-and-search process that combines searches on weights with searches on costs. However, our new algorithm reduces the size E by a fraction of only one tenth in each recursive call. This decrease in performance, compared to *robustness_uniform*, is due to the additional difficulty that multiple blocks E_i impose on finding good probe values. The algorithm uses an array *upper* of size ℓ, and it stores in *upper*(i) an upper bound on the maximum weight that can be assigned to any element in E_i. Each entry of *upper* is initialized to ∞. Let w_B be the weight of a minimum weight base of P according to the initial weights. The algorithm is the following.

```
Algorithm robustness_partition($E, w, c, b, upper$)
if $|E| = 1$ then
    Let $E = \{e\}$. Set $w(e) \leftarrow w(e) + b/c(e)$.
    Let $w_B^*$ be the weight of a minimum weight base of $P$ according to the increased weights.
    return $(w_B^* - w_B)$
else
    for $i = 1, 2, \ldots, \ell$ do
        Compute $w_i$, the weight of the $\lceil \frac{4}{5}|E_i| \rceil$th smallest element in $E_i$.
        $(c_i, r_i) \leftarrow$ up_to_weight $(w_i, E_i, w, c)$
    end for
    Compute $r'$, the weighted median of the rates $r_i$ using, for each $i$, $|E_i|$ as the weight for $r_i$.
    for $i = 1, 2, \ldots, \ell$ do $(c_i', w_i') \leftarrow$ up_to_rate $(r', E_i, w, c)$ end for
    if $\sum_{i=1}^{\ell} c_i' \geqslant b$ then
        for $i = 1, 2, \ldots, \ell$ do
            $E_i \leftarrow E_i - \{e | w(e) \geqslant w_i'\}$
            upper$(i) \leftarrow w_i'$
        end for
        return (robustness_partition($E, w, c, b, upper$))
    else
        Let $S = \cup_{\{i | r_i \leqslant r'\}} \{e | e \in E_i$ and $w(e) < w_i'\}$
        Compute $\bar{c}$, the cost of the $\lceil \frac{3}{4}|S| \rceil$th smallest cost element in $S$.
        for $i = 1, 2, \ldots, \ell$ do $(\tilde{c}_i, \tilde{w}_{n_i}) \leftarrow$ up_to_rate $(\bar{c}, E_i, w, c)$ end for
        if $\sum_{i=1}^{\ell} \tilde{c}_i \geqslant b$ then
            for $i = 1, 2, \ldots, \ell$ do $E_i \leftarrow E_i - \{e | w(e) \leqslant \tilde{w}_{n_i}$ and $c(e) \geqslant \bar{c}\}$ end for
            return (robustness_partition($E, w, c, b, upper$))
        else
            $(E, w, b) \leftarrow$ contract_partition $(E, w, c, b, \sum_{i=1}^{\ell} \tilde{c}_i, \{\tilde{w}_{n_i}, \ldots, \tilde{w}_{n_\ell}\}, \bar{c}, upper)$
            if $b = 0$ then
                Let $w_B^*$ be the weight of a minimum weight base of $P$ according to the increased
                weights.
                return $(w_B^* - w_B)$
            else return (robustness_uniform($E, w, c, b, upper$)) end if
        end if
    end if
end if
```

Note the correspondence between the structure of *robustness_uniform* and the structure of *robustness_partition*. The part of *robustness_partition* preceding the test "$\sum_{i=1}^{\ell} c_i' \geqslant b$" is more complex than the corresponding part of algorithm *robustness_uniform*. The reason is that *robustness_partition* has to consider all blocks E_i, and the weights of the elements in all the blocks are not increased at the same rate. This makes the computation of good probe values more difficult than for the case of a uniform matroid. Note also that the probe values w_i and $\bar{c}$ are different from the corresponding probe values chosen by *robustness_uniform*. These values were selected to ensure that each call to *robustness_partition* decreases the size of E by a fixed fraction. Algorithm *contract_partition* is described below.

```
Algorithm contract_partition($E, w, c, b, \tilde{c}, \{\tilde{w}_{n_1}, \ldots, \tilde{w}_{n_\ell}\}, \bar{c}, upper$)
Let $S_i = \{e | e \in E_i, w(e) \leqslant \tilde{w}_{n_i}$ and $c(e) < \bar{c}\}$, and
    $T_i = \{e | e \in E_i, w(e) < \tilde{w}_{n_i}$ and $c(e) = \bar{c}\}$, for all $i = 1, 2, \ldots, \ell$.
for $i = 1, 2, \ldots, \ell$ do
    Increase to $\tilde{w}_{n_i}$ the weight of every element in $S_i$.
end for
$b \leftarrow b - \tilde{c}$
for $i = 1, 2, \ldots, \ell$ do
    if $|E_i| > 1$ then
        for every $e \in T_i$ do
            $(w(e),\ b) \leftarrow (\min\{\tilde{w}_{n_i}, w(e) + b/c(e)\},\ \max\{0, b - (\tilde{w}_{n_i} - w(e)) * c(e)\})$
            if $b = 0$ then exit the inner for-loop end if
        end for
        if $b = 0$ then exit the for-loop
        else $E_i \leftarrow (E_i - S_i - T_i) \cup \{\hat{e}\}$, where meta-element $\hat{e}$ has $w(\hat{e}) = \tilde{w}_{n_i}$,
                                        and $c(\hat{e}) = c(S_i \cup T_i)$.
        end if
    else
        Let $E_i = \{e\}$.
        if $c(e) \leqslant \bar{c}$ then
            $(w(e),\ b) \leftarrow (\min\{upper(i), w(e) + b/c(e)\},\ \max\{0, b - (upper(i) - w(e)) * c(e)\})$
            if $b = 0$ then exit the for-loop
            else $E \leftarrow E - E_i$ end if
        end if
    end if
end for
return $(E, w, b)$
```

Algorithm *contract_partition* is similar to *contract_uniform*, but it has to deal with one situation that does not appear for the case of uniform matroids. If any one of the sets E_i has only one element, then it cannot be further contracted. In this case, *contract_partition* does the following. If the unique element $e \in E_i$ has cost at most $\bar{c}$, then *contract_partition* increases its weight to $upper(i)$ if the budget is large enough and then it discards block E_i. This can be done, since the weight of e cannot be increased above $upper(i)$, and when it reaches such weight *robustness_partition* does not have to consider it any more. But, if the remaining budget is too small to perform the weight increase, then the weight of e is increased only as much as the budget allows. Since the budget is exhausted, no more weight increases are possible.

Theorem 5.1. *Given a partition matroid $P = (E, \mathcal{I}, w, c)$ and a positive budget b, the value of $F_P(b)$ can be computed in* $O(|E|)$ *time.*

Proof. To show that the algorithm runs in $O(|E|)$ time, it suffices to show that each iteration of the while-loop reduces the size of E by at least $\frac{1}{10}|E|$. The value r' computed by *robustness_partition* is such that the weight increases defined by $\bar{\delta}(r')$ do not affect the weights of at least $\frac{1}{2} * (1 - \frac{4}{5})|E| = \frac{1}{10}|E|$ elements of E. Hence, if the cost of these weight increases exceeds b, *robustness_partition* discards those elements and reduces the size of E by at least $\lceil \frac{1}{10}|E| \rceil$. If the cost of the weight increases is smaller than b, *robustness_partition* makes a test on $\bar{c}$, the $\lceil \frac{3}{4}|S| \rceil$ smallest element cost in S. It is not difficult to see that $|S| \geqslant \frac{2}{5}|E|$. If the cost of the new weight increases is at least b, then *robustness_partition* discards at least $\lceil \frac{1}{4}|S| \rceil \geqslant \lceil \frac{1}{10}|E| \rceil$ elements form E.

If $\sum_{i=1}^{\ell} \tilde{c}_i < b$, then *contract_partition* contracts in each E_i all elements from S of cost at most $\bar{c}$. Let $S_i = E_i \cap S$. There is one situation in which the algorithm cannot contract the elements in S_i of cost at most $\bar{c}$. Suppose that set S_i has only two elements, one of cost at most $\bar{c}$ and the other of cost larger than $\bar{c}$, then *contract_partition* cannot reduce the size of E_i (since it would try to contract the element of cost at most $\bar{c}$ to a meta-element). Since $\bar{c}$ is the $\lceil \frac{3}{4}|S| \rceil$th smallest element cost, then there are at most $\lfloor \frac{1}{4}|S| \rfloor$ elements of cost larger than $\bar{c}$. Hence, there are at most $\lfloor \frac{1}{4}|S| \rfloor$ sets S_i of two elements for which the algorithm cannot contract their sizes as described above. These sets include at most $\lfloor \frac{1}{2}|S| \rfloor$ elements. The other $\lceil \frac{1}{2}|S| \rceil$ elements must belong to sets S_j that *contract_partition* contracts to at most half of their sizes. Therefore, *contract_partition* contracts the size of E by at least $\lceil \frac{1}{4}|S| \rceil \geqslant \lceil |E|/10 \rceil$ elements. $\square$

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] R.K. Ahuja, J.B. Orlin, A capacity scaling algorithm for the constrained maximum flow problem, Networks 25 (1995) 89–98.

[3] R.K. Ahuja, J.B. Orlin, C. Stein, R.E. Tarjan, Improved algorithms for bipartite network flow, SIAM J. Comput. 23 (1994) 906–933.

[4] O. Berman, D.I. Ingco, A.R. Odoni, Improving the location of minimum facilities through network modification, Ann. Oper. Res. 40 (1992) 1–16.

[5] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. System Sci. 7 (1973) 448–461.

[6] R.E. Burkard, B. Klinz, J. Zhang, Bottleneck capacity expansion problems with general budget constraints, RAIRO Oper. Res. 35 (2001) 1–20.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to algorithms, The MIT Press, Cambridge, MA, 1992.

[9] K.U. Drangmeister, S.O. Krumke, M.V. Marathe, H. Noltemeier, S.S. Ravi, Modifying edges of a network to obtain short subgraphs, Theoret. Comput. Sci. 203 (1998) 91–121.

[10] M.J. Eisner, D.G. Severance, Mathematical techniques for efficient record segmentation in large shared databases, J. ACM 23 (1976) 619–635.

[11] A. Federgruen, P. Zipkin, Solution techniques for some allocation problems, Math. Programm. 25 (1983) 13–24.

[12] G.N. Frederickson, Scheduling unit-time tasks with integer release times and deadlines, Inform. Process. Lett. 16 (1983) 171–173.

[13] G.N. Frederickson, R. Solis-Oba, Increasing the weight of minimum spanning trees, in: Proc. Seventh Annual ACM-SIAM Symp. on Discrete Algorithms, 1996, pp. 539–546.

[14] G.N. Frederickson, R. Solis-Oba, Efficient algorithms for robustness in matroid optimization, in: Proc. Eighth Annual ACM-SIAM Symp. on Discrete Algorithms, 1997, pp. 659–668.

[15] D.R. Fulkerson, G.C. Harding, Maximizing the minimum source-sink path subject to a budget constraint, Math. Programm. 13 (1977) 116–118.

[16] H.N. Gabow, R.E. Tarjan, Efficient algorithms for a family of matroid intersection problems, J. Algorithms 5 (1984) 80–131.

[17] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 209–221.

[18] D. Gusfield, L. Wang, P. Stelling, Graph traversals, genes and matroids: an efficient special case of the traveling salesman problem, Tech. Report No. CSE-96-3, University of California, Davis, 1996.

[19] T. Ibaraki, N. Katoh, Resource Allocation Problems: Algorithmic Approaches, MIT Press, Cambridge MA, 1988.

[20] J.R. Jackson, Scheduling a production line to minimize maximum tardiness, Res. Report No. 43, Management Science Research Project, University of California, Los Angeles, 1955.

[21] A. Juttner, On budgeted optimization problems. Third Hungarian–Japanese Symp. on Discrete Mathematics and its Applications, 2003.

[22] W. Karush, A general algorithm for the optimal distribution of effort, Mgmt. Sci. 9 (1962) 50–72.

[23] D.G. Luenberger, Linear and Nonlinear Programming, Addison-Wesley, Reading, MA, 1984.

[24] M. Mansour, S. Balemi, W. Troul (Eds.), Robustness of Dynamic Systems with Parameter Uncertainties, Birkhauser, Basel, 1992.

[25] C.A. Phillips, The network inhibition problem, in: Proc. 25th Annual ACM Symp. on Theory of Computing, 1993, pp. 776–785.

[26] H.S. Stone, Critical load factors in two-processor distributed systems, IEEE Trans. Software Engrg. 4 (1978) 254–258.

[27] D.J.A. Welsh, Matroid Theory, Academic Press, London, 1976.