

A ONE-WAY ARRAY ALGORITHM FOR MATROID SCHEDULING *

MATTHIAS F.M. STALLMANN†

Abstract. *The greedy algorithm is a standard paradigm for solving matroid optimization problems on sequential computers. This paper presents a greedy algorithm suitable for a fully-pipelined linear array of processors, a generalization of Huang's algorithm [Hua90] for minimum spanning trees. Application of the algorithm to uniprocessor scheduling with release times and deadlines is discussed in detail. A key feature of the algorithm is its use of matroid contraction.*

1. Introduction. Algorithms on one-dimensional arrays of processors with one-way data flow (*one-way arrays*) have several practical advantages. They are simple, fault-tolerant (can work correctly when faulty processors are bypassed), and allow for pipelining of successive problem instances. They are also intriguing from a theoretical point of view. Several combinatorial problems that have efficient sequential algorithms also have one-way array algorithms. These include connected components [AS85, TM83], biconnected components [SSP90], and minimum spanning trees [Hua90]. These algorithms are typically more elegant than their sequential counterparts, and require nontrivial insights. The purpose of this work is to propose a paradigm for solving matroid problems on one-way arrays, generalizing the algorithm of Huang [Hua90]. The paradigm yields an efficient one-way array algorithm for the scheduling problem described in the following paragraph, and suggests future work on one-way array algorithms for matroid problems.

The *deadline scheduling problem* is defined as follows. Given a collection of m tasks, where each task i has an integer deadline d_i and a profit w_i , find a one-processor schedule, i.e. assign each task i an integer *time slot* t_i , corresponding to the interval $(t_i - 1, t_i]$, so that the total profit of tasks assigned before their deadline, $\sum_{\{i|t_i \leq d_i\}} w_i$, is maximized. Note that it suffices to schedule the (max profit) set of tasks that can all be scheduled before their deadlines. The remaining tasks can be scheduled arbitrarily late (or not at all). The best sequential time bound for deadline scheduling, $O(m + n \log n)$, is due to Gabow and Tarjan [GT84].

The one-way array algorithm also extends to the scheduling problem if release times as well as deadlines are considered, as long as the release times and deadlines obey a monotonicity restriction. Each task has a release time r_i in addition to its deadline and profit, and task i must be assigned to time slot t_i with $r_i < t_i \leq d_i$ to earn profit. The *monotone scheduling problem* requires that any two tasks i and j having $r_i < r_j$ also have $d_i \leq d_j$. Monotone scheduling is a generalization of deadline scheduling. Gabow and Tarjan [GT84] give a sequential time bound of $O(m \log m + n^2)$ for scheduling with release times and deadlines (without the monotonicity restriction).

The tasks in a one-processor scheduling problem with release times and deadlines

* in *Proceedings of the Third Symposium on Parallel Algorithms and Architectures*, July, 1991, pages 349 – 356.

† Please direct correspondence to Matthias Stallmann, Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, or e-mail: matt@euler.csc.ncsu.edu

define a matroid in which a set of tasks is independent if all may be scheduled after their release times and before their deadlines. A *matroid* $\mathcal{M} = (E, \mathcal{I})$ is defined on a set of elements E , where $\mathcal{I} \subseteq 2^E$ denotes the class of *independent* subsets of E . This class obeys two axioms: (1) $A \in \mathcal{I}$ and $B \subseteq A$ implies $B \in \mathcal{I}$ (a subset of an independent set is independent), and (2) $A \in \mathcal{I}$ and $B \in \mathcal{I}$ with $|B| > |A|$ implies $\exists e \in B - A$ such that $A \cup \{e\} \in \mathcal{I}$ (an independent set can always be increased in size as long as a larger independent set exists). Classic examples of matroids include the set of columns of a matrix, where a subset is independent if the columns are linearly independent, and the set of edges in a graph, where a subset is independent if it induces no cycles.

Matroids are useful in the study of algorithmic paradigms. Suppose that each element of a matroid is assigned a weight and the objective is to find the *base* (maximum cardinality independent set), having maximum weight. Call this a *matroid optimization problem*. Specific instances of matroid optimization include finding a maximum (or minimum) weight spanning tree and finding a maximum profit schedule in any of the scheduling problems discussed previously. It is not hard to show that any problem of this general form can be solved by the *greedy algorithm*, which repeatedly adds the largest weight element that can be added without violating independence (see, e.g. [Law76]).

This paper discusses a variant of the greedy algorithm for a linear array of processors with one-way data flow. A generic linear-array algorithm for matroid optimization is presented, along with an efficient implementations for the deadline and monotone scheduling. The generic algorithm simplifies and generalizes an algorithm for the minimum spanning tree problem due to Huang [Hua90].

The one-way array algorithm has several advantages over the sequential greedy algorithm. First, it is an on-line algorithm, able to deal with matroid elements presented to it one at a time, in random order. The standard greedy algorithm either requires the matroid elements to be stored internally or to be sorted by decreasing weight prior to execution. If m denote $|E|$ and $n = r(E)$ denote the *rank* of E , the maximum cardinality of an independent subset of E . The sequential greedy algorithm requires $\Theta(m)$ storage locations capable of holding a matroid element, while an on-line algorithm only requires $\Theta(n)$ locations of random-access memory. Note that $n \leq m$ and m can be $\Omega(n^2)$ for spanning tree or monotone scheduling matroids (m can be arbitrarily large in relation to n if redundant elements are allowed; for general matroids m can be exponential in n or worse even without redundant elements).

On-line algorithms on the RAM model exist for both the minimum spanning tree problem (using dynamic trees [ST83]), and the deadline scheduling problem (using a data structure described in [GT84]). In each case, the *period* (time between processing successive inputs) is $\Theta(\log n)$. The one-way array algorithm requires the same area (in this case n processors, each capable of storing a constant number of matroid elements) and has a constant period for both problems.

Another advantage of a one-way array algorithm is that it is pipelineable, namely that successive instances of the same problem can be pushed through the array without an initialization delay between instances. Finally, a one-way array algorithm is fault-

tolerant in the sense that, if faulty processors can be bypassed, the algorithm will work without modification as long as there are sufficient non-faulty processors. Both these advantages hinge upon the unidirectional dataflow of the one-way array model.

The model of computation is a one-way linear array consisting of n processors (*cells*), numbered from 1 to n (recall that n is the rank of the matroid). During a single *time unit* cell i (for $i = 2, \dots, n - 1$) reads input from its left neighbor (cell $i - 1$), performs a fixed number of arithmetic operations, and generates output for its right neighbor (cell $i + 1$) to be read during the next time unit. Cells 1 and n are special cases in that cell 1 reads input from an external host instead of its left neighbor and cell n produces output for the external host instead of its right neighbor. It is assumed that each cell has a fixed number of registers, with each register capable of storing a matroid element. We also assume that data in each cell is initialized to some dummy value. A cell does not have to know its position in the array.

Input to the array consists of a stream of matroid optimization instances, consecutive instances separated by a marker, say \$. Each instance consists of a list of matroid elements in random order. Output is a stream of optimum (maximum weight) bases for the instances. Since n_j , the rank of the j th instance, is, in general, less than or equal to m_j , the number of elements in the j th instance, there are gaps long enough to compensate for this difference between output instances.

We conclude this section with some additional matroid terminology. Section 2 then describes the generic one-way array algorithm. Section 3 gives details and correctness arguments for deadline and monotone scheduling. Section 4 describes further details of the array implementation, such as the output of the optimum base. Section 5 presents some conclusions and open problems.

A *circuit* is a minimal set that is not independent. In the spanning-tree matroid a circuit corresponds to the edges of a cycle. In scheduling, a circuit is a minimal set of $d - r + 1$ tasks all having deadlines $\leq d$ and release times $\geq r$. A matroid element e is called a *loop* if $\{e\}$ is a circuit. If $\mathcal{M} = (E, \mathcal{I})$ is a matroid, then \mathcal{M}/A , \mathcal{M} *contracted* wrt A , is the matroid $\mathcal{M}' = (E - A, \mathcal{I}')$, where $B \in \mathcal{I}'$ iff $B \cup A \in \mathcal{I}$. In the spanning tree matroid, the contracted matroid wrt a set of edges A is obtained by identifying the endpoints of all edges in A . Contraction in the scheduling matroid has a more complicated description, which is the key to understanding the array algorithm for this special case. If A is an independent set and $A \cup \{f\}$ contains a circuit C , while $A \cup \{f\} - \{g\}$ is independent ($g \in C$), then f, g is called a *swap* wrt A .

2. The Generic Algorithm. Some simplifications are used to describe the array algorithm. First note, as was observed by Savage et al. [SSP90], that the actions of a one-way array can be described by tracing the actions of each individual input record as it propagates through the array. That is, a generic algorithm for a one-way array can be described as

for $i := 1$ **to** n **do**
 $(r, c_i) := F(r, c_i)$

where r is the new input record, c_i denotes the contents of cell i , and F is any

function that can be computed during a time unit, namely by a constant number of arithmetic operations or comparisons. This is true because the current contents of c_i only depend on the initial state and previous input records. Second, assume that each cell has the same initial value (a dummy record) before computation begins. This could be arranged by having a start marker propagate through the array before the first input record (see [SSK87]).

Subsequent description of the algorithm ignores any complications having to do with output. The generic algorithm only ensures that after all elements of the j th instance have been processed, the elements of the j th optimum base are stored in cells $1, \dots, n_j$ (the remaining $n - n_j$ cells, if any, will contain dummy records). The solution (optimum base) can then be “pushed” out using techniques described in [SSP90, SSK87]. A more detailed description of how this is accomplished, and how the tasks of the optimum schedule in the scheduling problem can be tagged with their time slot assignments, is given in Section 4.

The generic one-way-array algorithm for matroid optimization problems maintains the following invariant: If \hat{E} is the set of elements that have entered the array so far, the first $r(\hat{E})$ cells contain a maximum-weight independent subset B of \hat{E} , and cell i holds the element of B with i th largest weight. Initially all cells contain dummy elements with weight $-\infty$, and, if $r(\hat{E}) < n$, dummy elements occupy the last $n - r$ cells. Suppose e_1, e_2, \dots, e_n are the elements currently stored in the array (with $w(e_1) \geq w(e_2) \geq \dots \geq w(e_n)$) and let f be the new element to enter the array. The goal is to compute B' , the largest weight independent subset of $\hat{E} \cup \{f\}$. Let index p be such that $w(f) \leq w(e_p)$ and $w(f) > w(e_{p+1})$, i.e. f 's rightful position in the sorted order of the elements is between e_p and e_{p+1} .

If $\{e_1, \dots, e_p, f\}$ is not independent (f is in some circuit of $\{e_1, \dots, e_p, f\}$), then B' does not include f . On the other hand if $\{e_1, \dots, e_p, f\}$ is independent, f is added to B' and e_q is deleted (f, e_q is a swap wrt B'), where $q = \min\{i \mid \{e_1, \dots, e_i, f\} \notin \mathcal{I}\}$. If $B \cup f$ is independent then e_q is the first dummy element — here and in what follows, we pretend that dummy elements extend an independent set so that it becomes a base. The contents of the array then become $e_1, \dots, e_p, f, e_{p+1}, \dots, e_{q-1}, e_{q+1}, \dots, e_n$. Note that, due to sorting by weight, e_q is the least-weight element in the circuit containing f . Thus the swap f, e_q yields the best possible improvement when f is added. The following lemma ensures that the deleted element (either f or e_q) need never be considered for inclusion in any future base.

LEMMA 2.1. *If f_2, g_2 is a swap wrt $B \cup \{f_1\} - \{g_1\}$ (where f_1, g_1 is a swap wrt B) but not wrt B , then both f_1, g_2 and f_2, g_1 are swaps wrt B .*

Proof. See [FS87]. \square

Suppose f (or e_q) is deleted and let B' be the earliest base (after the deletion) for which f, e is a swap wrt B' having $w(f) > w(e)$. Let $B' = B \cup \{g\} - \{h\}$, where B is the base preceding B' . Thus $w(g) > w(h)$. From Lemma 2.1 we know that g, e and f, h are also swaps wrt B . We also know $w(e) \geq w(h)$, otherwise e would have been deleted in place of h . But then f, h is a swap wrt B having $w(f) > w(e) \geq w(h)$, contradicting the choice of B' .

The index p can be easily computed as f traverses the array. To determine whether or not f is to be inserted, and the identity of e_q , the algorithm requires an efficient independence oracle, one for which independence can be detected on the fly, by inspecting one element at a time. The key to the scheduling algorithm (and Huang’s spanning tree algorithm) is the maintenance of an efficient representation of f' , the element f in the matroid $\mathcal{M}/\{e_1, \dots, e_i\}$. Note that f' is a loop iff f is contained in a circuit of $\{e_1, \dots, e_i\}$ in \mathcal{M} .

The representation of f' is computed by the noncommutative binary operation $/$, applied to (representations of) individual matroid elements (e/f is referred to as e contracted wrt f). The i th cell in the array stores e'_i , where $e'_1 = e_1$ and $e'_i = (\dots((e_i/e'_1)/e'_2)/\dots e'_{i-1})$. In the spanning tree case, an element (edge) e is initially represented by its two endpoints, lower-numbered vertex first (vertices are assumed to be integers in the range $1, \dots, 2n$ — there are at most $2n$ vertices in a spanning forest having n edges). The operation $(i', j') = (i, j)/(k, l)$ is defined by $i' = k$ if $i = l$, and $i' = i$ otherwise, and similarly for j' . In other words, the effect is as if k and l are contracted to a single vertex whose label is k .

The algorithm relies on three properties of $/$, as follows.

1. *Order invariance*: $(e/a)/(b/a) = (e/b)/(a/b)$. This is used to argue (by a straightforward induction) that the representation e'_i depends only on the set of elements $\{e_1, \dots, e_{i-1}\}$, not the order in which they appear. If f is inserted into the array, representations $e'_{p+1}, \dots, e'_{q-1}$ can be updated by contracting them wrt f' . Henceforth, we may use f/A to denote $(\dots((f/e'_1)/e'_2)/\dots e'_k)$, where $A = \{e_1, \dots, e_k\}$ and e'_i is defined as above.
2. *Faithfulness*: f is in a circuit of $A \cup f$ iff f/A represents a loop. This property ensures that the decision whether not to insert f can be made correctly and that e_q can be correctly identified.
3. *Swap invariance*: if f, g is a swap wrt A then $e/(A \cup \{f\} - \{g\}) = e/A$. This ensures that e'_{q+1}, \dots, e'_n do not need to be updated when f replaces e_q .

The array algorithm can now be described as follows (i is used to indicate the current cell as the new record containing f propagates through the array):

Algorithm Array-Greedy

```

i := 1; f' := f
while  $w(f) \leq w(e_i)$  do
    f' := f'/e'i; i := i + 1 end do

/* i = p + 1 at this point */
if f' is a loop then exit
    (a dummy record traverses the remaining cells)

/* f' is inserted and  $e'_{p+1}, \dots, e'_{q-1}$  are pushed one
   cell to the right (note: this requires two records,
   f' and previous, to be propagated to the next
   cell each time unit) */

previous := f'
repeat
    /* update representations of both  $e'_i$  and f' */
    newf := f'/e'i;    $e'_i := e'_i / f'$ ;   f' := newf

    /* store  $e'_{i-1}$  (or f') here; send  $e'_i$  to next cell */
    swap  $e'_i$  and previous
until previous is a loop

/* here i = q + 1 and subsequent cells of the array
   remain unaltered */

```

In case of the spanning tree algorithm, it is straightforward to show that the contraction operation is order invariant, and that $(i, j)/A$, where A is a set of edges, is just $(s_A(i), s_A(j))$, where $s_A(x)$ is the lowest numbered vertex of x 's connected component in the subgraph defined by A . Faithfulness and swap invariance follow immediately. These observations give a simplified proof of correctness for Huang's algorithm.

3. The Scheduling Algorithm. The same generic algorithm can be used for the monotone scheduling problem (with deadline scheduling being a special case). Note that a task i is characterized by the pair (r_i, d_i) (weights can be ignored in defining contraction). The contraction $(r', d') = (r, d)/(r_0, d_0)$ is defined as follows.

$$(1) \quad (r', d') = \begin{cases} (r - 1, d - 1) & \text{if } r_0 < r, d_0 \leq d \\ (r, d - 1) & \text{if } r_0 \geq r, d_0 \leq d \\ (r, d) & \text{if } d_0 > d \end{cases}$$

A loop in this context is a task (r, d) with $d - r = 0$.

LEMMA 3.1. *The contraction operation defined in (1) is order invariant.*

Proof. Let (r', d') denote the result of two contractions $((r, d)/(r_1, d_1))/((r_2, d_2)/(r_1, d_1))$. By case analysis, we show that r' and d' depend only on the set $\{(r_1, d_1), (r_2, d_2)\}$ and not on the order. First observe that $d' = d - 2$ iff $d_1, d_2 \leq d$; also, $d' = d$ iff $d_1, d_2 \geq d + 1$ and $d_1 \neq d_2$. This takes care of showing order invariance for d' . Similarly, when $d' = d - 2$, it follows that $r' = r$ iff $r_1, r_2 \geq r$, and that $r' = r - 2$ iff $r_1, r_2 \leq r - 1$ and $r_1 \neq r_2$. When $d' = d$ it must be the case that $r' = r$. That leaves us with $d' = d - 1$. Here r' is either r or $r - 1$. In order for r' to be $r - 1$, at least one of the r_i , say r_1 , must be strictly less than r . Because of monotonicity, $d_1 \leq d$, so from earlier arguments d_2 must be strictly greater than d . Thus when $d' = d - 1$, the new release time r' will be $r - 1$ iff exactly one of $i = 1, 2$ has $r_i < r$ and $d_i \leq d$. \square

In order to establish faithfulness and swap invariance, we first define the notion of *slack*: Let $\delta(A, s, t) = t - s - |\{(r, d) \in A \mid r \geq s, d \leq t\}|$, so $\delta(A, s, t)$, the slack between s and t wrt A , is the amount of empty space between time slots s and t in the schedule for A . It is easy to see that $f = (r, d)$ is in a circuit of $A \cup \{f\}$ iff there exist $s \leq r$ and $t \geq d$ such that $\delta(A, s, t) = 0$. We now show that if $(r', d') = (r, d)/A$ then $d' - r' = \min\{\delta(A, s, t) \mid s \leq r, t \geq d\}$. First we prove two lemmas that are sufficient to establish faithfulness and swap invariance for deadline scheduling, where a loop corresponds to a deadline of 0. These lemmas are also used in the proofs for the more general monotone case.

LEMMA 3.2. *If $(r', d') = (r, d)/A$, then for any $t \geq d$, $d' \leq \delta(A, 0, t)$.*

Proof. Proof is by induction on $|A|$. If $A = \emptyset$ then $d' = d$ and the lemma is trivially true. Otherwise let $\hat{A} = A - \{(r_0, d_0)\}$, where (r_0, d_0) is any task in A . Let $(\hat{r}, \hat{d}) = (r, d)/\hat{A}$ and $(\hat{r}_0, \hat{d}_0) = (r_0, d_0)/\hat{A}$. By induction hypothesis, $\hat{d} \leq \delta(\hat{A}, 0, t)$. If $d_0 > t$, then $d' \leq \hat{d} \leq \delta(\hat{A}, 0, t) = \delta(A, 0, t)$. If $d_0 \leq t$ and $\hat{d} \leq \delta(\hat{A}, 0, t) - 1 = \delta(A, 0, t)$, then $d' \leq \delta(A, 0, t)$. Finally, if $d_0 \leq t$ and $\hat{d} = \delta(A, 0, t)$, we apply the induction hypothesis to \hat{d}_0 , to get $\hat{d}_0 \leq \delta(\hat{A}, 0, t)$. This means $\hat{d}_0 \leq \hat{d}$ and, by (1), $d' = \hat{d} - 1 \leq \delta(A, 0, t)$. \square

LEMMA 3.3. *If $(r', d') = (r, d)/A$, then there exists $t \geq d$ such that $d' \geq \delta(A, 0, t)$.*

Proof. Again, proof is by induction on $|A|$ with a trivial basis. Define \hat{A} , (r_0, d_0) , (\hat{r}, \hat{d}) , and (\hat{r}_0, \hat{d}_0) as in the previous proof. Using the induction hypothesis, let $\hat{t} \geq d$ be such that $\hat{d} \geq \delta(\hat{A}, 0, \hat{t})$; similarly choose $t_0 \geq d_0$ so that $\hat{d}_0 \geq \delta(\hat{A}, 0, t_0)$. If $d_0 \leq \hat{t}$, then $d' \geq \hat{d} - 1 \geq \delta(\hat{A}, 0, \hat{t}) - 1 = \delta(A, 0, \hat{t})$, and we choose $t = \hat{t}$. If $d_0 > \hat{t}$ and $d' = \hat{d} \geq \delta(\hat{A}, 0, \hat{t}) = \delta(A, 0, \hat{t})$, we choose $t = \hat{t}$. Finally, if $d_0 > \hat{t}$ and $d' = \hat{d} - 1$ then $\hat{d}_0 \leq \hat{d}$, and we are done by choosing $t = t_0$, since $d' = \hat{d} - 1 \geq \hat{d}_0 - 1 \geq \delta(\hat{A}, 0, t_0) - 1 = \delta(A, 0, t_0)$. \square

We have shown that $d' = \min\{\delta(A, 0, t) \mid t \geq d\}$, where $(r', d') = (r, d)/A$. So in deadline scheduling $d' = 0$ iff the task $(0, d)$ forms a circuit with the tasks of A . The following lemma, which is straightforward to prove by induction, is used throughout the remaining arguments. It shows, among other things, that monotonicity is preserved by the contraction operation.

LEMMA 3.4. *Let $(r'_i, d'_i) = (r_i, d_i)/A$ for $i = 1, 2$. Then $r_1 \geq r_2$ implies $r'_1 \geq r'_2$, and $d_1 \geq d_2$ implies $d'_1 \geq d'_2$. \square*

We now introduce constraints involving the release times after contraction, which, together with Lemmas 3.2 and 3.3 give us what we need for faithfulness, namely that

$d' - r' = \min\{\delta(A, s, t) \mid s \leq r, d \leq t\}$.

LEMMA 3.5. *If $(r', d') = (r, d)/A$, then for any s, t with $s \leq r, d \leq t$ we have $r' \geq \delta(A, 0, t) - \delta(A, s, t)$.*

Proof. Again we use induction on $|A|$ with a trivial basis. Here we carefully define $\hat{A} = A - \{(r_0, d_0)\}$ so that r_0 is as small as possible. The pairs (\hat{r}, \hat{d}) and (\hat{r}_0, \hat{d}_0) are as in earlier proofs. If $r' = \hat{r}$ then the lemma follows easily by induction. Suppose $r' = \hat{r} - 1$. This means $\hat{r}_0 < \hat{r}$ so, by Lemma 3.4, $r_0 < r$, which means $d_0 \leq d$. If $r_0 < s$ then $r' = \hat{r} - 1 \geq \delta(\hat{A}, 0, t) - \delta(\hat{A}, s, t) - 1 = \delta(A, 0, t) - \delta(A, s, t)$. Otherwise note that the choice of r_0 implies $\hat{r}_0 = r_0$. Since the induction hypothesis applies to \hat{r}_0 , we have $r' = \hat{r} - 1 \geq \hat{r}_0 \geq \delta(\hat{A}, 0, t) - \delta(\hat{A}, r_0, t) = \delta(A, 0, t) - \delta(A, r_0, t)$. We also know from the choice of r_0 that $\delta(A, r_0, t) \leq \delta(A, s, t)$ when $r_0 \geq s$, so the final expression is $\geq \delta(A, 0, t) - \delta(A, s, t)$ as desired. \square

Note that Lemma 3.5, together with Lemma 3.2, ensures that $d' - r' \leq \delta(A, s, t)$ for all relevant s, t . This means that when (r, d) is in a circuit of $A \cup \{(r, d)\}$ the algorithm will detect the circuit because $d' - r'$ will be 0. We complete the faithfulness argument by showing, via a more general argument, that when $d' - r'$ is 0, a circuit really does exist.

LEMMA 3.6. *If $(r', d') = (r, d)/A$, then there exist s, t with $s \leq r, d \leq t$ such that $\delta(A, s, t) \leq d' - r'$.*

Proof. Proof is by induction on $|A|$. In the basis, choose $s = r$ and $t = d$. In the induction step choose a task $(r_0, d_0) \in A$ with r_0 as small as possible. The definitions of \hat{A} (\hat{r}, \hat{d}) , and (\hat{r}_0, \hat{d}_0) are as in earlier proofs. We also use the induction hypothesis to define \hat{s}, \hat{t} with $\hat{s} \leq r, d \leq \hat{t}$ and $\delta(\hat{A}, \hat{s}, \hat{t}) \leq \hat{d} - \hat{r}$.

If $r' = \hat{r}, d' = \hat{d}$ or $r' = \hat{r} - 1, d' = \hat{d} - 1$ we can choose $s = \hat{s}$ and $t = \hat{t}$ and the lemma follows easily by induction. We are left only with the situation where $r' = \hat{r}$ and $d' = \hat{d} - 1$. Three separate cases need to be considered. In the first two cases we choose s, t so that $\delta(A, s, t) \leq \delta(\hat{A}, \hat{s}, \hat{t}) - 1$ and the lemma follows by induction. In the third case the lemma follows directly using earlier lemmas.

Case 1. $r_0 \geq \hat{s}, d_0 \leq \hat{t}$. Here choose $s = \hat{s}, t = \hat{t}$ and note that $\delta(A, s, t) = \delta(\hat{A}, \hat{s}, \hat{t}) - 1$.

Case 2. $r_0 < \hat{s}, d_0 \leq \hat{t}$. In this case choose $s = r_0$ and $t = \hat{t}$. Because r_0 was chosen to be as small as possible, we have $r_0 = \hat{r}_0 \geq \hat{r}$ and $\delta(A, s, t) = \delta(A, 0, \hat{t}) - r_0 = \delta(\hat{A}, 0, \hat{t}) - 1 - r_0 \leq \delta(\hat{A}, 0, \hat{t}) - 1 - \hat{r}$. Lemma 3.5 says that $\hat{r} \geq \delta(\hat{A}, 0, \hat{t}) - \delta(\hat{A}, \hat{s}, \hat{t})$, so the final expression is $\leq \delta(\hat{A}, \hat{s}, \hat{t}) - 1$ as desired. This case arises, for example, when $(r, d) = (1, 3)$, $(r_0, d_0) = (0, 1)$, and $\hat{A} = \{(0, 2)\}$.

Case 3. $d_0 > \hat{t}$. Here choose $s = r$ and $t = t_0$, where $t_0 \geq d_0$ and $\hat{d}_0 \geq \delta(\hat{A}, 0, t_0)$; such a t_0 exists by Lemma 3.3. We have $\delta(A, s, t) = \delta(\hat{A}, r, t_0) - 1$. Because of monotonicity we know that $r_0 \geq r$. So by the choice of r_0 there are no tasks in \hat{A} with deadlines less than r , implying that $\delta(A, s, t) = \delta(\hat{A}, 0, t_0) - r - 1$. By the choice of t_0 this is $\leq \hat{d}_0 - r - 1 \leq d' - r'$, as desired. An example of this case is when $(r, d) = (0, 2)$, $(r_0, d_0) = (1, 3)$, and $\hat{A} = \{(1, 3)\}$. \square

Lemmas 3.2, 3.5, and 3.6 together imply faithfulness of (1): we have established that (r, d) is in a circuit of $A \cup \{(r, d)\}$ iff there exist s, t with $s \leq r, d \leq t$ and $\delta(A, s, t) = 0$,

which is true iff $d' - r' = 0$. To establish swap invariance we prove that the *minimum surrounding slack* for arbitrary $a \leq b$, defined to be $\min\{\delta(A, s, t) \mid s \leq a, b \leq t\}$, does not change in value as the result of a swap. This suffices because $d' - r'$ is determined by the minimum surrounding slack for r, d and d' is determined by the minimum surrounding slack for $0, d$ (by Lemmas 3.2 and 3.3). The following lemma gives us what we need to complete the proof of swap invariance for (1).

LEMMA 3.7. *Let $(r_0, d_0), (r_1, d_1)$ be a swap wrt A and let $A' = A \cup \{(r_0, d_0)\} - \{(r_1, d_1)\}$. Then for any $a \leq b$, (i) there exist a', b' satisfying $a' \leq a, b \leq b'$ and $\delta(A', a', b') \leq \delta(A, a, b)$, and (ii) if $\delta(A', a, b) < \delta(A, a, b)$ then there exist \hat{a}, \hat{b} with $\hat{a} \leq a, b \leq \hat{b}$ and $\delta(A, \hat{a}, \hat{b}) < \delta(A, a, b)$.*

Proof. It is known (see, e.g. [GT84]) that $s \leq r_1, d_1 \leq t$, where s and t are chosen so that $s \leq r_0, d_0 \leq t$ and $\delta(A, s, t) = 0$. If $s \leq a, b \leq t$, the lemma follows immediately. If $a \leq s, t \leq b$ then $\delta(A', a, b) = \delta(A, a, b)$ follows by the choice of s, t and again the lemma is established. The remaining case is when $a < s < b < t$ (or, symmetrically, $s < a < t < b$). Here, note that the definition of slack implies $\delta(A', a, t) = \delta(A, a, t) \leq \delta(A, a, b) + \delta(A, s, t) - \delta(A, s, b) \leq \delta(A, a, b)$. So part (i) has been shown with $a' = a, b' = t$. For part (ii) observe that $\delta(A', a, b) < \delta(A, a, b)$ only if $d_0 \leq b$. Thus $\delta(A, a, t) = \delta(A', a, t) \leq \delta(A', a, b) < \delta(A, a, b)$ and we are done by choosing $\hat{a} = a, \hat{b} = t$. \square

Monotonicity appears to be essential in this particular approach to scheduling. Order invariance is violated, for example, if $a = (0, 3)$, $b = (1, 2)$, and $c = (0, 3)$. Then $(a/b)/(c/b) = (0, 2)$ but $(a/c)/(b/c) = (0, 1)$. This example could leave open the possibility of a more clever contraction operation for non-monotone scheduling. We dispel this notion by observing that matroids based on scheduling with release times and deadlines are not closed under contraction (unless they are monotone). Consider a scheduling matroid \mathcal{M} in which $a_i = b_i = (i-1, i)$ for $i = 1, 2, 3$ and let $c = (0, 3)$. Then $\mathcal{M}/\{c\}$ has circuits $\{a_i, b_i\}$ for $i = 1, 2, 3$ and all 8 sets of the form $\{x_1, x_2, x_3\}$, where each x_i is either a_i or b_i . It can be shown, using for example the characterization of Bondy and Welsh (see [Aig79], page 378), that $\mathcal{M}/\{c\}$ is not even a transversal matroid, much less a monotone scheduling matroid. Intuitively, at least 3 time slots are needed to represent the fact that $\{a_i, b_i\}$ is a circuit for each i , but the rank of $\mathcal{M}/\{c\}$ is 2. The smallest contraction-closed class of matroids known to contain monotone scheduling is the class of *gammoids* (see [Aig79]). Elements of a gammoid cannot be represented as succinctly as those of a scheduling matroid.

4. Implementation Details. We now describe how the optimum base is produced as output and illustrate, with an example, how the array algorithm works. When the separator $\$$ is received by a cell, the cell enters “push mode”. A cell receiving $\$$ at time t sends its matroid element, marked for output, to the next cell at time $t + 1$ and sends $\$$ at time $t + 2$. A cell receiving an element marked for output at time t may send either its own matroid element or the one it has just received to the next cell at time $t + 1$, and the remaining element at time $t + 2$. The ability to choose which element to send first allows the array to sort its output by any desired key. A sort by increasing release time (and increasing deadline as a secondary key) would allow the

last cell to assign time slots to the elements it outputs using the algorithm of Glover [Glo67] (the next feasible slot is assigned to a task with earliest release time — earliest deadline is used to break ties). Note that the last cell does not need to know it is last — all other cells can assign time slots using the same algorithm, but these assignments are superseded by the assignment of the last cell, which reaches the host.

Figure 1 shows an example of monotone scheduling with 4 tasks having rank 3. The top half of each cell i shows the current representation of e'_i as the pair r'_i, d'_i . The bottom half shows the original element e_i as r_i, d_i, w_i . Arrows indicate the input/output values at the start of the current time unit. When both f' and *previous* are sent to the next cell (see Algorithm Array-Greedy on page 6), f' is shown above the arrow and *previous* is shown below. The output phase is initiated at time unit 4. During time unit 5, cell 1 reinitializes itself to a dummy value and is ready to process the next problem instance. The output is sorted by release time and then deadline in cell 2 during time unit 6 and cell 3 during time units 7 and 8.

The actions of the generic algorithm (page 6) may be traced for an input record f appearing at time j by looking at cell i , time $j + i$, for $i = 1$ to n . Task $a = 0, 2, 5$ (which appears at time 0) is inserted in cell 1 at time 1. Task $b = 0, 1, 6$ is inserted in cell 1 at time 2, causing $a' = a/b$ to replace the dummy record in cell 2 at time 3. Task $c = 2, 3, 4$ moves past b (cell 1, time 3) and a' (cell 2, time 4) to find its proper position in cell 3 at time 5. Task $d = 1, 3, 7$ inserts itself in front of b (cell 1, time 4), which in turn pushes forward b' (from cell 1 to cell 2, time 4) and a' (from cell 2 to cell 3, time 5). Finally, d' and c' discover a circuit (d'/c' is a loop) in cell 3 at time 6, causing c' to be deleted (a' is inserted in its place). The optimal schedule includes tasks a , b , and d with b assigned to slot 1, a to slot 2, and d to slot 3.

5. Conclusions. What has been described is a generic one-way array algorithm for matroid problems that is analogous to the greedy algorithm in the sequential model. Like the greedy algorithm, the one-way array algorithm can, theoretically, be adapted to solve any matroid optimization problem. The primary issue, however, is the efficiency of testing for a circuit. For example, were we to store all of A in our representation of f/A , and allow sufficient computational power per time unit to solve a linear system of equations, the generic one-way algorithm could be used to solve optimization problems on arbitrary *linear* matroids (see [Aig79] for a definition). Linear matroids include most problems that are of practical interest. A straightforward implementation of this idea would require $\Theta(n^3)$ total area and a period in $\Theta(M(n))$, where $M(n)$ is the time for $n \times n$ matrix multiplication (the period can be reduced if each cell is allowed to be a parallel machine, or is custom designed in VLSI). It is likely that efficient implementations of the one-way algorithm (constant period and roughly linear area) exist for matroids which have efficient (subquadratic) greedy algorithms, as has already been demonstrated for minimum spanning trees and monotone scheduling (the author is not aware of sequential algorithms specifically for monotone scheduling, but it is possible that techniques for deadline scheduling described by Gabow and Tarjan [GT84] can be generalized).

One interesting open question related to one-way array algorithms for matroid prob-

lems is whether there exists an efficient one-way array algorithm for the non-monotone scheduling problem. The fact that non-monotone scheduling is not closed under contraction rules out an approach based on the generic greedy algorithm given here — the most efficient result would have to be based on linear matroids (gammoids are linear matroids). It may, however, be possible to obtain an efficient algorithm if tasks are sorted in the array by some other criterion than weight, or auxiliary information is cleverly encoded throughout the array. Lack of closure under contraction may also explain why the best time bound known for non-monotone scheduling is $O(m \log n + n^2)$ [GT84] and the known efficient algorithms for independence testing appear to require non-trivial data structures and use of random access memory [Fre83, GT85, LP81].

REFERENCES

- [Aig79] M. Aigner. *Combinatorial Theory*. Springer Verlag, 1979.
- [AS85] S. Ashtaputre and C. Savage. Systolic arrays with embedded tree structures for connectivity problems. *IEEE Transactions on Computers*, C-34(5):483 – 484, 1985.
- [Fre83] G. N. Frederickson. Scheduling unit-time tasks with integer release times and deadlines. *Information Processing Letters*, 16:171 – 173, 1983.
- [FS87] G. N. Frederickson and M. A. Srinivas. On-line updating of solutions to a class of matroid intersection problems. *Information and Computation*, 74:113 – 139, 1987.
- [Glo67] F. Glover. Maximum matching in a convex bipartite graph. *Naval Research Logistics Quarterly*, 14:313 – 316, 1967.
- [GT84] H. N. Gabow and R. E. Tarjan. Efficient algorithms for a family of matroid intersection problems. *Journal of Algorithms*, 5:80 – 131, 1984.
- [GT85] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209 – 221, 1985.
- [Hua90] Shing-Tsaan Huang. A fully pipelined minimum-cost-spanning tree constructor. *Journal on Parallel and Distributed Computing*, 9(1):55 – 62, 1990.
- [Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
- [LP81] W. Lipski, Jr. and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329 – 346, 1981.
- [SSK87] C. D. Savage, M. F.M. Stallmann, and A. Z. Kotob. Simulation of two-way computations on arrays with one-way data flow. Technical Report CCSP-TR-87/6, North Carolina State University Center for Communications and Signal Processing, 1987.
- [SSP90] C.D. Savage, M. Stallmann, and J.E. Perry. Solving some combinatorial problems on arrays with one-way dataflow. *Algorithmica*, 5:179 – 199, 1990.
- [ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362 – 391, 1983.
- [TM83] M. Tchuente and L. Melkemy. Reseaux systoliques pour le calcul des composantes connexes et le triangularisation des matrices bandes. Technical Report 366, Laboratoire d’Informatique et de Mathematiques Appliquees de Grenoble, 1983.

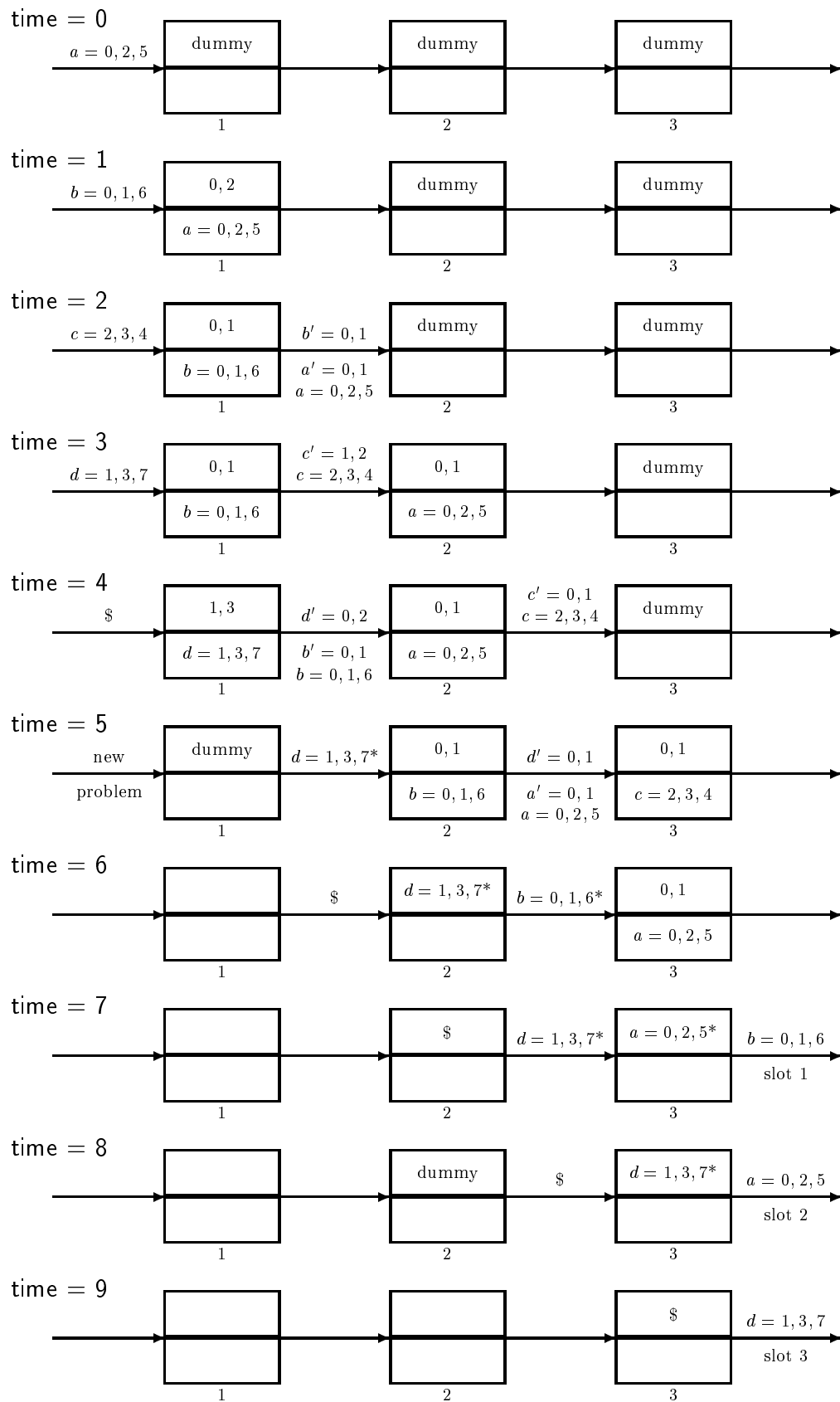


FIG. 1. An example of the algorithm for monotone scheduling