Efficient Theoretic and Practical Algorithms for Linear Matrod Intersection Problems

Harold N. Gabow and Ying Xu

CU-CS-424-89 January 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Efficient Theoretic and Practical Algorithms for Linear Matroid Intersection Problems

Harold N. Gabow[1]
Ying Xu
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309

January 18, 1989

Abstract.

Efficient algorithms for the matroid intersection problem, both cardinality and weighted versions, are presented. The algorithm for weighted intersection works by scaling the weights. The cardinality algorithm is a special case, but takes advantage of greater structure. Efficiency of the algorithms is illustrated by several implementations on linear matroids. Consider a linear matroid with m elements and rank n. Assume all element weights are integers of magnitude at most N. Our fastest algorithms use time $O(mn^{1.77}\log(nN))$ and $O(mn^{1.62})$ for weighted and unweighted intersection, respectively; this improves the previous best bounds, $O(mn^{2.4})$ and $O(mn^2\log n)$, respectively. Corresponding improvements are given for several applications of matroid intersection to numerical computation and dynamic systems.

---

[1] Research supported in part by NSF Grant Nos. MCS-8302648, DCR-851191.

1. Introduction.

The matroid intersection problem is interesting from a theoretic point of view and because of its rich set of applications. Concerning theory, matroid intersection generalizes bipartite graph matching; powerful duality results (e.g., the König-Egervary minimax theorem) generalize to matroids (e.g., Edmonds' matroid intersection theorem) [L]. Concerning applications, matroid intersection can be used to analyze continuous systems arising in electrical networks, elastic structures and chemical processing plants. The analysis is attractive because problems that might naturally be solved by numerical computation, subject to rounding error and inaccuracies in the data, are replaced by discrete calculations, which are more efficient and relatively insensitive to these numerical difficulties. Examples of such applications of matroid intersection include solvability and dynamic degrees of freedom of an electrical network [I] and their generalization to arbitrary dynamic systems governed by linear differential equaitons [M87a]. (For electrical networks the matroids tend to be graphic; for general systems they are linear. Applications are discussed further in Section 5.)

This paper presents efficient algorithms for a number of matroid intersection problems. The main algorithm is for weighted matroid intersection. It is based on cost scaling, generalizing recent work in matching [GaT87] and network flow [GoT]. A special case is of this problem is cardinality matroid intersection, and several improvements are given for this case. The efficiency of our intersection algorithms is illustrated by implementations on linear matroids. This is the broadest class of matroids with efficient representations. (Implementations of our general matroid algorithm on other matroids are given in [GX].)

To state the specific results, assume that a linear matroid is given consisting of m elements with dimension n. For weighted intersection assume that all weights are integers at most N in magnitude. We distinguish two types of algorithms: those that use only naive procedures for manipulating matrices (e.g., the simple $O(n^3)$ algorithm for matrix multiplication) and those that take advantage of sophisticated procedures (e.g., the $O(n^{2.4})$ multiplication algorithm of [CW]). Algorithms of the first type do not have as good an asymptotic time bound. However they can be more practical than algorithms of the second type, which have large constants hidden in the asymptotic bounds.

The main previous contribution for linear matroids is the cardinality intersection algorithm of Cunningham [Cu]. It finds a maximum cardinality matching on a linear matroid in time $O(mn^2 \log n)$, using naive matrix manipulation. Our naive algorithm for cardinality intersection is essentially the same as Cunningham's. Using fast matrix multiplication our algorithm runs in time $O(mn^{1.62})$.

For the weighted problem, our naive algorithm finds a maximum weight perfect matching in time $O(mn^2 \log n \log(nN))$. This bound is just a factor $\log(nN)$ more than Cunningham's bound for the simpler problem without weights. Using fast matrix multiplication our algorithm runs in time $O(mn^{1.77} \log(nN))$. The previous best bound for this problem is achieved by Frank's algorithm [F] (and a number of similar algorithms). This gives time $O(mn^3)$ using naive multiplication and $O(mn^{2.4})$ using fast multiplication. Frank's algorithm has the theoretic advantage that the time bound is strongly polynomial. (Advantages and disadvantages of strong polynomial algorithms compared to scaling algorithms are discussed in [G, GoT].)

It is interesting to compare our results with recent scaling algorithms for minimum cost network flow [GoT], the assigment problem [GaT87], minimum weight matching [GaT88] and others. These papers present scaling algorithms with a time bound for the weighted problem that is essentially within a factor $\log(nN)$ of the best-known bound for the (simpler) cardinality problem. As mentioned, this is also the case for our weighted matroid intersection algorithm using naive multiplication. However using fast multiplication there is much larger gap between our bounds for weighted and cardinality problems, about $n^{.15}$. We leave open the problem of closing this gap, or making other improvements to our algorithms.

The rest of this section gives some terminology and notation; then it sketches Frank's algorithm for weighted matroid intersection [F]. Our algorithm uses several ideas of this algorithm. Section 2 presents our algorithm for weighted matroid intersection, on general matroids. Section 3 gives the algorithm for cardinality matching, plus other extensions of the weighted algorithm. Section 4 gives implementations of the matching algorithms on linear matroids. Section 5 sketches the extensions to related problems such as independent assignment and linking (definitions of these problems are in Section 5). It concludes by giving a number of applications to problems in numerical computation and dynamic systems.

If S is a set and e an element, $S+e$ denotes $S \cup \{e\}$ and $S-e$ denotes $S-\{e\}$. The *symmetric difference* of sets S and T is denoted $S \oplus T$. We use interval notation for sets of integers: for integers i and j, $[i..j] = \{k | k \text{ is an integer, } i \leq k \leq j\}$, $[i..j) = \{k | k \text{ is an integer, } i \leq k < j\}$, etc. The *log* function denotes logarithm base two. We use the following convention to sum the values of a function: If f is a real-valued function defined on elements and S is a set of elements, then $f(S) = \sum\{f(s) | s \in S\}$.

The algorithms for linear matroids do matrix multiplication. We could state the time bounds in terms of $M(n)$, the time to multiply two $n \times n$ matrices. Instead we use a slightly less precise notation, that has the advantage of simplifying the algebra: We state the bounds in terms of ω,

a real value satisfying $M(n) = O(n^{2+\omega})$. Hence $0 \leq \omega \leq 1$. The simple algorithm for matrix multiplication gives $\omega = 1$; $\omega < 0.376$ using the algorithm of [CW], although this algorithm is not practical.

Strictly speaking our time estimates do not take into account the fact that the size of the numbers can grow because of repeated matrix multiplications and inversions. Hence the bounds either assume we are working over a finite field, or the bounds count the number of arithmetic operations.

$\mathbf{I}$ denotes the identity matrix. The dimension of this matrix will be clear from context.

The reader is assumed familiar with the basic notions of matroids (see, e.g., [A, L, W]). Let $\mathbf{M}$ be a matroid over a set of elements E. The *span* of a set of elements S is denoted $sp(S)$. Suppose e is an element and B is a base. The notation $C(e,B)$ denotes the *fundamental circuit* of e in B if $e \notin B$, or the *fundamental cocircuit* of e for B if $e \in B$. (It will be clear from context whether or not $e \in B$.) On occasion we write $C(e,B,\mathbf{M})$ if the matroid $\mathbf{M}$ is not clear. If S and T are sets of elements then S/T denotes the matroid *restricted* to S with T *contracted.* If $\mathbf{M}$ and $\mathbf{N}$ are matroids, their *direct sum* is denoted $\mathbf{M} + \mathbf{N}$.

Let the element set E be partitioned into blocks of size two called *pairs.* Thus each element e has a mate, denoted $\overline{e}$, such that $\{e, \overline{e}\}$ is a pair. (We usually drop brackets and denote a pair as $e, \overline{e}$.) A *matching* M is an independent set of pairs, i.e., $e \in M$ if and only if $\overline{e} \in M$. A *maximum cardinality matching* has the greatest number of pairs possible. A matching is *perfect* if it is a base of $\mathbf{M}$. If $w : E \rightarrow \mathbf{R}$ is a function that assigns a real-valued *weight* to each element, the weight of a set S of elements is defined as $w(S) = \sum\{w(e) | e \in S\}$. A *maximum perfect matching* is a perfect matching that has the largest weight possible. *Maximum weight maximum cardinality matching* is defined similarly. A *maximum weight matching* is a matching with the largest weight possible.

The above problems generalize the problem of matching on graphs. This paper is concerned with the "bipartite" versions of these matching problems. Specifically, throughout this paper the $\mathbf{M}$ is a direct sum of matroids, $\mathbf{M} = \mathbf{M}_0 + \mathbf{M}_1$, where $\mathbf{M}_i$ is a matroid on elements E_i, $i = 0, 1$ (so $E = E_0 \cup E_1$) and every pair contains one element from each matroid. The *cardinality matroid intersection problem* is to find a maximum cardinality matching on such a matroid $\mathbf{M}$. The *weighted matroid intersection problem* is to find either a maximum perfect matching, maximum weight maximum cardinality matching, or maximum weight matching, on $\mathbf{M}$.

The following notation is useful in the context of intersection problems. For any set of elements S and $i \in \{0,1\}$, S_i denotes the set $S \cap E_i$. The parameters m and n denote the number of elements in E and the rank of $\mathbf{M}$ respectively, unless stated otherwise.

1.1. Frank's Algorithm.

This section sketches Frank's matroid intersection algorithm [F], which contains several fundamental concepts used in Section 2. We modify Frank's algorithm slightly to suit our purposes. (The main modification is using "singleton" elements).

We take the basic problem to be finding a maximum perfect matching. Other variants of matroid intersection easily reduce to perfect matching. For convenience assume the given matroid has a perfect matching. (The algorithm is easily modified to detect a matroid that does not have a perfect matching.)

It is convenient for the algorithm to always work with a base. To do this we introduce "singleton" elements. A *singleton* is an element parallel to an element of E; it does not have a mate. S denotes the set of singletons; S is disjoint from the given elements E. If M is a matching, $\overline{M}$ denotes the set $M \cup S$. The algorithm maintains a set of singletons S and a matching M so that $\overline{M}$ is a base for $\mathbf{M}$.

Frank's algorithm is a primal-dual algorithm in the sense of linear programming [D]. There is a *dual function* $x : E \cup S \rightarrow \mathbf{R}$; the dual value $x(e)$ is sometimes called the *weight* of e. Note that our notational convention for functions implies the following: If T is a set of elements, $x(T) = \sum\{x(e)|e \in T\}$. Thus for a pair $e, \overline{e}$, $x(e, \overline{e}) = x(e) + x(\overline{e})$. Similarly if M is a matching, $x(M) = \sum\{x(e)|e, \overline{e} \in M\}$.

The algorithm maintains the dual function to be *feasible*, i.e., for every pair $e, \overline{e}$,

$$w(e, \overline{e}) = x(e, \overline{e}).$$

Obviously a perfect matching M is maximum if there is a feasible dual function x such that M is a maximum weight base for $\mathbf{M}$ with respect to the weight function x. Recall that a base B has maximum weight for weight function x if and only if it is *dominating*, i.e.,

$$x(e) \leq \min\{x(f)|f \in C(e, B)\} \qquad \text{for } e \notin B.$$

The algorithm maintains x so it is feasible and $\overline{M}$ so it is dominating for x. Hence when $S = \emptyset$, M is the desired maximum perfect matching and the algorithm halts.

The algorithm repeatedly augments M to increase the number of matched pairs by one. This is done as follows. A *swap* (for $\overline{M}$) is an ordered pair of elements e, f such that $\overline{M} \oplus \{e, f\}$ is a base and either $e \in \overline{M}_0, f \in E_0 - \overline{M}$, or $e \in E_1 - \overline{M}, f \in \overline{M}_1$; furthermore, $x(e) = x(f)$. To *execute swap e,f* means to change the current base $\overline{M}$ to $\overline{M} \oplus \{e, f\}$. A *weighted augmenting path* (*wap*)

for $\overline{M}$ is a sequence of swaps

$$P = \overline{e_0} f_0 \overline{f_0} e_1 \overline{e_1} f_1 \overline{f_1} \ldots f_k \overline{f_k} e_{k+1}.$$

More precisely P consists of pairs $e_i, \overline{e_i} \in M$, $0 < i \leq k$, singletons $\overline{e_0}, e_{k+1} \in \overline{M}$, and pairs $f_i, \overline{f_i} \notin M$, $0 \leq i \leq k$, such that for $0 \leq i \leq k$, $\overline{e_i}, f_i$ and $\overline{f_i}, e_{i+1}$ are swaps for $\overline{M}$; furthermore $\overline{M} \oplus P$ is a base. Note this base $\overline{M} \oplus P$ has one more pair than $\overline{M}$. The process of converting $\overline{M}$ to $\overline{M} \oplus P$ is called *augmenting the matching along* P. This amounts to executing the swaps of P.

Frank's algorithm as presented in [F] uses a slightly more restrictive definition of *wap*. It requires that P has no "shortcuts", i.e., no swaps $\overline{e_i}, f_j$ or $\overline{f_i}, e_{j+1}$ with $j > i$. The no-shortcut condition implies that $\overline{M} \oplus P$ is a base, but the condition is not necessary. All other algorithms for matroid intersection (either weighted and cardinality intersection) that we know of use the no-shortcut condition as part of the definition of augmenting path (e.g., [Cu,GS85,L], and for matroid partitioning, [E,K,W]). The algorithm of this paper weakens the no-shortcut condition to the condition that $\overline{M} \oplus P$ is a base. We could not achieve the desired level of efficiency using augmenting paths that have no shortcuts.

Frank's algorithm finds a *wap* by doing a *Hungarian search* (so-called because it generalizes the Hungarian search of graph matching [L]). The input to the search is a dominating base $\overline{M}$ with feasible dual function x. The output is a new feasible dual function for which $\overline{M}$ is still dominating and there is a *wap*.

The Hungarian search works by growing a forest $\mathcal{F}$ whose nodes are elements of $\overline{M}$. Recall that by our notational convention, $\mathcal{F}_i$ denotes the elements of $\mathcal{F}$ in E_i. Call an element in $\mathcal{F}$ *even* or *odd*, depending on its level. (A root of a search tree of $\mathcal{F}$ is on level zero and so is even). The roots of $\mathcal{F}$ are the singletons of $\overline{M}_0$. If e is an even element, each of its children f gives a swap e, f. If e is an odd element that is not a singleton of $\overline{M}_1$ then e has one child, $\overline{e}$. If e is an odd element that is a singleton of $\overline{M}_1$ then $\mathcal{F}$ contains a *wap* (and the search is done).

The Hungarian search starts with a forest $\mathcal{F}$ consisting of roots, all the singletons of $\overline{M}_0$. It repeatedly does a "grow step" followed by a "dual adjustment step", until $\mathcal{F}$ contains a *wap*. A *grow step* adds elements to $\mathcal{F}$ until it is maximal. For an element e define

$$\epsilon(e) = \textbf{if } e \in E_0 \textbf{ then } -1 \textbf{ else } 1.$$

A *dual adjustment step* calculates a positive quantity δ, and then increases the duals of all elements $e \in \mathcal{F}$ by $\epsilon(e)\delta$. The dual adjustment quantity δ is chosen as large as possible to keep $\overline{M}$ dominating.

This ensures that there is a new swap, allowing an element to be added to the forest in the following grow step.

The Hungarian search stops when $\mathcal{F}$ contains a *wap* P. Then the algorithm does an *augment step*, which changes the matching to $M \oplus P$ and deletes the singletons of P from S. Hence the new set $\overline{M} = M \cup S$ is a base.

To summarize, the following algorithm finds a maximum perfect matching. The input is a base of singletons S and a feasible dual function x, such that S is dominating for x. The algorithm repeatedly does a Hungarian search to find a *wap* P, and then it augments the matching. The algorithm halts when the matching is perfect.

2. Weighted Intersection Algorithm.

This section presents the algorithm for weighted intersection on general matroids, along with its analysis.

We state our algorithm for the problem of maximum perfect matching. In addition we assume that the matroid has a perfect matching. The other versions of weighted matroid intersection easily reduce to this problem, with no loss in asymptotic efficiency. For instance, consider the problem of finding a maximum weight matching on a matroid $\mathbf{M}$. It reduces to maximum perfect matching on a matroid $\mathbf{N}$, constructed as follows. Start with $\mathbf{M}$ and also $\mathbf{M}'$, a copy of $\mathbf{M}$ (with the same pairs). In both $\mathbf{M}$ and $\mathbf{M}'$, create a copy of each element. Pair each new element of $\mathbf{M}$ with its copy (also new) in $\mathbf{M}'$. Each such pair has weight zero. If $\mathbf{M}$ is the direct sum $\mathbf{M_0} + \mathbf{M_1}$ then $\mathbf{N}$ is the direct sum, $(\mathbf{M_0} + \mathbf{M_1'}) + (\mathbf{M_1} + \mathbf{M_0'})$, and each pair has one element in both summands. Further it is easy to see that a matching of $\mathbf{M}$ corresponds to a perfect matching of $\mathbf{N}$ having twice the weight. A similar reduction holds for maximum weight maximum cardinality matching, the only difference being that in $\mathbf{N}$ pairs consisting of two copies of an element have weight $-nN$ instead of zero (here n is the sum of the ranks of $\mathbf{M}_i$.

A *1-feasible matching* consists of a matching M, with singleton set S giving base $\overline{M} = M \cup S$, plus a dual function $x : E \cup S \to \mathbf{Z}$ ($\mathbf{Z}$ denotes the integers), such that

$$x(e,\overline{e}) \geq w(e,\overline{e}) - 1, \qquad \text{for } e,\overline{e} \notin M; \tag{1a}$$

$$x(e,\overline{e}) = w(e,\overline{e}), \qquad \text{for } e,\overline{e} \in M; \tag{1b}$$

$$x(e) \leq \min\{x(f) | f \in C(e,\overline{M})\}, \qquad \text{for } e \notin \overline{M}. \tag{1c}$$

(Condition (1c) is that $\overline{M}$ is dominating for x; we repeat the definition of dominance for convenience). A *1-optimal matching* M is a 1-feasible perfect matching. We denote a 1-feasible matching

as M, x, or M if the dual function is understood.

This definition generalizes 1-optimality for the assignment problem as defined in [GaT87]. The following fundamental property also generalizes [GaT87].

Lemma 2.1. If some integer $k > n$ divides each weight $w(e, \overline{e})$ evenly, then any 1-optimal matching is a maximum perfect matching.

Proof. Let M be a 1-optimal matching and P any perfect matching. Dominance implies that M is a maximum weight base for the weight function x. Hence $w(M) = x(M) \geq x(P) \geq w(P) - n$. Since $w(M)$ and $w(P)$ are both multiples of k, this implies $w(M) \geq w(P)$ as desired. ∎

This lemma is the basis for the *main routine* of the algorithm. It scales the weights. The routine starts by computing a new weight $\overline{w}(e, \overline{e})$ for each pair $e, \overline{e}$, equal to $n + 1$ times the given weight. Consider each $\overline{w}(e, \overline{e})$ to be a signed binary number $\pm b_1 b_2 \ldots b_k$ of $k = \lfloor \log(n+1)N \rfloor + 1$ bits. The routine maintains a variable $w(e, \overline{e})$ for each pair $e, \overline{e}$, equal to its weight in the current scale. The routine initializes each $w(e, \overline{e})$ to 0, each dual $x(e)$ to 0, and M to an arbitrary perfect matching. Then it executes the following loop for index s going from 1 to k:

Double Step. For each element $e \in M$, create a singleton e' parallel to e, and set $x(e') \leftarrow x(e)$. Let S be the set of all such singletons. For each pair $e, \overline{e}$, set $w(e, \overline{e}) \leftarrow 2w(e, \overline{e}) +$ (signed bit b_s of $\overline{w}(e, \overline{e})$). For each element $e \in E \cup S$, set $x(e) \leftarrow 2x(e) + 1$.

Match Step. Call the *match routine* with the 1-feasible matching $\emptyset, x$ to find a 1-optimal matching M, x for weights w. ∎

The *main routine* is correct, since Lemma 2.1 implies that it halts with a maximum perfect matching, assuming the *match routine* operates as described in the Match Step. Note that as claimed, the empty matching with dual function x is 1-feasible. This follows from the Double Step. (Also note that the *match routine* described below eventually deletes all of the singletons created in the Double Step.)

Each iteration of the *main routine* is called a *scale*. There are $O(\log(nN))$ scales.

Now we describe the *match routine*. It works by repeatedly finding and augmenting a *wap*. Its efficiency stems from two closely related ideas which we now present.

The first idea is to use short augmenting paths. To achieve this we add one more requirement to the definition of *wap* (as already given in Section 1.1). Define the *weight-length* of a pair $e, \overline{e}$

with respect to a matching M to be

$$wl(e,\overline{e}) = w(e,\overline{e}) + (\textbf{if } e \notin M \textbf{ then } -1 \textbf{ else } 0).$$

A pair $e,\overline{e}$ is *eligible* if $x(e,\overline{e}) = wl(e,\overline{e})$. For instance any matched pair is eligible. We add to the definition of *wap* the requirement that every pair is eligible. We shall see (Lemma 2.4) that this produces short augmenting paths. (Further intuition for this definition can be found in [GaT87].)

The second idea is to find the augmenting paths in batches. By this we mean that as many *waps* as possible are found and augmented, before doing a Hungarian search to create new *waps*. The details of this idea are given below.

The following *match routine* is called with a set of singletons S and a dual function x such that the empty matching is 1-feasible (see the Match Step of the *main routine*).

procedure *match.*
Initialize the matching M to $\emptyset$. Then repeat the following until the Search Step returns with the desired matching:

Augment Step. Repeat the following until M does not have a *wap*: Find a *wap* P. For each $e \in (P_0 \cap M) \cup (P_1 - \overline{M})$ (i.e., e is not a singleton and is the first element of a swap of P) increase $x(e)$ by $\epsilon(e)$. Augment M along P. Delete the two singletons of P from S.

Search Step. If M is perfect then return the 1-optimal matching M, x. Otherwise do a Hungarian search to adjust the duals, maintaining 1-feasibility, and find a *wap*. ∎

To analyze the *match routine*, first observe that it is correct, for the following reasons: The Augment Step preserves 1-feasibility, as proved below in Lemma 2.6 (b). (Clearly the changes to the duals maintain the 1-feasibility relations ($1a$) – ($1b$).) It will be obvious that the Hungarian search preserves 1-feasibility. If M is not a perfect matching, the Hungarian search creates a *wap*. Hence the algorithm eventually returns with a 1-optimal matching.

Before giving the implementation of the two steps of *match* we derive some properties that are useful for analyzing the running time. These properties are similar to [GaT87]. The properties depend only on the details of *match* presented so far, and also the following basic properties that we shall see:

The Hungarian search is essentially the same as in Frank's algorithm (Section 1.1). The main modification is that the search forest $\mathcal{F}$ only contains eligible edges. At any point in the algorithm

define

$\Delta =$ the sum of all dual adjustment quantities δ in all Hungarian searches so far;

$S =$ the current singleton set;

$\sigma = |S_0|$;

$x =$ the current dual function.

We use a subscript of "minus" to refer to the time *match* begins, so x_- denotes the dual function on entry to *match*, etc. For a singleton $s \in S_0$, every dual adjustment of every Hungarian search decreases $x(s)$; for $s \in S_1$ no dual adjustment changes $x(s)$. Hence for $s \in S$,

$$x(s) = x_-(s) - (\textbf{if } s \in S_0 \textbf{ then } \Delta \textbf{ else } 0). \tag{2}$$

(Although the Augment Step changes duals, it does not change duals of singleton elements, so it preserves (2).) Finally we will show that because the Augment Step finds as many *waps* as possible, each Hungarian search adjusts the duals by at least one (Lemma 2.7).

Lemma 2.2. At any point in the execution of *match*, $\sigma\Delta \leq 5n$.

Proof. Let w be the weight function of the current scale and M the current matching (so $\overline{M} = M \cup S$). Let M_- be the 1-optimal matching of the previous scale (or in the first scale, the initial matching). It suffices to prove that

$$w(M_-) - n \leq x(\overline{M}) \leq w(M_-) + 4n - \sigma\Delta,$$

since the lemma follows by rearranging these inequalities.

The left inequality follows from the 1-feasibility of M, x: $x(\overline{M}) \geq x(M_-) \geq w(M_-) - n$.

To prove the right inequality, observe several inequalities: $x(M) = w(M) \leq x_-(M) + n$, by the 1-feasibility of $\emptyset, x_-$. $x(S) = x_-(S) - \sigma\Delta$, by (2). The 1-optimality of M_- in the previous scale and the Double Step imply $x_-(\overline{M}) \leq x_-(M_-)$, $x_-(M_-) \leq w(M_-) + 3n$. Combining these and using $\overline{M} = M \cup S$ gives the desired inequality:

$$x(\overline{M}) \leq (x_-(M) + n) + (x_-(S) - \sigma\Delta) \leq x_-(M_-) + n - \sigma\Delta \leq w(M_-) + 4n - \sigma\Delta. \qquad \blacksquare$$

Lemma 2.3. There are at most $2\sqrt{5n}+1$ iterations of the loop of *match*.

Proof. Each execution of the Augment Step (except possibly the first) augments along at least one *wap*, because of the preceding Hungarian search. Hence at most $\sqrt{5n}$ iterations start with $\sigma \leq \sqrt{5n}$. From Lemma 2.2, $\sigma \geq \sqrt{5n}$ implies $\Delta \leq \sqrt{5n}$. As already remarked we will show (Lemma 2.7) that each Hungarian search increases Δ by at least one. Thus at most $\sqrt{5n}+1$ iterations start with $\Delta \leq \sqrt{5n}$. ∎

The next property shows the algorithm finds short augmenting paths. Define

$$A = \text{ the total length of all augmenting paths found by } match.$$

Lemma 2.4. $A = O(n \log n)$.

Proof. Consider an augment done by *match*. Let M be the matching before the augment, M' the matching after, P the *wap* and ℓ its length (measured as its number of unmatched pairs). Let wl denote the weight-length function with respect to M. Observe that $P - \overline{M}$ and $P \cap M$ are both sets consisting entirely of pairs. Define the quantity

$$W = wl(P - \overline{M}) - wl(P \cap M).$$

Consider the quantity $\sum W$, where the summation is taken over all *waps* found by *match*. We estimate $\sum W$ in two different ways.

The definition of weight-length implies $W = w(P - \overline{M}) - \ell - w(P \cap M) = w(M') - w(M) - \ell$. Summing gives $\sum W = w(M_n) - w(M_0) - A$, where M_n is the final matching and M_0 is the initial matching. The relations $M_0 = \emptyset$ and $w(M_n) \leq x_-(S_-) + n$ imply our first estimate,

$$\sum W \leq x_-(S_-) + n - A.$$

When the *wap* P is found, let x denote the dual function and Δ denote its current value. Let P contain singletons $s_i \in S_i$, $i = 0, 1$. Since the swaps of P are eligible, $W = x(P - \overline{M}) - x(P \cap M) = x(s_0) + x(s_1)$. Hence by (2),

$$\sum W = x_-(S_-) - \sum \Delta,$$

where the second summation is over the values of Δ for each *wap*.

Combining the two relations for $\sum W$ shows $A \leq n + \sum \Delta$. Lemma 2.2 shows that for the i^{th} *wap* found, $\Delta \leq 5n/(n-i+1)$. Hence $\sum \Delta = O(n \log n)$. This gives the desired bound on A. ∎

Now we discuss the details of the Augment Step. The main issue in implementing the batching of the Augment Step is that each swap of an augment can change the valid swaps. Thus care must be taken to ensure that each *wap* gives a valid augment. We use a "topological numbering" of the elements to guide the search for a *wap*. This numbering is based on an "acyclic" property of swaps that is similar to [GoT, GaT87]. However it seems that on general matroids, the acyclic property cannot be maintained without explicitly maintaining the topological numbering.

Define a (*topological*) *numbering* to be a function $t : E \cup S \rightarrow \mathbf{N}$ ($\mathbf{N}$ denotes the natural numbers) with these properties:

(i) any swap e, f has $t(e) \geq t(f)$;

(ii) any pair e_0, e_1 with $e_i \in E_i$ has $t(e_0) > t(e_1)$ if and only if $e_0, e_1 \notin M$.

(The name is motivated by the fact that t is essentially a topological numbering of the directed graph whose edges are swaps e, f.) Subsequent sections also use a slight extension of this definition, to handle problems other than weighted intersection: A pair e_0, e_1 that is not eligible need not satisfy (ii). This section mentions any slight modifications to the algorithm and proofs needed for this more general definition. The reader may ignore this issue until subsequent sections, if he or she desires.

The definition of topological order implies that t decreases along a *wap* P, in the following sense: for $i = 0$ or 1, if e, f and g, h are swaps in P_i with e, f preceding g, h, then $t(f) > t(g)$. (Note that the algorithm can use a slightly simpler definition of topological number: any swap e, f has $t(e) > t(f)$ and two elements g, h have $t(g) = t(h)$ if and only if g, h is a pair. The resulting algorithm is similar to that given below. The definition used here simplifies the more general intersection algorithms of Section 5.)

The *match routine* maintains a topological numbering (that changes as the valid swaps change). The initial numbering is $t(e) =$ **if** $e \in E_0$ **then** 1 **else** 0. Thereafter t is maintained by the Augment and Search Steps.

Now we give the main property of topological numberings. It enables the algorithm to search for a *wap* without actually executing any swaps. (Recall the definition of "executing a swap" from Section 1). Assume the matching is 1-feasible and there is a valid topological numbering. Let g, h be an arbitrary swap.

Lemma 2.5. Executing a swap g, h does not create or destroy a swap of the form e, f where $t(e) < t(h)$.

Proof. The following notation is convenient. Let e, f be two elements, exactly one of which is

in $\overline{M}$. Write $\{e, f\} = \{b_e, n_e\}$, where $b_e \in \overline{M}$ and $n_e \notin \overline{M}$ ("b" and"n" stand for "basic" and "nonbasic", respectively). Let C_e be the fundamental circuit of n_e immediately before swap g, h is executed. For example this defines b_g, n_g and C_g; furthermore, $b_g \in C_g$.

Suppose that when swap g, h is executed, e, f changes from a valid swap to invalid or vice versa. Both possibilities imply that

$$b_g \in C_e,\ b_e \in C_g,\ x(e) = x(f).$$

By 1-feasibility, $x(b_g) \geq x(n_e)$ and $x(b_e) \geq x(n_g)$. Thus $x(e) = x(f) = x(g) = x(h)$. This implies e, h is a valid swap (before g, h). Hence $t(e) \geq t(h)$, as desired. ∎

We note a useful consequence of this argument. The proof shows that if e, f changes from a valid to invalid swap or vice versa, then e, h is a valid swap immediately before g, h is executed. Similarly g, f is a valid swap immediately before g, h is executed.

The Augment Step uses a variant of the dynamic-base cycle and cocycle problems introduced in [GS]. We state the Augment Step in terms of a strong version of these problems. Section 3 gives two weaker versions of the problems that can be used.

The problems are defined on a matroid where an initial base is given, along with two functions $x,\ t : E \cup S \rightarrow \mathbf{Z}$. (In our application x and t are the dual function and topological numbering, respectively). The *dynamic-base cycle problem* is to process a sequence of intermixed operations $c(e)$ and $update(e, f)$. The operation $update(e, f)$ changes the current base B to $B - e + f$. (Hence the operation requires $e \in B$, $f \notin B$, and $e \in C(f, B)$. In our application *update* is used to execute a swap.) The operation $c(e)$ outputs an element $f \in C(e, B)$, where B is the current base, $e \notin B$, and f is an element with maximum value $t(f)$ subject to the restrictions that $x(f) = x(e)$ and f has not been previously output in any other c operation of the sequence. By definition, a given element f gets output at most once in the entire sequence of operations. If $c(e)$ is executed and no such f exists, the operation returns $\emptyset$. The *dynamic-base cocycle problem* is defined similarly, with operations $update(e, f)$ and $c(e)$ (for element $e \in B$). The operation $c(e)$ still returns the maximum element of $C(e, B)$; the only difference is that now $C(e, B)$ is a cocycle.

The Augment Step does a depth-first search to find a maximal sequence of *waps* P. It uses the following data structures. A stack represents the path P that eventually becomes the *wap*. The dynamic cocycle operations are used on matroid $\mathbf{M}_0$ and dynamic cycle operations on $\mathbf{M}_1$. A variable T gives the currently largest topological number.

The Augment Step works by examining each singleton $s \in S_0$. For each s it initializes path P

to contain s. Then it executes the following steps, until either the Dead-end Step discovers that no *wap* from s exists or the Complete Step augments the matching.

Scan Step. Let e be the last element in P. Set f to the element returned by $c(e)$. (For $e \in E_i$, $c(e)$ is a dynamic cocycle operation if $i = 0$ and a dynamic cycle operation if $i = 1$. In either case $f \in C(e, \overline{M})$.) If $f = \emptyset$ then go to the Dead-end Step. If f is a singleton then add f to the end of P (now P is a *wap*) and go to the Complete Step. Otherwise (the mate $\overline{f}$ exists) if $f, \overline{f}$ is eligible, add $f, \overline{f}$ to the end of P; if $f, \overline{f}$ is not eligible decrease $x(f)$ by one; in both cases go to the Scan Step.

Dead-end Step. If e is a singleton (i.e., $e = s$) stop (no *wap* from s exists). Otherwise (P ends with the pair $\overline{e}, e$) for $g = \overline{e}$ and e, delete g from P and increase $x(g)$ by $\epsilon(g)$; then set $t(e) \leftarrow T$, $t(\overline{e}) \leftarrow T + 1$, $T \leftarrow t(\overline{e})$ and go to the Scan Step.

Complete Step. For each $e \in (P_0 \cap M) \cup (P_1 - \overline{M})$ (i.e., e is not a singleton and is the first element of a swap of P) increase $x(e)$ by $\epsilon(e)$ and set $t(e) \leftarrow t(\overline{e}) + 1$. Augment M along P. (Execute each swap of P using the dynamic *update* routine.) Delete the two singletons of P from S. (This is essentially the procedure of the Augment Step of *match*.) Stop. ∎

A minor addition to the algorithm is needed if we use the extended definition of topological numbering for Section 5: If the Scan Step decreases $x(f)$ it also assigns $t(f)$ and T the value $T + 1$, to ensure property (ii).

Now we prove that this implementation of the Augment Step is correct. The main step is the following lemma, which shows that the algorithm maintains the desired structure. We first make some preliminary observations. Outside of the Complete Step topological numbers decrease along P, in the sense noted above. In the Complete Step topological numbers of pairs $e, \overline{e}$ are set in accordance with the definition. Any element f has its dual changed at most once in the entire Augment Step (since the dynamic c operation outputs f at most once). In part (a) below we say that a dual variable "has changed" if it has changed during the Augment Step. Conditions $(1a)$ and $(1b)$ are clearly preserved by Augment Step: The Scan Step only decreases a dual if $(1a)$ is satisfied with strict inequality; the Dead-end Step does not change any sum $x(e, \overline{e})$; the Complete Step makes $x(e, \overline{e}) = wl(e, \overline{e})$ (after the augment) for any pair in P.

Lemma 2.6. The following properties hold throughout the Augment Step.

(a) If e, f is a valid swap and $x(f)$ has changed, then $x(e)$ has changed.

(b) The dominance property ($1c$) always holds.

(c) The topological numbering is valid.

(d) Any augment results in a valid matching.

Proof. We prove the assertions by induction on the number of steps executed. We consider a Scan Step, Dead-end Step and Complete Step in turn.

A Scan Step can only decrease the dual of an element in $E_0 - \overline{M}$. Clearly this preserves (a) – (d).

Consider a Dead-end Step. We must verify parts (a) – (c). The Dead-end Step removes the pair $\overline{e}, e$ from P, where $\overline{e}$ is the second element of a swap of P. First consider $\overline{e}$. After $x(\overline{e})$ is changed, no valid swap involves $\overline{e}$. Thus (a) and (c) are vacuous for swaps involving $\overline{e}$, and dominance (b) holds for $\overline{e}$.

Next consider e. We first show that immediately before $x(e)$ is changed, there are no valid swaps e, f. For the sake of contradiction assume e, f is valid. Since $c(e)$ returned $\emptyset$ in the Dead-end Step, f has already been output. Since e is the last element in P and $t(e) \geq t(f)$, f is not in P. Hence either f was never added to P or f was removed from P. In the first case, the Scan Step changes $x(f)$. In the second case, the Dead-end or Complete Step changes $x(f)$. (If f was removed in a Complete Step that augmented along a *wap* Q, f changed from matched to unmatched or vice versa, and so f was first in a swap of Q.) Hence (a) (which is true by induction) implies that $x(e)$ has already changed. This is the desired contradiction.

The preceding observation implies that when the Dead-end Step changes $x(e)$, dominance (b) is preserved. Since $t(e)$ is changed to the largest topological number (c) holds. Obviously the conclusion of (a) becomes true. (N.B. The change to $x(e)$ can create valid swaps e, f: swaps that became invalid when $x(f)$ was changed in the Augment Step, or entirely new swaps if $x(f)$ was never changed.) We conclude the Dead-end Step works as desired.

Now consider a Complete Step. It is convenient to assume that the Complete Step processes the swaps of P in order. More precisely, the swaps g, h of P are executed in left-to-right order, and the dual $x(g)$ is changed immediately after the swap.

We first verify (d). This amounts to showing that after executing a swap g, h in P, any subsequent swap e, f in P is still valid. This follows from Lemma 2.5.

Next we verify (a) and (c). Let e, f be a swap that is valid after swap g, h is executed. We

first show (a) and (c) in the case that $\{e,f\} \cap \{g,h\} = \emptyset$. Clearly we can assume that e,f is a swap that gets created when g,h is executed. From Lemma 2.5 $t(e) \geq t(h)$, and as noted after the proof, before g,h is executed swaps e,h and g,f are valid.

Since g,f is valid and $x(g)$ has not changed, (a) shows that $x(f)$ has not changed. This implies that (a) is vacuous for e,f. It also implies that f has not been processed in a Scan, Dead-end or Complete Step (for the latter note that f would have been the first element of a swap). Thus when the dynamic operation $c(g)$ outputs h, f has not been output. Furthermore swap g,f is valid at that time. (To see this observe that the only swaps that are executed between when h is output and when g,h is executed are the swaps k,ℓ preceding g,h in P. We can assume $t(\ell) > t(g)$, so these swaps do not affect the validity of g,f, by Lemma 2.5.) The definition of the dynamic c operation now shows $t(h) \geq t(f)$. Also since e,h is valid, $t(e) \geq t(h)$. These inequalities show $t(e) \geq t(f)$, as desired for (c).

Now we show (a) and (c) in the second case, when a new swap involves g or h. The Complete Step changes $x(g)$ so that g is not in any valid swap. So consider a new swap h,k. Observe that before g,h is executed, k is in g's fundamental circuit (cocircuit). Hence g,k is a swap. Since $x(g)$ has not changed, (a) shows that $x(k)$ has not changed. This implies that (a) is vacuous for swap h,k. It also implies that $t(h) \geq t(k)$ (examining the dynamic operation $c(g)$ as above). This gives (c).

It remains to verify (b). It suffices to consider an element e whose fundamental cycle is changed by the swap g,h. The argument is similar to the above and so is omitted. ∎

Lemma 2.7. Each Hungarian search increases Δ by at least one.

Proof. It suffices to show that at the end of an Augment Step, no remaining singleton s has a valid swap s,f. We first show this holds when the Dead-end Step deletes s from P. At this point the dynamic operation $c(s)$ returns $\emptyset$. Hence if there is a valid swap s,f, then f has been output. Hence $x(f)$ has changed in the Augment Step. Thus (a) implies that $x(s)$ has changed, a contradiction.

A valid swap s,f is not created in a Scan or Dead-end Step, which can only decrease $x(f)$. Similarly the remark after Lemma 2.5 shows that s,f is not created in a Complete Step (if executing swap g,h creates s,f, then s,h is valid before the swap). ∎

This concludes the proof that the Augment Step is correct. We turn to the Hungarian search (in the Search Step). It grows a forest $\mathcal{F}$ of singletons, eligible pairs and unmatched elements of

E_0. $\mathcal{F}$ has similar structure to the search forest of Frank's algorithm, with one main change: A pair $e, \overline{e}$ is in $\mathcal{F}$ only if it is eligible. Thus if $e, \overline{e}$ is unmatched and ineligible, it is possible that $e \in \mathcal{F}_0$ and $\overline{e} \notin \mathcal{F}$.

The search grows a maximal search forest $\mathcal{F}$. If $\mathcal{F}$ cannot be enlarged and yet no *wap* has been found, a dual adjustment is done. This step calculates a dual adjustment quantity δ; then it increases each dual $x(e)$, $e \in \mathcal{F}$, by $\epsilon(e)\delta$. δ is chosen so that the new dual function is 1-feasible and has a new valid swap or eligible pair, so $\mathcal{F}$ can be enlarged. More precisely to define δ first define a quantity $\delta(f)$ for elements $f \notin \mathcal{F}$: For $f \in (E_0 - M - \mathcal{F}) \cup (M_1 - \mathcal{F})$, $\delta(f) = \min\{|x(e) - x(f)| \mid e \in C(f, \overline{M}) \cap \mathcal{F}\}$. For $f \in E_1 - M - \mathcal{F}$ and $\overline{f} \in \mathcal{F}$, $\delta(f) = x(f, \overline{f}) - w(f, \overline{f}) + 1$. Define δ as the smallest of all these quantities $\delta(f)$.

The Hungarian search alternates between grow steps and dual adjustments. Eventually the desired *wap* is found.

In general the difficulty in implementing the Hungarian search is performing the dual adjustment. This leads to more elaborate organizations for the Hungarian search (see [GX]). However for linear matroids the Augment Step seems to dominate the time; this is true for the algorithms of this paper. Hence the above simple organization for the Hungarian search suffices for our purposes.

It remains to specify how the Hungarian search maintains the topological numbering. Suppose the search has a valid numbering t, and then it does a dual adjustment. We define the new numbering by specifying the relative order of all elements. Let $\mathcal{F}$ be the search forest before the dual adjustment. For any element e let $s(e) =$ (**if** $e \in \mathcal{F}$ **then** 1 **else** 0). Then the new ordering is determined by lexical order of the values $(s(e), t(e))$.

To see this rule is correct, first note that property (ii) of topological numberings holds: If a pair e_0, e_1 has $s(e_0) = s(e_1)$, (ii) follows from the same property for t. If $s(e_0) \neq s(e_1)$, then $e_0, e_1 \notin M$, $s(e_0) = 1$ and $s(e_1) = 0$, so (ii) again holds. To show property (i), consider a swap e, f that is valid after the dual adjustment. If it became valid in the dual adjustment then $x(e)$ changed but $x(f)$ did not, whence $s(e) = 1$, $s(f) = 0$ and (i) holds. On the other hand if e, f was a swap before the dual adjustment, (i) follows from the same property for t, unless $s(e) = 0$ and $s(f) = 1$. In this case $x(f)$ changes but $x(e)$ does not. But then e, f is not valid after the adjustment. Thus (i) always holds.

To implement the rule efficiently we extend it to the entire Hungarian search. Let t be the topological numbering at the start of a Hungarian search. Suppose the search does k dual adjustments. Assign each element e a value $s(e)$ as follows. If e is in the search forest $\mathcal{F}$ before the i^{th} dual adjustment but not earlier, then $s(e) = -i$ (e.g., a singleton $e \in S_0$ has $s(e) = -1$). Otherwise

$s(e) = -k - 1$. Then it is easy to see that at the end of the Hungarian search the topological order is given by lexical order of the values $(s(e), t(e))$. A radix sort at the end of the Hungarian search can construct a list of all elements in the new topological order, in $O(m)$ time.

We close the discussion of the Hungarian search with some further implementation details. We show how to calculate δ and how to adjust duals (by $\epsilon(e)\delta$). For the latter we use offsets, as follows. When an element e is added to $\mathcal{F}$, its current dual value and the current value of Δ are saved as $x^0(e)$ and $\Delta^0(e)$, respectively. Then at any point in the Hungarian search the current value of $x(e)$ can be calculated as

$$x(e) = x^0(e) + (\Delta - \Delta^0(e))\epsilon(e).$$

Hence all duals can be appropriately changed by simply increasing the value of Δ. At the end of the Hungarian search all the duals $x(e)$ are changed to their correct value (using the above formula) in $O(m)$ time.

Next we show how to calculate δ. Lemma 2.2 shows that $\Delta \leq 5n$. Each Hungarian search maintains an array $Q[1..5n]$. Each entry $Q[d]$ points to a list of pairs of elements p, c. For each such pair, c can be added to $\mathcal{F}$ as the child of p if Δ reaches the value d and c is still not in $\mathcal{F}$. To find the next dual adjustment, the algorithm scans down Q and chooses Δ as the smallest value whose list $Q[\Delta]$ gives a grow step.

Pairs are added to Q as follows. Assume that routines are available to compute fundamental cycles and cocycles. Specifically for a given base B and an element e, a routine $cyc(e)$ returns the elements of $C(e, B)$, which is the fundamental cycle of e for $e \notin B$ and the fundamental cocycle for $e \in B$. When a grow step adds an element e to $\mathcal{F}$, the routine $cyc(e)$ is used to find the elements f of its fundamental cycle or cocycle. For each such f not in $\mathcal{F}$, the quantity $d = |x^0(e) - x(f)| + \Delta^0(e)$ is calculated. If $d \leq 5n$ then the pair e, f is added to $Q[d]$. A similar calculation is done when a grow step adds an element e but not $\bar{e}$ to $\mathcal{F}_0$. (The space for this data structure is $O(mn)$. This is easily reduced to $O(m)$: For each element c, maintain only the entry for c that is on the list $Q[d]$ with smallest index d. This is easy to do if the lists of Q are doubly-linked.)

To calculate the time associated with Q, note that one Hungarian search adds $O(nm)$ pairs to Q: an unmatched element can be in n pairs and a matched element (or singleton of S_1) can be in m pairs. The algorithm spends $O(1)$ time on each pair. In addition it uses $O(n)$ time scanning down the Q array. Thus aside from the time for cyc routines, the total time is $O(nm)$.

Finally we discuss maintaining the topological numbering in the Augment Step. An array $t[1..m]$ is used to store topological numbers. This allows topological numbers $t(e)$ to be accessed

and changed in $O(1)$ time. Note that if all topological numbers are at most m at the start of an Augment Step, they are at most $2m$ at the end.

We conclude this section by estimating the time to find a maximum perfect matching. First consider Augment Steps. Let t_d denote the time for all dynamic operations in all Augment Steps of one scale. It is easy to see that the time for an Augment Step is dominated by the time for the dynamic operations. Hence the total time for Augment Steps in one scale is $O(t_d)$. t_d can be described more precisely as follows. There are $O(\sqrt{n})$ different sequences of dynamic operations in all Augment Steps. Each sequence does $O(m)$ c operations: each operation $c(e)$ either outputs an element or is the last operation for e in the sequence. The sequences collectively contain $O(n \log n)$ *update* operations. More precisely after the d^{th} sequence, the remaining *updates* occur as $O(n/d)$ subsequences (each subsequence corresponding to a *wap*). This follows from Lemma 2.2, since $\Delta \geq d$ after the d^{th} sequence.

Next consider Search Steps. Let t_h denote the time to execute one Hungarian search. Clearly the time for all Search Steps in one scale is $O(\sqrt{n}t_h)$. For linear matroids a good estimate of t_h is the following. Let t_{cyc} denote the time to execute a sequence of *cyc* operations, possibly outputting every element's fundamental cycle or cocycle. Then

$$t_h = O(nm + t_{cyc}).$$

This follows since calculating dual adjustments and processing Q takes $O(nm)$ time.

We give two time bounds for the entire algorithm. The first is for general matroids (for applications, see [GX]). The second is oriented to linear matroids, and is applied in Section 4.

Theorem 2.1. The weighted matroid intersection problem can be solved in time $O((\sqrt{n}(m + t_h) + t_d)\log(nN))$. The time is also $O((\sqrt{n}(nm + t_{cyc}) + t_d)\log(nN))$.

Proof. First consider maximum perfect matching. There are $\log(nN)$ scales. In each scale various operations use time $O(m)$. (For example the Double Step of the main routine uses time $O(m)$: To create singletons, we can assume that every element of E is given with a copy, which can be used as a singleton.) In each scale the *match* routine does $O(\sqrt{n})$ Augment and Search Steps. The above discussion shows that the time bound of the Theorem applies to these steps.

Next consider the other variants of weighted matching, maximum weight maximum cardinality matching, and maximum weight matching. They are solved using the reductions at the start of this section. ∎

3. Cardinality Intersection Algorithm and Other Extensions.

This section discusses the specialization of the weighted intersection algorithm to maximum cardinality matching. The cardinality algorithm is given and anlayzed. Properties that make it more efficient are discussed. Most of these properties are proved in the context of the weighted intersection algorithm. Some of the properties are applied elsewhere [GX] for efficient weighted intersection algorithms on other matroids.

We begin by stating the algorithm for maximum cardinality matching. It is a specialization of the weighted intersection algorithm. The given weight function w is taken to be the zero function. Scaling is not needed for cardinality matching — the algorithm works in just one scale. It initializes the singleton set S to an arbitrary base of $\mathbf{M}$, and the dual function x to the zero function. Then it calls *match*. The *match routine* works as before except for the halting criterion: It halts when the Hungarian search fails to find a *wap*.

This algorithm is correct for two reasons. First note that the initialization makes the empty matching 1-feasible when *match* is called, as desired. Second, the halting criterion is correct, i.e., *match* returns a maximum cardinality matching. This follows from a standard duality argument: Suppose the algorithm halts with matching M after growing a search forest $\mathcal{F}$. Any unmatched pair e_0, e_1 with $e_i \in E_i$ is either in $\mathcal{F}$, in which case $e_1 \in sp(M_1 \cap \mathcal{F})$, or not in $\mathcal{F}$, in which case $e_0 \in sp(M_0 - \mathcal{F})$. Thus any matching has at most $|M_1 \cap \mathcal{F}| + |M_0 - \mathcal{F}| = |M|$ elements, as desired.

The timing analysis of this algorithm follows Section 2. First define

$$\begin{aligned} \mu &= \text{ the cardinality of a maximum matching;} \\ r &= \text{ the rank of } \mathbf{M}_0; \\ \sigma &= |S_0| - (r - \mu). \end{aligned}$$

Since $\mu \leq r \leq n$, the following result is stronger than Lemma 2.2.

Lemma 3.1. At any point in the execution of *match*, $\sigma\Delta \leq \mu$.

Proof. Let M be the current matching, S the current singleton set, and M_- a maximum cardinality matching. Add elements of $\overline{M}$ to M_- to get a base B of $\mathbf{M}$. Then $x(\overline{M}) = x(S) = -|S_0|\Delta$. Also $x(\overline{M}) \geq x(B) \geq -\mu - (r - \mu)\Delta$ (the last inequality follows since in each matroid $\mathbf{M}_i$ the singletons have the smallest dual value). Combining gives the desired inequality. ∎

The proofs of the remaining lemmas of Section 2 are essentially unchanged. In Lemmas 2.3 and 2.4, "n" can be changed to "μ". This gives an analog of Theorem 2.1 for cardinality matching,

with time bounds $O(\sqrt{\mu}(m + t_h) + t_d)$ and $O(\sqrt{\mu}(nm + t_{cyc}) + t_d)$. The algorithm can be made even more efficient because of three properties related to the dynamic operations, which we now derive.

Lemma 3.2. In any *wap* found in the cardinality matching algorithm, each swap g, h has a distinct dual value $x(g)$.

Proof. Consider a *wap* P found at a certain value of Δ. If g, h is a swap of P_0 then the next swap of P has dual value $-x(g) - 1$, and the following swap (if it exists) has dual value $x(g) + 1$. Hence (2) of Section 2 implies that the dual values of the swaps of P form the sequence $x, -x - 1,\ x = -\Delta, \ldots, -1$. Note that the last swap is in $\mathbf{M}_1$, with dual value zero. There are no swaps after it, since singletons are the only elements of $\overline{M}_1$ with dual value zero. This follows from the way the Complete Step changed duals. The lemma follows. ∎

Observe that the proof of the lemma implies the function $n+\epsilon x$ is a valid topological numbering. (Here we use the extended definition of topological numbers, i.e., an ineligible pair need not satisfy (ii).) If this numbering were used, the algorithm would behave just as if there were no topological numbers at all. We conclude that the algorithm works correctly without topological numbers.

This gives the first simplification of the algorithm: Topological numbers are not maintained. The dynamic operation $c(e)$ is defined to output an element $f \in C(e, B)$ that has not been previously output and has $x(f) = x(e)$.

The rest of this section derives properties of the weighted matching algorithm; it will be obvious that the properties also hold for the cardinality matching algorithm. These properties combined with Lemma 3.2 give an efficient cardinality matching algorithm in Section 4. The weighted matching algorithm of Section 4 could be improved by using these properties, if one could develop an analog of Lemma 3.2.

The main property is similar to Lemma 2.5: the portion of $C(e, \overline{M})$ that is relevant for the operation $c(e)$ is changed by an operation $update(g, h)$ only if $x(g) = x(e)$. (A similar property is used in [GS85] for cardinality matching.) The next lemma proves a more general statement. For any set of elements A and a set of integers S, define

$$A_S = \{a | a \in A \text{ and } x(a) \in S \text{ at the start of the Augment Step}\}.$$

Consider a sequence of swaps executed in the Augment Step that changes the matching from M to N. M and N can be chosen at any point (even in the middle of a Complete Step, if desired).

Consider an interval $[\ell..h]$. Let B be the result of starting with base $\overline{M}$ and executing only the swaps with dual value in $[\ell..h]$, i.e., $B = \overline{M}_{(-\infty..\ell)\cup(h..\infty)} \cup \overline{N}_{[\ell..h]}$.

Lemma 3.3. *B is a base. For any element $e \in E_{[\ell..h]}$, $C(e,\overline{N})_{[\ell..h]} = C(e,B)_{[\ell..h]}$.*

Proof. The first part of the lemma, the fact that B is a base, follows from the second part, by an easy induction. Hence it suffices to prove the second part.

Observe that the dual function at the start of the Augment Step is dominating throughout the Augment Step. This follows from Lemma 2.6(a). In the rest of the argument "dominance" refers to this dominance of the original dual function.

Note that for any i, $sp(\overline{M}_{[i..\infty)}) = sp(\overline{N}_{[i..\infty)})$. This follows by an easy induction on the number of swaps, using dominance.

Now consider elements $e, f \in E_{[\ell..h]}$. By dominance, $f \in C(e,\overline{N})$ if and only if $f \in C(e,\overline{M}_{(-\infty..\ell)} \cup \overline{N}_{[\ell..\infty)})$. Since $sp(\overline{M}_{(h..\infty)}) = sp(\overline{N}_{(h..\infty)})$ the latter is true if and only if $f \in C(e,\overline{M}_{(-\infty..\ell)\cup(h..\infty)} \cup \overline{N}_{[\ell..h]})$. The base in the last expression is B, so the lemma follows. ∎

The lemma allows the dynamic operations to be simplified, as follows: At the start of each Augment Step, partition the elements into the sets E_x (here E_x stands for the notation $E_{\{x\}}$). Each such partition maintains a copy of the base, B^x; this copy is modified by an operation $update(e,f)$ only if $e, f \in E_x$. Thus each *wap* executes one update in each set E_x, by Lemma 3.2. The dynamic operation $c(e)$, where $x(e) = x$, outputs an element $f \in C(e,B^x)_x$. Correctness of this modification follows from the lemma with $\ell = h = x$.

The lemma allows us to avoid executing swaps during one Augment Step. We now generalize to avoid executing swaps during a sequence of Augment and Search Steps. Consider a sequence of swaps changing M to N, that extends over a number of Augment and Search Steps. We construct the base B as follows. In the above definition of A_S, use dual values $x(a)$ at the start of the current Augment Step. Choose an initial interval $[\ell..h]$ so that $h - \ell \geq 2(\Delta_N - \Delta_M)$, where Δ_M is the value of Δ when the matching is M and similarly for Δ_N. After a Search Step that increases Δ by δ, assign $\ell \leftarrow \ell + \delta$, $h \leftarrow h - \delta$. In the Augment Step, modify B by executing the swaps with (current) dual value in the (current) interval $[\ell..h]$.

In the corollary below, all quantities are evaluated at the current point in the algorithm.

Corollary 3.1. B is a base. For any element $e \in E_{[\ell..h]}$, $C(e, \overline{N})_{[\ell..h]} = C(e, B)_{[\ell..h]}$.

Proof. As in the lemma it suffices to prove the second part. We do this by induction (on the number of steps executed in the algorithm). The inductive hypothesis makes two assertions in addition to the second part of the lemma: (i) $\overline{N}_{[\ell..h]} = B_{[\ell..h]}$; (ii) $sp(\overline{N}_{(h..\infty)}) = sp(B_{(h..\infty)})$.

Assume the inductive hypothesis holds after some Augment Step. Suppose the next Hungarian Search adjusts duals by δ. Let ℓ and h denote their value after they have been modified by δ. We first show that (i) – (ii) hold at the start of the next Augment Step.

Equation (i) holds since before the Hungarian search changed duals, $\overline{N}_{[\ell-\delta..h+\delta]} = B_{[\ell-\delta..h+\delta]}$. Furthermore, a dual value in $[\ell..h]$ at the start of the Augment Step was previously in $[\ell-\delta..h+\delta]$. (Even though the Augment Step changes duals, it is easy to see that no dual changes by more than δ from the start of one Augment Step to the next).

Equation (ii) holds since before the Hungarian search $sp(\overline{N}_{(h+\delta..\infty)}) = sp(B_{(h+\delta..\infty)})$ and $\overline{N}_{(h-\delta..h+\delta]} = B_{(h-\delta..h+\delta]}$, by induction. Furthermore the set $E_{(h..\infty)}$ at the start of the Augment Step consists of the previous set $E_{(h+\delta..\infty)}$ plus a subset of the previous set $E_{(h-\delta..h+\delta]}$.

It is easy to see (i) – (ii) are preserved throughout the Augment Step. These equations imply that for any $i \in [\ell..h+1]$, $sp(\overline{N}_{[i..\infty)}) = sp(B_{[i..\infty)})$.

Now consider elements $e, f \in E_{[\ell..h]}$. Observe that $f \in C(e, \overline{N})$ if and only if $f \in C(e, B_{(-\infty..\ell)} \cup \overline{N}_{[\ell..\infty)})$. This follows from dominance and the fact that $sp(\overline{N}_{[\ell..\infty)}) = sp(B_{[\ell..\infty)})$. Since $sp(B_{(h..\infty)}) = sp(\overline{N}_{(h..\infty)})$ the last membership holds if and only if $f \in C(e, B_{(-\infty..\ell)\cup(h..\infty)} \cup \overline{N}_{[\ell..h]})$. The base in the last expression is B, so the corollary follows. ∎

The corollary allows us to execute a Hungarian search knowing only a portion of each element's fundamental cycle or cocycle, as follows. Define a *period* of the algorithm to be the time when $\Delta_0 \leq \Delta \leq \Delta_1$, for some values Δ_0, Δ_1. At the start of this period the algorithm computes all fundamental cycles and cocycles. During the period the algorithm maintains only a portion of each fundamental cycle and cocyle. Specifically for an element e not initially in the base, it maintains the cycle $C(e, \overline{M})_{[\ell..h]}$ for $\ell = x(e) - (\Delta_1 - \Delta)$, $h = x(e) + (\Delta_1 - \Delta)$. This can be done by executing only the swaps with dual value in $[\ell..h]$, by Corollary 3.1. Fundamental cocycles are similar. Note that as long as a Hungarian search keeps Δ within the period, the *cyc* operations can be executed correctly, knowing just the above subsets of the fundamental cycles and cocycles.

We now summarize the time for the cardinality matching algorithm. Define t_d and t_h as in Theorem 2.1. However the dynamic operations, and consequently t_d, can be simplified: There are no topological numbers. If desired, an Augment Step can partition the elements into sets E_x, on

which only *update* operations for swaps with dual value x are executed. Alternatively if desired, portions of the algorithm can be declared periods, where again, only a subset of *update* operations need be executed on each element.

Theorem 3.1. The cardinality matroid intersection problem can be solved in time $O(\sqrt{\mu}(m + t_h) + t_d)$. The time is also $O(\sqrt{\mu}(nm + t_{cyc}) + t_d)$.

We close this section by noting that Lemma 3.3 can be used to modify the weighted matching algorithm and make it more efficient on other matroids, such as graphic matroids [GX]. This modification does not seem to be useful for linear matroids, but we state it here for completeness. The idea is that because of the lemma, the dynamic routines can be executed on matroids of rank less than n. To define these matroids we first remove elements that cannot be in a *wap*. More precisely let M denote the matching at the start of the Augment Step. The proof of the lemma shows that any element e in a *wap* is not in $sp(\overline{M}_{(x(e)..\infty)})$. Thus we can restrict attention to the elements

$$F = \{e | e \notin sp(\overline{M}_{(x(e)..\infty)}) \text{ and either } e \in S \text{ or } \overline{e} \notin sp(\overline{M}_{(x(\overline{e})..\infty)}) \text{ and } e, \overline{e} \text{ is eligible}\}.$$

Now define a matroid $\mathbf{N}$ as follows. For a set of elements A with $i = 0$ or 1 and x an integer, abbreviate the notation $(A_i)_{\{x\}}$ to $A_{i,x}$. $\mathbf{N}$ is the direct sum of matroids $\mathbf{N}_i$, $i = 0, 1$. Each $\mathbf{N}_i$ is itself the direct sum of matroids $\mathbf{N}_{i,x}$, where x ranges over all distinct dual values $x(e)$ of elements $e \in \overline{M}_i$, and

$$\mathbf{N}_{i,x} = (\overline{M}_i \cup F_{i,x})/(\overline{M}_i - F_{i,x}).$$

$\mathbf{N}$ uses the pairing relation of $\mathbf{M}$.

Consider the following implementation of the Augment Step. It starts by constructing matroid $\mathbf{N}$. It searches for *waps* and augments the matching using the same three step procedure given above, working on matroid $\mathbf{N}$. It ends by using the final matching on $\mathbf{M}$ and discarding $\mathbf{N}$.

Such an execution of the Augment Step on $\mathbf{N}$ corresponds to an execution on $\mathbf{M}$. This follows from the lemma: B_x is a base of $\mathbf{N}_{i,x}$, and swaps with respect to the current base of $\mathbf{N}$ correspond to swaps with respect to the current base of $\mathbf{M}$. Hence the modified Augment Step is correct.

4. Linear Matroids.

This section starts by reviewing relevant ideas from linear algebra. Then it gives implementations of our intersection algorithm on linear matroids, first weighted intersection and then cardinality.

A *representation* of a linear matroid is an $n \times m$ matrix, where each matrix column corresponds to a matroid element [W]. Assume that the input to the algorithm is a representation $\mathbf{A}$ of the matroid $\mathbf{M}$.

In a (*standard*) *representation with respect to base* B, the columns for the elements of B form a permutation matrix. The element corresponding to the r^{th} column of the identity is called the r^{th} *basic element.* Let B be a base. Let $\mathbf{B}$ denote the columns of $\mathbf{A}$ corresponding to B, in some order. Then $\mathbf{B}^{-1}\mathbf{A}$ is the representation with respect to B where the r^{th} basic element corresponds to the r^{th} column of $\mathbf{B}$. From this representation it is easy to compute fundamental cycles and cocyles with respect to B: they are determined by the rows and columns of the representation, respectively.

We are interested in efficiently computing fundamental cycles and cocycles when the base is repeatedly changed by swaps. Consider a swap that replaces e by f. Let $\mathbf{A}_-$ be a representation with respect to the base before the swap, in which e is the r^{th} basic element, and f has column vector with entries f_i. If f becomes the r^{th} basic element, the new representation is $\eta\mathbf{A}_-$. Here η is the $n \times n$ identity matrix with r^{th} column replaced by the values

$$\eta_{ir} = (\textbf{if } i = r \textbf{ then } 1/f_r \textbf{ else } -f_i/f_r), \quad 1 \leq i \leq n.$$

Such a matrix η is called an *eta matrix* in linear programming [Ch].

Suppose we start with a base having representation $\mathbf{A}_-$, and execute a sequence of s swaps. Let the corresponding eta matrices be η_i, $i = 1, \ldots, s$ (η_i is calculated immediately before the i^{th} swap). The new representation is

$$\eta_s \ldots \eta_1 \mathbf{A}_-.$$

The following lemma summarizes how to compute this new representation efficiently. To achieve the given bounds we store an eta matrix in $n + 1$ words: the index r and the values η_{ir}, $1 \leq i \leq n$. We store a representation $\mathbf{A}$ in mn words as a two-dimensional array. This allows each element to be accessed in time $O(1)$, given its indices.

Lemma 4.1. Consider a product $\mathbf{A}' = \eta_s \ldots \eta_1 \mathbf{A}$, where each η_i is an $n \times n$ eta matrix and $\mathbf{A}$ is an $n \times m$ matrix, $n \leq m$.

(a) One column of $\mathbf{A}'$ can be computed in time $O(sn)$.

(b) One row of $\mathbf{A}'$ can be computed in time $O(sm)$.

(c) For $s \leq n$, $\mathbf{A}'$ can be computed in time $O(mns^{\omega})$.

(d) For t indices i (each index in $1 \leq i \leq s$) the t products $\eta_i \ldots \eta_1 \mathbf{A}$ can be computed in time $O(mns^{\omega}t^{1-\omega})$. This bound holds even if the sequence of indices is given on-line in increasing order, i.e., each product is computed before the next higher index is known.

Proof. (a) To calculate one column, associate the multiplications to the right. This gives $O(n)$ time to multiply one eta matrix. The total time is $O(sn)$ as desired.

(b) To calculate the r^{th} row, let $\mathbf{r}$ denote the r^{th} row of $\eta_s \ldots \eta_1$, so we seek $\mathbf{rA}$. Compute $\mathbf{r}$ by associating the multiplications to the left. Hence $\mathbf{r}$ is found in time $O(sn)$. Note that $\mathbf{r}$ has at most $s+1$ nonzero entries, one in column r and one in the column of each eta matrix. Hence entry column of $\mathbf{rA}$ can be found in time $O(s)$. The total time to compute the row is $O(sm)$, as desired.

(c) First compute the product $\mathbf{E} = \eta_s \ldots \eta_1$, and then compute the desired product $\mathbf{EA}$. We discuss the latter first.

Let C denote the indices of the columns corresponding to the s eta matrices. As noted above we can write

$$\mathbf{E} = \mathbf{I} + \mathbf{S}. \tag{1}$$

Here as usual $\mathbf{I}$ is the identity matrix and $\mathbf{S}$ is a matrix that is zero except in the columns of C. This shows the desired product can be found in time $O(mn)$ from the product $\mathbf{SA}$. Let $\mathbf{S}'$ denote the $n \times s$ matrix consisting of the columns for C in $\mathbf{S}$; let $\mathbf{A}'$ denote the $s \times m$ matrix consisting of the rows for C in $\mathbf{A}$. Clearly $\mathbf{SA} = \mathbf{S}'\mathbf{A}'$. Since $s \leq n \leq m$ the latter multiplication can be done by decomposing $\mathbf{S}'$ into $\lceil \frac{n}{s} \rceil$ $s \times s$ matrices, $\mathbf{A}'$ into $\lceil \frac{m}{s} \rceil$ $s \times s$ matrices, and performing $O(mn/s^2)$ multiplications of $s \times s$ matrices. This gives the desired time.

It remains to compute the product $\mathbf{E}$. Suppose we use naive matrix multiplication, associating to the left. The time is $O(sn)$ for one multiplication, by (1). This gives $O(s^2 n)$ time total. This bound will often be within the desired time bound. If not the time can be improved to $O(ns^{1+\omega})$, less than the desired bound. This is done using the following divide-and-conquer scheme.

Consider first the product of two $n \times n$ matrices, each having at most k nonzero columns (the columns can be different in the two matrices). The product can be found in time $O(nk^{1+\omega})$. This

is because it amounts to multiplying matrices of dimension $n \times k$ and $k \times k$. The latter can can be done as $\lceil \frac{n}{k} \rceil$ multiplications of $k \times k$ matrices, giving the desired bound.

To compute $\mathbf{E}$, assume for convenience that s is a power of two. The algorithm recursively computes $\mathbf{E}_1$, the product of the first $s/2$ matrices, and $\mathbf{E}_2$, the product of the remaining matrices. Now we seek the desired product $\mathbf{E} = \mathbf{E}_1\mathbf{E}_2$. Applying (1) to each matrix $\mathbf{E}_i$ shows that computing the product amounts multiplying two matrices of $s/2$ nonzero columns each. As already noted this is done in time $O(ns^{1+\omega})$. Hence the time for the entire computation is given by the recurrence $t(s) = 2t(s/2) + ns^{1+\omega}$. This implies $t(s) = O(ns^{1+\omega})$ as desired.

(*d*) We use the algorithm of part (*c*) to compute each product. Let the j^{th} product include s_j more eta matrices than the $j-1^{st}$, $1 \leq j \leq t$. Hence $\sum_{j=1}^{t} s_j = s$, and the total time is proportional to $\sum_{j=1}^{t} mns_j^{\omega}$. Since $\omega \leq 1$ the sum is maximized when each s_j equals s/t. This gives the desired bound. ∎

Now we present the algorithms for matroid intersection. They all implement the algorithms of Sections 2–3. Recall that to completely specify these algorithms we need only give implementations of the dynamic routines and the *cyc* routine; the time is given by Theorem 2.1.

We start with two algorithms for weighted intersection. The first algorithm is not asymptotically fastest but has the practical advantage of using only naive matrix multiplication.

The first algorithm implements the dynamic and *cyc* routines by simply maintaining a representation with respect to the current matching $\overline{M}$. An *update* operation multiplies the representation by the eta matrix corresponding to the swap. A *c* or *cyc* operation scans a column, or row, of the representation to find the desired cycle, or cocycle, respectively. (A *c* operation scans the column or row in decreasing topological number.)

To estimate the time for this algorithm recall Theorem 2.1. The dynamic time $t_d = O(mn^2 \log n)$. It is dominated by the time for $O(n \log n)$ *update* operations, since the matrix multiplication for one swap uses time $O(mn)$ (Lemma 4.1(*a*)). The dynamic *c* operations use less time $O(n^{1.5}m)$, since each Augment Step scans the entire representation in time $O(mn)$. Similarly for the Hungarian Search, $t_{cyc} = O(mn)$. (This includes $O(m)$ time to sort the elements of the matching by topological number, as needed for the dynamic *c* operations.) This gives the following result.

Theorem 4.1. The weighted matroid intersection problem on a linear matroid can be solved in time $O(mn^2 \log n \log(nN))$ (using only naive matrix multiplication). ∎

The theorem generalizes Cunningham's bound for cardinality matching on linear matroids, $O(mn^2 \log n)$ [Cu]. In fact, the time bound of the theorem can be achieved by a simpler version of the first algorithm. This algorithm does not use topological numbers, but rather always augments along a *wap* of minimal length (i.e., a *wap* without shortcuts). This simplified algorithm is very much like Cunningham's cardinality algorithm. Further details are left to the reader.

The second algorithm for maximum perfect matching uses fast matrix multiplication. We present the algorithm in terms of two integral parameters d and s, to be specified later.

The second algorithm works as follows. It computes the representation for the current base at various points, using Lemma 4.1(d). These points are (i) after every Augment Step; (ii) when $\Delta > d$, after every augment; (iii) when $\Delta \leq d$, after every s swaps.

Because of (i), the *cyc* routine amounts to scanning the appropriate row or column of the representation. A similar remark holds for the dynamic c routine when $\Delta > d$, because of (ii). When $\Delta \leq d$ the routine $c(e)$ computes e's fundamental cycle or cocycle, by computing the appropriate column or row of the current representation. It does this using Lemma 4.1(a–b) to compute the product of the last representation matrix $\mathbf{A}$ and the eta matrices for the subsequent swaps (by (iii) there are at most s eta matrices).

Theorem 4.2. The weighted matroid intersection problem on a linear matroid can be solved in time $O(mn^{1.77} \log(nN))$. More precisely the time is $O(mn^{1+(2/(3-\omega))} \log^{(1+\omega)/(3-\omega)} n \log(nN))$.

Proof. The time for computing the representations in (i)–(iii) is

$$O(mn^{1+\omega} \log^{\omega} n(\sqrt{n} + (n/d) + (n \log n)/s)^{1-\omega}).$$

This follows from Lemma 4.1(d) and these facts: There are $O(\sqrt{n})$ Augment Steps; $O(n/d)$ *waps* are found when $\Delta > d$; there are $O(n \log n)$ swaps in total. The time to compute all fundamental cycles and cocycles in c routines when $\Delta \leq d$ is, by Lemma 4.1(a–b),

$$O(dsmn).$$

To balance these terms choose $s = (n \log^2 n)^{1/(3-\omega)}$ and $d = s/\log n$. This gives the desired bound. (Note that $\omega < 1$ implies that $1/(3-\omega) < 1/2$, so $s = o(\sqrt{n})$ and $\sqrt{n} = o(n/d)$. Also note that for $\omega = 0.4$ our choice makes s and d about $n^{0.38}$.)

It is usually nontrivial to compute the values s and d. In this case observe that it suffices to use values s' and d', where $s/8 \leq s' \leq s$ and similarly for d'. Such a value s' can be found by computing *log* s to within 3, which is easily done; d' is similar. ∎

We turn to cardinality matching. It is convenient to define

$$\alpha = 1/(2-\omega).$$

(For $\omega = 0.4$, $\alpha = 0.625$; in general $1/2 \leq \alpha \leq 1$.) As in Section 3 μ denotes the cardinality of a maximum matching. In general μ is unknown at the start of the algorithm. Let $\overline{\mu}$ be a known upper bound on μ (we can take $\overline{\mu} = n$ but a better bound may be available; this is illustrated in Corollary 5.2). The algorithm has two phases which we now describe.

The first phase is when $\Delta \leq \overline{\mu}^{1-\alpha}$. The first phase computes the representation for the current base after every Augment Step. Thus as in the previous algorithm, the *cyc* routine amounts to scanning the appropriate row or column of the representation.

Now consider the Augment Step for the first phase. It is based on Lemma 3.3, which allows it to partition the elements into sets E_x on which only *update* operations for swaps with dual value x are executed. Define $\overline{\Delta} = \max\{\Delta, 1\}$ and

$$s = (\overline{\mu}/2^{2\lfloor \log \overline{\Delta} \rfloor})^{\alpha}.$$

(Note that $s \geq 1$ since the definition of first phase implies $s \geq \overline{\mu}^{(2\alpha-1)\alpha}$ and $\alpha \geq 1/2$.) Associated with each dual value x is a matrix $\mathbf{A}_x$, that has a row (column) for each element of $E_x \cap \overline{M}$ ($E_x - \overline{M}$). The algorithm updates each $\mathbf{A}_x$ after every s *waps*, using Lemma 4.1(c) on the last matrix $\mathbf{A}_x$ and the subsequent eta matrices corresponding to swaps with dual value x. Lemma 3.2 implies there are precisely s such eta matrices. The routine $c(e)$ computes the appropriate column or row for e using Lemma 4.1(a–b) on the last matrix $\mathbf{A}_{x(e)}$ and the subsequent eta matrices for swaps with dual value $x(e)$. Again there are at most s of these.

The second phase is when $\Delta > \overline{\mu}^{1-\alpha}$. It is based on Corollary 3.1, which allows portions of the algorithm to be declared periods, where only a subset of *update* operations need be executed on each element. Define

$$r = 2^{\lceil \log \Delta \rceil}/\overline{\mu}^{1-\alpha}.$$

(Note that $r \geq 1$ by the definition of the second phase.) The second phase is divided into periods corresponding to the intervals $ir \leq \Delta < (i+1)r$ for integers i. At the start of each period the algorithm computes the representation for the current base. During a period the algorithm maintains only a portion of the expansion of each element $e \notin \overline{M}$ with respect to the current base $\overline{M}$. Specifically it maintains the coefficients in e's expansion for all elements $f \in \overline{M}$ such that $|x(e) - x(f)| \leq (i+1)r - \Delta$. To do this, after augmenting along any *wap*, the expansion of each

$e \notin \overline{M}$ is updated by multiplying by the eta matrices for swaps with dual value in the above range. Lemma 3.2 implies there are at most r such swaps. The $cyc(e)$ and $c(e)$ routines work by scanning the appropriate column or row for e. Corollary 3.1 implies that the relevant matrix entries are correct.

Theorem 4.3. A maximum cardinality matching on a linear matroid can be found in time $O(mn^{1.62})$. More precisely the time is $O(mn\overline{\mu}^{1/(2-\omega)} \log \mu)$.

Proof. Correctness of the algorithm follows from the discussion. Now we show the desired time bound, which equals $O(mn\overline{\mu}^{\alpha} \log \mu)$. It is useful to note that

$$\omega\alpha = 2\alpha - 1.$$

For a later application involving matroid duality (Corollary 5.2(c)) it is useful to observe the time is dominated by the dynamic and *cyc* routines, and our timing analysis depends on only two facts about the time for linear algebra operations, both consequences of Lemma 4.1: Lemma 4.1(a–b), summarized by saying that the time to compute every row (column) of a representation once during a sequence s swaps is $O(smn)$; and Lemma 4.1(c). Note that Lemma 4.1(d) is implied by Lemma 4.1(c).

We analyze the two phases separately. Consider the first phase, and start with the computation of matrices $\mathbf{A}_x$. Fix a value of s; let Δ be the smallest value corresponding to s. The total time for computing all matrices $\mathbf{A}_x$ for this value s is

$$O(mn\overline{\mu}^{\alpha}/\overline{\Delta}^{\omega\alpha}). \tag{2}$$

To show this note that by Lemma 4.1(c), an $\mathbf{A}_x$ matrix with n_x rows and m_x columns is updated once in time $O(m_x n_x s^{\omega})$. (Lemma 3.2 ensures that $n_x \geq s$.) Summing over all x gives time $O(mns^{\omega})$ to update all $\mathbf{A}_x$ matrices once. There are at most $\sigma \leq \mu/\overline{\Delta}$ *waps* for the chosen value of s (Lemma 3.1). Hence there are at most $\overline{\mu}/(\overline{\Delta}s)$ updates to the $\mathbf{A}_x$ matrices. This implies the time for all updates is $O(mn\overline{\mu}/(\overline{\Delta}s^{1-\omega}))$. (Note $\overline{\mu} \geq \overline{\Delta}s$ since $s \leq \overline{\mu}^{\alpha}$.) By definition, $\overline{\Delta}s^{1-\omega} = \overline{\mu}^{1-\alpha}\overline{\Delta}^{\omega\alpha}$. This gives (2).

The quantities (2) (for distinct values of s) form a geometric progression with ratio $(1/2)^{\omega\alpha} < 1$. Hence the sum is dominated by the first term, when $\overline{\Delta} = 1$. Thus the total contribution of (2) for the entire phase is $O(mn\overline{\mu}^{\alpha})$, less than the desired bound.

Next consider the $c(e)$ routine. Again fix a value of s and let Δ be the first value corresponding to s. The total time for all operations $c(e)$ for this value s is (2). To show this note that there are

at most $2\overline{\Delta}$ Augment Steps for s. Each Augment Step can execute $c(e)$ for each element e. Hence Lemma 4.1(a–b) gives total time $O(smn\overline{\Delta})$. (Recall there are at most s eta matrices for each $c(e)$. These estimates are valid since $s \geq 1$.) This expression simplifies to (2). Hence the time is again $O(mn\overline{\mu}^{\alpha})$.

Finally consider the time to compute the representation after every Augment Step. There are $O(\overline{\mu}^{1-\alpha})$ Augment Steps in the first phase and hence that many representations. We now show that the second phase computes more representations, whence the time is dominated by the second phase.

Consider the second phase. Start with the time to compute representations. Fix a value of r; let Δ be the first value corresponding to r. The number of representations computed for this value r is at most $\overline{\mu}^{1-\alpha}$. This follows since there are at most $\Delta/r = \overline{\mu}^{1-\alpha}$ periods for r. We conclude that the second phase computes $O(\overline{\mu}^{1-\alpha} \log \mu)$ representations. (Note this is more than the number in the first phase). Now Lemma 4.1(d) shows that the total time to compute representations is $O(mn\mu^{\omega}\overline{\mu}^{(1-\alpha)(1-\omega)} \log \mu) = O(mn\overline{\mu}^{\alpha} \log \mu)$, which is the desired bound.

Next consider maintaining the coefficients in expansions during a period. Again fix a value of r and let Δ be the first value corresponding to r. The total time to maintain expansions for this value r is $O(mn\overline{\mu}^{\alpha})$. To show this note that there are at most μ/Δ *waps* for r. The time to update expansions after a *wap* is $O(rmn)$ by Lemma 4.1(a–b). (This estimate is valid since $r \geq 1$). Hence the total time for r is $O(rmn\mu/\Delta) = O(mn\overline{\mu}\alpha)$, as desired. Thus the total time for all distinct values of r is $O(mn^{1+\alpha} \log \mu)$, the desired bound.

Finally note that as in Theorem 4.2, it is usually nontrivial to compute $\overline{\mu}^{1-\alpha}$, s and r. As in that theorem it is easy to compute values that are c times the desired value, for c varying between $1/2$ and 1. Using these values in the algorithm gives the same asymptotic time bound. (For instance in the geometric progressions for the first phase, each term for the actual time is at most a constant times the exact value.) ∎

5. Generalized Intersection Problems and Applications.

This section extends previous algorithms to more general matroid intersection problems, including the independent assignment and linking problems. It concludes by applying our algorithms to a class of problems arising in control theory and numerical computation.

We begin with an observation about matroid duality that can make our algorithm more efficient. In an intersection problem (weighted or cardinality) denote one of the matroids $\mathbf{M}_i$ as $\mathbf{N}$, and

let $\mathbf{N}^*$ be its matroid dual. If desired the algorithm can work on $\mathbf{N}^*$ rather than $\mathbf{N}$. In proof, let B be a base of $\mathbf{N}$ and B^* its complementary base of $\mathbf{N}^*$. Any element e has $C(e, B, \mathbf{N}) = C(e, B^*, \mathbf{N}^*)$. Hence the dynamic routines on $\mathbf{N}$ can be implemented by the dynamic routines on $\mathbf{N}^*$. Specifically the algorithm maintains B^*, the complement of the base of $\mathbf{N}$; an *update* operation on $\mathbf{N}$ is executed as the same operation on $\mathbf{N}^*$; the dynamic cycle (cocycle) problem on $\mathbf{N}$ is the dynamic cocycle (cycle) problem on $\mathbf{N}^*$. The *cyc* routine can be similarly implemented on $\mathbf{N}^*$.

To illustrate, suppose a matroid intersection problem has $\mathbf{M}_0$ given in a standard representation as an $n \times m$ matrix. The dual $\mathbf{M}_0^*$ is represented by an easily calculated $(m-n) \times m$ matrix [W]. If $m - n < n$ it is more efficient to implement the algorithms of Section 4 on $\mathbf{M}_0^*$ rather than $\mathbf{M}_0$. This principle is used in Corollary 5.2(c) below.

Now consider the independent matching problem. It generalizes matroid intersection by allowing the pairing function to be given by a bipartite graph. More precisely, consider a bipartite graph G with vertex sets E_i, $i = 0, 1$ and edges E. Matroids $\mathbf{M}_i$ are defined on E_i, and matroid $\mathbf{M}$ is the direct sum $\mathbf{M}_0 + \mathbf{M}_1$. An (*independent*) *matching* is a matching on G whose endpoints are independent in $\mathbf{M}$. A *maximum cardinality* (*independent*) *matching* has as many edges as possible. If there is a weight function $w : E \to \mathbf{R}$ then *maximum perfect matching*, *maximum weight maximum cardinality matching*, and *maximum weight matching* are defined as for matroid intersection. The *independent assignment problem* refers to any of these variants of weighted matching.

Other definitions follow by analogy with matroid intersection. To state resource bounds, let $\mathbf{M}$ have rank r and n elements; let $|E| = m$. Assume that all weights are integers of magnitude at most N.

We use a straightforward reduction of independent matching on $\mathbf{M}$ to intersection on a matroid $\mathbf{N}$. Construct $\mathbf{N}$ as follows. Replace each edge $vw \in E$ by a pair v', w' where v' and w' are parallel copies of v and w respectively. Thus $\mathbf{N}$ is the same as $\mathbf{M}$ except it has parallel copies of elements. For a vertex v of $\mathbf{M}$, let E_v denote the set of all elements of $\mathbf{N}$ that are parallel copies of v. We use obvious notation for $\mathbf{N}$ derived from notation for $\mathbf{M}$, e.g., $\mathbf{N} = \mathbf{N}_0 + \mathbf{N}_1$. Clearly a set of edges is a matching on $\mathbf{M}$ if and only if the corresponding set of pairs is a matching on $\mathbf{N}$. Thus an independent matching problem can be solved by the algorithm for the corresponding intersection problem. This holds for all variants of the problems considered in this paper, both weighted and unweighted.

We solve the intersection problem on $\mathbf{N}$ using the algorithms of Sections 2–3. Note that each matroid $\mathbf{N}_i$ has m elements. We modify the intersection algorithm so the time for each iteration of *match* is as if the matroids had only n elements, plus $O(m)$ overhead.

There are three main changes in *match*. First, it uses the extended definition of topological numbering given in Section 2: An ineligible pair need not satisfy property (ii) of the definition of topological numbering.

For the second change it is convenient to extend the algorithm's dual function and topological numbering to the vertices v of $\mathbf{M}$. Let μ_i denote *max* if $i = 0$ and *min* if $i = 1$. Define $x(v) = \max\{x(e)|e \in E_v\}$, and for $v \in E_i$, $t(v) = \mu_i\{t(e)|e \in E_v \text{ and } x(e) = x(v)\}$.

Immediately before each Augment Step and Search Step, the algorithm "normalizes" values as follows.

Normalize Step. For each vertex v and each $e \in E_v$ set $x(e) \leftarrow x(v)$ and $t(e) \leftarrow t(v)$. ∎

This normalization takes $O(m)$ time, which is within the desired bound. Now observe that normalization does not destroy any relations needed by the algorithm: It does not change the values of a matched element. It gives 1-feasible duals and a valid topological numbering (for the latter, note that property (ii) of topological numbering need only hold for eligible pairs). It may change a pair from eligible to ineligible. However it does not destroy any *wap* found by a Hungarian search, since no dual of an element in a valid swap is changed. It does not create a *wap* after an Augment Step, since it does not change a singleton's dual (recall Lemma 2.7).

By normalization, the Augment and Search Steps can assume every copy of an element has the same values. This allows these steps to be implemented efficiently. This is the third and final change, which we now describe.

Consider the Augment Step. Suppose we are given implementations of the dynamic operations on $\mathbf{M}$. We adapt them to $\mathbf{N}$ as follows: The *update* routines are unchanged. Consider a dynamic operation $c(e)$, where e is a copy of v in $\mathbf{N}_i$. If $i = 1$ (dynamic cycle), $c(e)$ executes $c(v)$ unless another copy of v has done so; in the latter case $c(e)$ returns $\emptyset$. If $i = 0$ (dynamic cocycle), $c(e)$ executes $c(v)$ in the following sense. Suppose repeated calls to $c(v)$ would output the sequence of elements $f_1, \ldots, f_k$. Then $c(e)$ outputs the sequence of unmatched elements in E_w, for $w = v, f_1, \ldots, f_k$ (elements within a set E_w can be output in arbitrary order).

To see the dynamic c operations are correct, first consider $\mathbf{N}_1$. Suppose a number of operations $c(e)$ are done, but not all elements of $C(e, \overline{M})$ are output (here M is the current matching); then later on $c(f)$ is done, where $e, f \in E_v$. We must check it is correct for $c(f)$ to return $\emptyset$. Clearly e was in a *wap*. When $c(f)$ is executed, $C(f, \overline{M}) = \{e, f\}$ and $x(e) > x(f)$. Hence $\emptyset$ is the correct output.

Next consider $\mathbf{N}_0$. Normalization implies that the sequence of copy elements output in c operations is correct. We must verify another detail: Suppose operations $c(e)$ are done, and some but not all copies of an element $v \in C(e, \overline{M})$ are output. We must check it is correct for no subsequent c operation to output a copy of v. The copy $g \in E_v$ output last was in a *wap*. Hence any $f \in E_v$ has $C(f, \overline{M}) = \{g, f\}$. Thus such elements f can be ignored in subsequent c operations, as desired.

The time for an Augment Step is as desired: If t_d is the time to execute the dynamic operations corresponding to all Augment Steps of one scale on $\mathbf{M}$, the total time used by the algorithm on $\mathbf{N}$ is $O(t_d + m)$.

Next consider the Hungarian search. By normalization, an element $e \in E_v$ can be added to the search forest $\mathcal{F}$ precisely when all elements of E_v can be added. Furthermore the routine $cyc(e)$ need only be executed for one copy of a given element v (the first copy added to $\mathcal{F}$). Thus it is easy to see that a Hungarian search can be implemented in time $O(m)$ plus the time for the corresponding search on $\mathbf{M}$.

We now summarize the results. The following theorem uses the notation of Theorem 2.1. In the corollary μ denotes the cardinality of a maximum matching.

Theorem 5.1. The independent assignment problem can be solved in time $O((\sqrt{r}(m + t_h) + t_d)\log(rN))$. ∎

Corollary 5.1. (*a*) The linear independent assignment problem can be solved in time $O((nr^{1.77} + \sqrt{r}m)\log(rN))$, or alternatively time $O((nr^2 \log r + \sqrt{r}m)\log(rN))$ using only naive matrix multiplication.

(*b*) The linear cardinality independent matching problem can be solved in time $O(nr^{1.62} + \sqrt{\mu}m)$ using fast matrix multiplication. ∎

The algorithm generalizes to matroid versions of various network flow problems. We describe a representative problem, linking; other problems are similar. Given is a directed graph $G = (V, E)$. The vertices are partitioned into sets V_i, $i = 0, 1, 2$. For $i = 0, 1$, matroids $\mathbf{M}_i$ are defined on ground sets V_i. No edge goes from V_1 to V_0. A *Menger linking* is a set $\mathcal{L}$ of vertex-disjoint simple paths from V_0 to V_1. A *linking* is defined similarly, except the paths are edge-disjoint and may intersect in vertices of V_2. The *endpoints* of a linking are the vertices of V_i, $i = 0, 1$ on paths. Linking $\mathcal{L}$ is *independent* if its endpoints are independent in $\mathbf{M}_i$, $i = 0, 1$. A linking has *maximum cardinality*

if it has as many paths as possible; it is *perfect* if its endpoints are a base of $\mathbf{M}_i$, $i = 0, 1$. If each edge e has a cost $c(e)$, the *cost* of a linking is the total cost of all its edges. A *minimum perfect linking* is a perfect linking with minimum possible cost. Other terminology follows by analogy with matching. The *weighted independent linking problem* is to find a minimum perfect linking, minimum cost maximum cardinality linking, or minimum cost linking.

Let the total rank of the two matroids $\mathbf{M}_i$ be r, and let $|V| = n$, $|E| = m$. Assume all costs are integers at most N in magnitude. Furthermore assume that G has no negative cycles.

An independent linking problem on G reduces to an independent matching problem on a bipartite graph B as follows. For a vertex v, let $d(v)$ be its total degree in G. Modify graph G to G': For each $v \in V_2$, add $d(v)$ loops, i.e., edges vv, of cost zero. Let $E_0(v)$ be the set of edges directed from v in G' and $E_1(v)$ the set of edges directed to v in G' (each loop is in both these sets). For each $v \in V_2$ define uniform matroids $\mathbf{M}_{vi}$, $i = 0, 1$, to have elements $E_i(v)$ and rank $d(v)$. Define a bipartite graph B: Its vertex set is the disjoint union of the ground sets of all matroids $\mathbf{M}_{vi}$, $\mathbf{M}_0$ and $\mathbf{M}_1$. Its edges are formed by joining the two copies of every edge of G'. Edges in B have the same cost as in G'. Define the matroid $(\mathbf{M}_0 + \sum \mathbf{M}_{v0}) + (\mathbf{M}_1 + \sum \mathbf{M}_{v1})$ on the vertices of B.

A perfect independent linking on G corresponds to a perfect independent matching on B. Thus a minimum perfect linking can be found using this reduction. (This depends on the fact that there are no negative cycles). Similarly a minimum weight maximum cardinality linking and a minimum weight linking can be found. (For these problems, use the reduction at the start of Section 2 to transform a matching problem into perfect matching. However, instead of creating a copy of every element of the matroid, only create copies of the elements of $\mathbf{M}_i$, $i = 0, 1$.)

A maximum cardinality linking can also be found using this reduction, as follows. A matching on B corresponds to a linking on G if it contains a base of every matroid $\mathbf{M}_{vi}$. To find such a matching, of maximum cardinality, execute *match* on B, initializing the matching M to contain all edges corresponding to loops vv. Since $sp(M)$ is increasing in *match*, the final matching has the desired property, since the initial matching does.

To compute the time for linking, first observe that the dynamic operations are trivial for a uniform matroid $\mathbf{U}$: Let x^* denote the smallest dual value of an element of the base of $\mathbf{U}$. In a valid swap both elements have dual value x^*. The only data structure needed for dynamic cocycle operations is a list of elements not in the base, having dual x^*, that have not been output, ordered on topological number; for dynamic cycle a similar list of elements in the base is used. A dynamic *update* operation uses time $O(1)$.

This shows that in one Augment Step, the time for all dynamic c operations on uniform matroids $\mathbf{M}_{vi}$ is $O(m)$. Similarly in one Hungarian search the time for all *cyc* operations on these matroids is $O(m)$. Thus Theorem 2.1 shows that the time to solve a weighted independent linking problem is

$$O((\sqrt{m}(m + t_h) + t_d)\log(mN)).$$

Here t_d and t_h are the time for dynamic operations and Hungarian search, respectively, in matroids $\mathbf{M}_i$, $i = 0, 1$ (these quantities are described more completely in Theorem 2.1). The cardinality independent linking problem can be solved in time

$$O(\sqrt{m}(m + t_h) + t_d).$$

Here t_d and t_h are as described in Theorem 3.1. The weighted and cardinality independent Menger linking problems have the similar bounds, $O((\sqrt{n}(m + t_h) + t_d)\log(nN))$ and $O(\sqrt{n}(m + t_h) + t_d)$, respectively.

We illustrate these results by considering a matroid partition problem. Recall that if $\mathbf{M}_i$, $i = 0, 1$ are matroids on the same ground set, their *sum* $\mathbf{M}_0 \vee \mathbf{M}_1$ is a matroid whose independent sets I are those that can be written $I = I_0 \cup I_1$ with I_i independent in $\mathbf{M}_i$. The *matroid partition problem* is to find a base of $\mathbf{M}_0 \vee \mathbf{M}_1$.

Next consider a linear matroid $\mathbf{T}$ represented by a matrix whose nonzero entries are distinct variables (equivalently, the nonzero entries are distinct and algebraically independent); columns of $\mathbf{T}$ are interpreted as vectors over the field of rational numbers extended by these variables. It is well-known that such a matroid is a transversal matroid [W]. Specifically, suppose $\mathbf{T}$ is represented by a matrix A with row set R, column set C, and all nonzero entries algebraically independent. Form the bipartite graph $B(\mathbf{T})$ having vertex sets R and C and edges $\{rc | a_{rc} \neq 0\}$. A set of columns is independent in $\mathbf{T}$ if the corresponding vertices can be covered by a matching in $B(\mathbf{T})$.

Define the *mixed linear partition problem* to be a matroid partition problem where $\mathbf{M}_i$, $i = 0, 1$ are linear matroids, with $\mathbf{M}_0$ represented as a matrix of integers and $\mathbf{M}_1$ a transversal matroid represented like $\mathbf{T}$ above.

It is most convenient and efficient to solve such a problem by incorporating the transversal matroid into the graph of a matching or linking problem. In this instance we reduce the mixed linear partition to a matching problem as follows. Let the two matroids $\mathbf{M}_i$ have the same column set E; let $\mathbf{M}_1$ have row set R. Form a bipartite graph G with vertex sets $V_0 = E$, $V_1 = E \cup R$ (strictly speaking sets V_i contain two distinct copies of E). G has edges $\{ee | e \in E\} \cup B(\mathbf{M}_1)$. Define the free matroid on ground set V_0, the free matroid $\mathbf{F}$ on R, and the matroid $\mathbf{M}_0 + \mathbf{F}$ on V_1.

It is easy to see that independent sets of $\mathbf{M}_0 \vee \mathbf{M}_1$ and independent matchings on G correspond with each other. Hence the mixed linear partition problem can be solved using the algorithm for linear cardinality independent matching (and the above routines for uniform matroids). To compute the time assume each matroid $\mathbf{M}_i$ has at most m columns and n rows. A maximum cardinality matching on G has at most $2n$ elements, and G has at most $m(n+1)$ edges. Thus Corollary 5.1(b) shows the time is $O(mn^{1.62})$.

The mixed linear partition problem and related problems have applications in control theory and numerical calculation. We state our results and briefly discuss the problems below; further details are in the references cited.

Corollary 5.2. (a) The rank of an $m \times n$ mixed matrix ($m \geq n$) [MIN] can be found in time $O(mn^{1.62})$.

(b) The combinatorial canonical form of an $m \times n$ layered mixed matrix ($m \geq n$) [MIN] can be found in time $O(mn^{1.62})$.

(c) The structural controllability of a descriptor system with m input variables and n internal variables [M87b] can be tested in time $O((m+n)n^{1.62})$.

(d) The dynamic degree of a descriptor system system with n internal variables [M87a] can be found in time $O(n^{2.77})$, or alternatively $O(n^3 \log^2 n)$ using only naive matrix multiplication.

Proof. (a) In a *mixed matrix* each entry is an integer or a variable that occurs only once. The rank problem reduces to a mixed linear partition problem. The problem can be used, e.g., to test the unique solvability of an electrical network [MIN]. It is also investigated in [SVY].

(b) This canonical form generalizes LU-decomposition and Dulmage-Mendelsohn decomposition. It can be used to efficiently solve a system of linear equations with varying coefficients. [MIN] reduces the canonical form problem to calculations that are dominated by a linear cardinality independent matching problem. Inspection shows that the latter is a mixed linear partition problem.

(c) [M87b] reduces the controllability problem to calculations that are dominated by two mixed linear partition problems and a cardinality independent Menger linking problem. Inspection shows that the latter is a cardinality independent matching problem on a graph with structure similar to the above G. The main difference is that the matroid on V_0 has rank n, while the matroid on V_1 has $m+2n$ elements and rank $m+n$, being the dual of a matroid of $m+2n$ elements and rank n. To solve this matching problem we use our cardinality algorithm with upper bound $\overline{\mu} = n$ on the size of a matching; we implement the dynamic routines on V_1 using dynamic routines for

the dual. Theorem 4.3 shows that in this case the first term in the estimate of Corollary 5.1(b) is $O((m+n)n^{1.62})$, as desired.

(d) The dynamic degree gives the number of degrees of freedom of the system. [M87a] reduces the dynamic degree problem to a maximum weight maximum cardinality independent Menger linking problem. Inspection shows this problem amounts to weighted independent matching on a graph similar to G. Also the weights are integers in the range $[0..n+1]$ (this follows from [M87a, p. 145, (18.13); note the range is independent of m). The dynamic degree generalizes the order of complexity of an electrical network. The latter is also found by a weighted independent matching problem, but on a graphic matroid [I]. ∎

Previous bounds for these problems are time $O(mn^2 \log n)$ for (a), using an extension of [Cu], and similar expressions for (b) and (c); time $O(n^4)$ for (d), using [F]. Most of our improvements are at present only theoretic, since they use matrix multiplication routines that have very large hidden constants [CW]. However we believe that the $O(n^3 \log^2 n)$ time algorithm of part (d) would be simple and practical to implement, based on experience with related scaling algorithms [GaT87].

References.

[A] M. Aigner, *Combinatorial Theory*, Springer-Verlag, New York, 1979.

[Ch] V. Chvátal, *Linear Programming*, W.H. Freeman and Co., New York, 1983.

[Cu] W.H. Cunningham, "Improved bounds for matroid partition and intersection algorithms", *SIAM J. Comput., 15*, 4, 1986, pp. 948–957.

[CW] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Proc.* 19^{th} *Annual ACM Symp. on Theory of Comp.*, 1987, pp. 1–6.

[D] G.B. Dantzig, *Linear Programming and Extensions,* Princeton Univ. Press, Princeton, N.J., 1963.

[E] J. Edmonds, "Minimum partition of a matroid into independent subsets", *J. Res. National Bureau of Standards* 69B, 1965, pp. 67-72.

[F] A. Frank, "A weighted matroid intersection algorithm", *J. Algorithms 2,* 1981, pp. 328-336.

[FT] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *J. ACM 34*, 3, 1987, pp. 596–615.

[G] H.N. Gabow, "Scaling algorithms for network problems", *J. of Comp. and Sys. Sciences 31*, 2, 1985, pp. 148-168.

[GS85] H.N. Gabow and M. Stallmann, "Efficient algorithms for graphic matroid intersection and parity", *Automata, Languages and Programming:* 12^{th} *Colloquium, Lecture Notes in Computer Science 194*, W. Brauer, ed., Springer-Verlag, 1985, pp. 210–220.

[GS86] H.N. Gabow and M. Stallmann, "An augmenting path algorithm for the parity problem on linear matroids", *Combinatorica 6,* 2, 1986, pp. 123–150.

[GX] H.N. Gabow and Y. Xu, "Efficient algorithms for independent assignment on graphic and other matroids", manuscript in preparation.

[GaT87] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for network problems", *SIAM J. Comp.*, to appear.

[GaT88] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for general graph matching problems", unpublished manuscript.

[GoT] A.V. Goldberg and R.E. Tarjan, "Solving minimum-cost flow problems by successive approximation", *Proc.* 19^{th} *Annual ACM Symp. on Th. of Computing*, 1987, pp. 7–18.

[I] M. Iri, "Applications of matroid theory", *Mathematical Programming — The State of the Art*, A. Bachem, M. Grötschel and B. Korte, eds., Springer-Verlag, New York, 1983, pp. 158–201.

[K] D.E. Knuth, "Matroid partitioning", Tech. Rept. STAN-CS-73-342, Comp. Sci. Dept., Stanford Univ., Stanford Calif., 1973.

[L] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart and Winston, New York, 1976.

[M87a] K. Murota, *Systems Analysis by Graphs and Matroids: Structural Solvability and Controllability,* Algorithmics and Combinatorics 3, Springer-Verlag, New York, 1987.

[M87b] K. Murota, "Refined study on structural controllability of descriptor systems by means of matroids", *SIAM J. Control and Optimization 25*, 4, 1987, pp. 967–989.

[MIN] K. Murota, M. Iri and M. Nakamura, "Combinatorial canonical form of layered mixed matrices and its applications to block-triangularization of systmes of linear/nonlinear equations", *SIAM J. Alg. Disc. Meth. 8*, 1, 1987, pp. 123–149.

[SVY] P.W. Shor, V.V. Vazirani and M. Yannakakis, "Testing singularity of a class of matrices by linking matching and Gaussian elimination", manuscript.

[T] R.E. Tarjan, *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA., 1983.

[W] D.J.A. Welsh, *Matroid Theory,* Academic Press, New York, 1976.