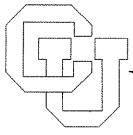
Efficient Algorithms for Independent Assignment on Graphic Matroids

Harold N. Gabow and Ying Xu

CU-CS-468-90 March 1990



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Efficient Algorithms for Independent Assignment on Graphic Matroids

Harold N. Gabow and Ying Xu

CU-CS-468-90 March 1990

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

Efficient algorithms are presented for the matroid intersection problem and its generalizations on graphic matroids. For weighted intersection, a strongly-polynomial algorithm runs in time $O(n(m+n\log n)\log n)$ and a scaling algorithm runs in time $O(\sqrt{n}\,(m+n\log n)\log n\log(nN))$ (n and m denote the number of vertices and edges, respectively; the bound involving N assumes weights are integers of magnitude at most N). For maximum cardinality intersection, an implementation of the scaling algorithm runs in time $O(\sqrt{n}\,(m+n\log n))$. Each of these bounds is within a factor of $\log^2 n$ of the best corresponding bound for trivial matroids, i.e., ordinary bipartite graph matching.

1. Introduction.

The matroid intersection problem is interesting theoretically as a generalization of bipartite graph matching [L]. Its many practical applications include the analysis of continuous systems that model elastic structures and chemical processing plants [I, Mu]; graphic matroid intersection is used to analyze electrical networks [I]. This paper presents several efficient algorithms for graphic matroid intersection.

The starting point is our previous paper [GX], which presents an algorithm for weighted matroid intersection (and its generalizations like independent assignment) on arbitrary matroids. The algorithm uses cost scaling. The algorithm can be used to find a maximum cardinality intersection; it becomes more efficient by taking advantage of greater structure in the problem.

This paper presents efficient implementations of the general matroid algorithm for the case of graphic matroids. In doing so it gives two new data structures that may have further applications, the heavy path representation of a graph and the connecting tree. (For example [G] uses the heavy path representation to improve several known algorithms.) The algorithms for graphic matroids all have a running time that is within a factor $\log^2 n$ of the best corresponding bound for trivial matroids, i.e., ordinary bipartite graph matching. (An abbreviated and preliminary version of this paper appears in [GXa]. The results in this paper are asymptotically superior.)

Before summarizing these algorithms, recall the time bounds for ordinary matching on a bipartite graph. If we seek a maximum cardinality matching, the best-known algorithm of Hopcroft and Karp runs in time $O(\sqrt{n}m)$ [HK]. If we seek a minimum cost matching, the cost-scaling algorithm of [GT89] achieves almost the same time, $O(\sqrt{n}m\log(nN))$; the best-known strongly polynomial bound is $O(n(m + n\log n))$ [FT]. Here and in all other bounds in this paper, n and m denote the number of vertices and edges, respectively; N denotes the largest magnitude of a cost. When N is used it is assumed that the given costs are integral.

Consider the problem of finding a maximum cardinality intersection on a graphic matroid. It arises in determining if an electrical network is uniquely solvable in a general sense. There is a large literature on this application, e.g., see [RI] and the survey paper [I]. Our time bound is $O(\sqrt{n}(m+n\log n))$, at worst a logarithmic factor more than the Hopcroft-Karp bound for ordinary matching. (Ordinary matching is a simple special case of our problem.) For planar graphs the time is the same as [HK], $O(n^{3/2})$.

Previous work on graphic cardinality intersection is in [GS]. For graphs with $m = \Omega(n^{3/2}\log n)$ they achieve the bound of [HK]. For sparser graphs the running time degrades: for $m = \Omega(n\log n)$ and $O(n^{3/2}\log n)$, the bound is $O(nm^{2/3}\log^{1/3} n)$, and a similar bound otherwise. For planar

graphs they achieve time $O(n^{3/2} \log n)$.

The graphic weighted intersection problem is used in determining the order of complexity of an electrical network [I] and also the unique solvability of open networks [R]. The best previous bound for this problem is achieved by Frank's general matroid algorithm [Fra] and many equivalent algorithms. The time for an implementation of Frank's algorithm is $O(n^2 m)$. Our techniques implement Frank's algorithm in time $O(n(m + n \log n) \log n)$. This is the best strongly polynomial bound for graphic weighted intersection; it is a factor $\log n$ more than the bound of [FT] for ordinary graph matching. We also present a scaling algorithm that achieves time $O(\sqrt{n}(m + n \log n) \log n \log(nN))$. This is within a factor of $\log^2 n$ of the bound of [GT89] for ordinary graph matching.

The rest of this section gives some terminology and notation. Section 2 briefly summarizes the general matroid weighted intersection algorithm. Sections 3 – 4 implement the general cost-scaling algorithm on graphic matroids. Section 3 implements the Search Step of the algorithm; it also gives the strongly polynomial bound for graphic intersection. Section 4 implements the Augment Step of the algorithm, and completes the analysis of the cost-scaling algorithm. Section 5 presents the algorithms for graphic cardinality intersection. The reader interested in graph theory may prefer to skip Section 2 and refer back to it only as needed.

Our algorithms use a version of the dynamic tree data structure. In addition to the usual operations *link*(e), *cut*(e) [ST, T83] each node of the dynamic tree has a real-valued key; each key can be changed arbitrarily, and the minimum or maximum key in a path, tree or subtree can be found. Goldberg, Grigoriadis and Tarjan [GGT] show that how to implement each of these operations in amortized time $O(\log n)$ for an n node tree. We refer to this data structure as a *minimizing* (*maximizing*) *dynamic tree.*

Let T be a graph. We often identify T with its set of vertices or edges, e.g., for vertex v or edge e, we write $v \in T$ or $e \in T$. Usually in this paper this notation is used for T a tree.

For the basic notions of matroids see [L, W]. Let $\mathbf{M}$ be a matroid over a set of elements E (for a graphic matroid, E is the set of edges of a graph G). Let e be an element and B a base (B is a spanning tree of G). The notation $C_B(e)$ denotes the *fundamental circuit* of e in B if $e \notin B$, or the *fundamental cocircuit* of e for B if $e \in B$ (fundamental cycle or cocycle of edge e). It will always be clear from context whether or not $e \in B$. When B is clear from context we write $C(e)$.

Consider a matroid $\mathbf{M}$ that is a direct sum of $\mathbf{M}_0$ and $\mathbf{M}_1$, where $\mathbf{M}_i$ is a matroid on elements E_i, $i = 0, 1$. (In graphic matroids we have graphs G_0 and G_1 with disjoint vertex sets.) Suppose $E_0 \cup E_1$ is partitioned into blocks of size two called *pairs*, each pair containing one element from each

E_i. If e is an element, $\overline{e}$ denotes the other element in its pair; we write a pair as $e, \overline{e}$. A *matching* M is an independent set of pairs. (In a graphic matroid an independent set is a forest. M consists of a forest in each G_i.) A *maximum cardinality matching* has the greatest number of pairs possible. A matching is *perfect* if it is a base of M (in graphic matroids we have a spanning tree of each G_i). Suppose each element e has a real-valued *weight* $w(e)$. A *maximum perfect matching* is a perfect matching that has the largest possible total weight. Variants of this notion, like maximum weight matching (with unrestricted cardinality), maximum cardinality-k matching, etc., have the obvious meaning. The *weighted matroid intersection problem* is to find a maximum perfect matching, or to find any of the above variants. The independent assignment problem is a slight generalization of weighted matroid intersection, allowing many-to-many pairing functions. All of these problems are solved by the algorithm of [GX].

2. Matroid intersection algorithm.

This section briefly summarizes the weighted matroid intersection algorithm for general matroids. It points out the features that are difficult to implement on specific matroids. A complete development is in [GX].

The matroid algorithm generalizes the bipartite graph matching algorithm of [GT89]. It uses Frank's version of the linear programming dual problem for matroid intersection [Fra]. Each element e has an integral *dual value* $x(e)$. The algorithm consists of $O(\log(nN))$ *scales.* Each scale adds one bit of precision to the element weights and finds a 1-*optimal matching* for these weights. The last scale halts with the desired maximum perfect matching.

The algorithm for a scale is based on the idea of augmenting paths, generalizing augmenting paths for ordinary graph matching and network flow [L]. To understand this generalization first consider ordinary graph matching. Let e, f be two consecutive edges in an augmenting path for a matching M. The relation between e and f is that precisely one of e, f is in M, and e and f have a common vertex. This relation generalizes to matroids as follows. Recall the matching M is an independent set. It is convenient to enlarge M to a base $\overline{M}$ by adding artificial elements, called *singletons.* A pair of elements e, f is a *swap (for M)* if precisely one of e, f is in $\overline{M}$, $\overline{M} \oplus \{e, f\}$ is a base, and $x(e) = x(f)$. (The last condition is analogous to a similar condition for weighted graph matching.) Note that the common vertex condition in graph matching has been replaced by the condition, $\overline{M} \oplus \{e, f\}$ is a base. The latter is equivalent to $e \in C(f)$ (which is also equivalent to $f \in C(e)$) where fundamental circuits and cocircuits are with respect to the base $\overline{M}$. To *execute swap* e, f means to replace the current matching $\overline{M}$ by $\overline{M} \oplus \{e, f\}$. An *augmenting path* is a

sequence of the form

$$\ldots, e_i, \overline{e_i}, e_{i+1}, \overline{e_{i+1}}, \ldots$$

whose pairs $e_i, \overline{e_i}$ are alternately in M and not; consecutive elements of the form $\overline{e_i}, e_{i+1}$ are swaps; and executing all these swaps gives a matching with one more pair. (In addition the path begins and ends with a singleton; both singletons are in swaps of the path, like any other element; executing the swaps reduces the number of singletons by two.)

What makes matroid matching harder than ordinary graph matching is that the essentially trivial common vertex condition is replaced by a complicated condition ($e \in C(f)$). To illustrate the difficulties this causes, note that e, f and g, h may both be valid swaps for base $\overline{M}$, but executing e, f can make g, h invalid (for the new matching). The reader can easily illustrate this on a graphic matroid.

The algorithm for a scale works as follows. It initializes the matching to $\emptyset$. Then it repeatedly does a Search Step followed by an Augment Step until the desired perfect matching is found. The Search Step adjusts the duals values so there is an augmenting path composed of eligible pairs ("eligible" is a condition on the dual values, not defined here). The Augment Step repeatedly finds an augmenting path of eligible pairs and uses it to enlarge the matching. It stops when there are no more such paths.

We give some details of the two Steps. The Search Step is a generalization of the Hungarian search for bipartite graph matching [L] (equivalently it generalizes Dijkstra's shortest path algorithm [T83]). It makes a sequence of changes to the dual values. Each change is based on a positive *dual adjustment quantity* δ. Calculating the appropriate value of δ is the main difficulty in implementing the Search Step. Section 3 discusses this.

The Augment Step is done using a depth-first search. It is similar to the analogous depth-first search in network flow algorithms [e.g., T83, p. 104]. The important difference is in the operation of adding an edge to the search path: The new edge must be selected to ensure that the path gives a sequence of swaps that can be executed validly. Previous matroid intersection algorithms [e.g., Cu, E, Fra, GS, K, L, W] do this by requiring that the augmenting path has no "shortcuts". Depth-first search can give paths that have shortcuts. So we use a different strategy, as follows.

A *topological numbering* assigns a nonnegative integer $t(e)$ to each element e, such that

(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 $e_0, e_1 \notin M$.

Now suppose the depth-first search explores a path P, where for $i = 0$ or 1, e, f and g, h are swaps

in $P \cap E_i$ and e, f precedes g, h in P. The definition implies that

$$t(e) \geq t(f) > t(g) \geq t(h). \tag{1}$$

As shown in [GX] this implies that an augmenting path P is valid. More precisely, if the swaps of P are executed according to their order along P then each swap results in a valid matching.

The intersection algorithm changes t as it progresses. To preserve the validity of t, the depth-first search must choose edges according to this rule: To extend the depth-first search path P from an end e, add an element f such that e, f is a swap and $t(f)$ is as large as possible. The main difficulty in implementing the Augment Step is finding f. Section 4 discusses this.

The basic properties of a scale in the matroid algorithm are similar to the graph matching algorithm of [GT89]. At any time in the algorithm define Δ as the sum of all dual adjustment quantities δ in all Hungarian searches of the current scale; also define p as the number of pairs needed to add to the current matching to make it perfect.

Lemma 2.1. Any scale has the following properties.

(*i*) At any time, $p\Delta \leq 5n$.

(*ii*) $O(\sqrt{n})$ Augment and Search Steps are executed.

(*iii*) There are $O(n \log n)$ swaps total in all augmenting paths. ∎

3. Hungarian search.

This section describes the heavy path representation of a graph. It helps solve problems involving cycles and cocycles of a static spanning tree. We illustrate this by implementing the Hungarian search for graphic matroid intersection; this gives an efficient strongly polynomial algorithm.

The heavy path representation of a graph is defined for a graph G with spanning tree T. It is based on the decomposition of T into heavy paths due to Tarjan [T79] which we now review. Consider a tree with root r. For any nonleaf vertex v, the *heavy child* of v is the child with the greatest number of descendants (ties for "greatest" are broken arbitrarily). A tree edge joining parent to heavy child is a *heavy edge* and all other tree edges are *light.* A path P that goes from a leaf d to one of its ancestors a is an *h-path* if every edge preceding the last edge is heavy, and either the last edge is light or $a = r$. The h-paths partition the edges of T. The path from any vertex v to the root r is thus partitioned into at most $\log n$ subpaths – each subpath is included in some h-path H and ends with the last edge of H.

In the *heavy path representation of* G (illustrated in Fig. 1) each nontree edge e of G is represented by between one and three *back edges.* Each back edge is a directed edge from a vertex

to one of its ancestors; it is called either *initial* or *final.* The back edges of an edge $e = vw$ are defined as follows. Let a be the nearest common ancestor of v and w. For $x = v, w$, let P_x be the path from x to a. Let z be the vertex on P_x that is closest to a and is on a light edge of P_x ($z = x$ if no such light edge exists). If $z \neq x$ then the directed edge xz is the *initial* (*back*) *edge of* e *for* x. If $z \neq a$ then the directed edge za is the *final* (*back*) *edge of* e *for* x. If z equals x or a the corresponding back edge does not exist. Any nontree edge e has at most one final edge and at most two initial edges. Ignoring nontree edges, the fundamental cycle of e is the disjoint union of the fundamental cycles of its back edges.

The heavy path representation – the decomposition of T into heavy paths, plus the back edges for all nontree edges, can be found in time $O(m)$ [HT]. Simpler algorithms running in time say $O(m \log n)$ suffice for the applications of this section.

The heavy path representation is usually mapped into the following data structure. Consider a back edge e. If e is final it joins two vertices on the same h-path H; information about e is stored in a data structure for H, typically a segment tree or interval tree [Me]. If e is initial it is associated with a subtree of T as follows. A vertex s of T that has a light child is the root of an *initial tree* I. I is the subtree of T containing s and all descendants of light children of s. All initial edges e directed to s are associated with I; information about e is stored in a data structure for I, typically a search tree or dynamic tree. Often, as in the algorithms of this paper, the h-path data structure reduces to the initial tree data structure.

This data structure is efficient because efficient data structures usually exist for the h-paths and initial trees. Note that the number of back edges is $O(m)$, the size of all h-paths is $O(n)$, and the size of all initial trees is $O(n \log n)$. The latter holds because a vertex is in at most $\log n$ initial trees. The initial trees can usually be modified to have size $O(m)$, as illustrated below.

The data structure can be used to solve many problems involving cycles and cocycles for a static tree. It is used to implement the 2-data structure for the Augment Step in Section 4.3. Here we use it to implement the Hungarian search. Additional applications are given in [G].

The Hungarian search reduces to solutions to the (*weighted static-base*) *cycle* and *cocycle problems*, defined as follows. Given is a graph G with spanning tree T. Each edge e has a numeric value $x(e)$ (for the Hungarian search this is its dual value). Call an ordered pair e, f with $e \in C_T(f)$ a *swap*, having *value* $x(e) - x(f)$. Each edge is initially unmarked. The problem is to process on-line an arbitrary sequence of the following operations:

find_min – return a swap e, f with minimum value, subject to the constraint that e is marked

and f is unmarked;

mark(e) – mark the currently unmarked edge e.

The Hungarian search uses the value of the swap e, f returned by *find_min* as the next dual adjustment quantity δ; it changes duals values, making the value of swap e, f equal to zero, and then does *mark*(f). Note that a swap with value zero is a swap in the sense of Section 2. Using "swap" in a more general sense in this section will not cause confusion. Also, although the Hungarian search changes dual values, the changes can be ignored here, assuming the search is implemented using the appropriate offset quantities to dual values, as is usually done. Finally, in the *mark* operation the Hungarian search assigns labelling information to e. This also can be ignored in this section.

In the static cycle problem, the swap returned by *find_min* must have $e \notin T$, while in the static cocycle problem $e \in T$. Using the heavy path representation, each of these two problems splits into two – for initial edges and for final edges. We now show that for each of these four problems, $O(m)$ operations can be processed in total time $O((m + n \log n) \log n)$. Note that $O(m)$ operations allows each edge to be returned in a swap of *find_min* and also to be marked; this bounds the work done in a Hungarian search.

We start by discussing the problems for initial edges. The data structures are based on these observations. Call an edge *available* if it can be returned in a swap by *find_min*, i.e., in the cycle problem it is a marked nontree edge or an unmarked tree edge, and in the cocycle problem it is a marked tree edge or an unmarked nontree edge. Consider an initial tree with root s. Call a tree edge *undominated* if it is the unique smallest available edge in its path to s; call a nontree edge vs undominated if it is the largest nontree edge joining v and s. Clearly a swap returned by *find_min* need only involve undominated edges. Note that a given edge e of T can occur in $\log n$ initial trees. Some occurrences can be undominated and others not (more precisely e will be undominated in all initial trees up to a certain size).

Consider an initial tree I rooted at s. The data structures for the cycle and cocycle problems both use two minimizing (or maximizing) dynamic trees for I. It is convenient to name them using matroid terminology – B stores the basic elements and N the nonbasic elements. In graph terminology, B is a dynamic tree for I representing the tree edges; N is a dynamic tree for I representing the nontree edges. Now we describe the properties of these dynamic trees that are common to both the cycle and cocycle data structures. To simplify notation, for vertex v let v' denote the parent of v in the given tree T.

In B, a vertex v corresponds to the tree edge vv'. When vv' is in the data structure B,

$key(v) = x(vv')$. If it is not in the data structure its key is $\pm\infty$, where the sign is positive for a minimizing dynamic tree and negative otherwise. (We write ∞ for $+\infty$, which is usually the case.)

In N, each vertex v corresponds to the initial edge vs. It has $key(v)$ equal to the x value of the undominated edge vs, if it exists; otherwise the key is $-\infty$. In addition each undominated tree edge is cut in N.

The reason for the cutting is that if tree edge vv' is undominated, the subtree descending from v in N (assuming cuts have been made) contains precisely the vertices d whose smallest tree-edge ancestor is vv'. In other words a smallest swap for the nontree edge ds uses vv'.

The last part of the data structure is a priority queue pq serving all initial trees. For each undominated tree edge e, pq contains the swap formed by e and the largest edge in its subtree in N. The priority of the swap equals its value. Note that a given edge e of T can have up to $\log n$ entries in pq, but the total number of entries in pq is at most m.

Now we describe the remaining details of the data structures and the algorithms for the cycle and cocycle initial edge problems.

First consider the cocycle problem. Data structure B contains the undominated tree edges. Each vertex has two dynamic tree keys whose value is defined as above – one key supports finding the largest vertex in a subtree and the other key supports finding the smallest vertex in a path. The data structure for N has an additional component: Each vertex v maintains a list $\ell(v)$ of available, i.e., unmarked, initial edges vs, sorted according to nonincreasing x value. Thus $key(v)$ is the x value of the first edge in $\ell(v)$.

The algorithms for *find_min* and *mark* follow in a straightforward way from the description of the data structure. We now give the details and verify the time bound.

Consider *mark*(e) for a tree edge $e = vv'$. This edge is in at most $\log n$ initial trees I. Do the following for each such tree I in order of increasing size: In B, find the nearest ancestor a of v. If $x(aa') \leq x(e)$ then the processing of e is complete – e does not get added to B for this initial tree or for any larger one. Otherwise add e to B and cut e in N. Then make sure that in B all proper descendants d of v are undominated: In B find the descendant d of v with largest value $x(dd')$; if $x(dd') \geq x(e)$ then delete d from B, link dd' in N, delete the entry for dd' from pq, and repeat. Finally find the best swap for e: in N, find the largest descendant d in the subtree of v, and make a corresponding entry in pq; if d was previously the best swap for the above ancestor aa', execute a similar procedure for aa'.

To account for the time, observe that adding a tree edge e to the data structures for a given initial tree involves $O(1)$ dynamic tree and priority queue operations; the same holds for deleting

e. Since e is in $O(\log n)$ initial trees this gives total time $O(n \log^2 n)$.

Next consider *mark*(e) for a nontree edge $e = vs$ in the initial tree of s. In the associated tree N, delete edge e from list $\ell(v)$, and set $key(v)$ to the value of the edge now first in $\ell(v)$. If e was previously the best swap for tree edge ww', then delete this swap from pq; in N find the largest vertex in the subtree of w and add a corresponding swap to pq.

The time amounts to $O(1)$ dynamic tree and priority queue operations to mark e. This gives total time $O(m \log n)$.

The implementation of *find_min* is trivial, since the desired minimum value swap is the smallest entry in pq. The time for *find_min* is $O(1)$. We conclude that the desired time bound holds for the initial edge data structure for the cocycle problem.

Next consider the cycle problem for initial edges. The data structure for B contains the available tree edges. In addition each undominated tree edge is cut in B as well as in N.

Consider *mark*(e) for a nontree edge $e = vs$ in the initial tree of s. In the associated tree N, if $key(v) \geq x(e)$ then the processing of e is complete. Otherwise set $key(v)$ to $x(e)$. In N let v be in the subtree for tree edge ww'. If e gives a smaller swap for ww' than its current swap then make e, ww' the new entry in pq. The total time is $O(m \log n)$.

Finally consider *mark*(e) for a tree edge $e = vv'$. Do the following for each initial tree I where e is undominated. Let e' be the undominated edge whose subtree contains v'. In e's subtree of B (i.e., the subtree of B containing v) find the smallest descendant w, corresponding to available tree edge $ww' = g$. If $x(g) < x(e')$ then g is undominated, cut g in B and N, find the best swap for g in N, add it to pq, and repeat. In the opposite case $x(g) \geq x(e')$, link e in B and N, delete v from B (i.e., set its key to ∞) and recompute the best swap for e'; this completes the processing of e in B.

Note that the data structure for B is initialized by executing *mark*(e) for a fictitious edge e joining the root of I to a fictitous parent, with $x(e') = \infty$. The total time for *mark*(e) for all tree edges e is $O(n \log^2 n)$, as in the cocycle problem.

Now consider space. The cycle and cocyle algorithms both use $O(m + n \log n)$ space. The first term accounts for all swaps in pq, since a nontree edge is in at most two such swaps. The second term is due to the fact that the initial trees have total size $O(n \log n)$. To eliminate the second term we introduce an idea that is central in Section 4.

Consider a tree T and a subset of vertices $A \subseteq V(T)$. Define two associated trees, illustrated in Fig. 2(a) – (c). (In Fig. 2 tree edges are solid and nontree edges are dotted. In Fig. 2(a) ignore the circle and dotted edges, which refer to Section 4. In this figure the set A_0 contains the five

circled vertices.) The *connecting tree* $C(A,T)$ contains all vertices and edges of T that are on paths joining two vertices of A. Let $\overline{A} = A \cup \{v | v \text{ is a vertex with degree at least three in } C(A,T)\}$. The *contracted connecting tree* $\overline{C}(A,T)$ has vertices $\overline{A}$. Two vertices of $\overline{A}$ are adjacent if the path joining them in T contains no other vertex of $\overline{A}$. Less formally, the paths joining nodes of $\overline{A}$ in $C(A,T)$ become edges in $\overline{C}(A,T)$. It is easy to see that $|\overline{A}| \leq 2|A| - 2$. Section 4 uses some obvious extensions of this notation: We allow A to contain vertices not in T, and we allow T to be a forest. (The above definitions are unchanged in these cases.)

The space for the initial edge data structures is reduced by maintaining, for each initial tree I rooted at s, the trees B and N as dynamic trees for the tree $\mathcal{I} = \overline{C}(A,I)$, where $A = \{v |$ an initial edge vs exists$\} \cup \{s\}$. Clearly this reduces the total space to $O(m)$. Note that tree edges in the same edge of $\mathcal{I}$ contain exactly the same initial edges vs in their cocycle.

To use $\mathcal{I}$ we need an efficient algorithm to find the edge of $\mathcal{I}$ containing a given tree edge vv'. We use the following characterization. Assume T, the tree that is given for the Hungarian search, is rooted and numbered in preorder; for convenience identify each vertex with its preorder number. Let d be the smallest vertex of $\mathcal{I}$ that is at least v. Then vv' is in an edge of $\mathcal{I}$ if and only if it is in the edge from d to its parent in $\mathcal{I}$ if and only if v is an ancestor of d.

The characterization follows easily from the observation that an edge of $\mathcal{I}$ joins a vertex to one of its descendants. It is easy to find the desired edge of $\mathcal{I}$ (or determine that it does not exist) in time $O(\log n)$ by storing the preorder numbers of the vertices of $\mathcal{I}$ in sorted order, and doing a binary search to find d.

The trees $\mathcal{I}$ can be constructed in total time $O(m + n \log n)$ as follows. Process each initial tree I bottom-up. At each vertex v, with subtree I_v in I, construct the contracted connecting tree $\mathcal{I}_v = \overline{C}(A - s, I_v)$. This is easy to do: If v has at least two children w with nonempty contracted connecting trees then $\mathcal{I}_v$ contains vertex v with children the roots each such $\mathcal{I}_w$. The other cases are similar.

Using the trees $\mathcal{I}$ in the initial data structures does not increase the time. It only adds an operation to find the $\mathcal{I}$ edge containing e in operations $mark(e)$ for tree edges. This completes the initial edge data structures.

Now consider the final edge data structures. Using interval trees these data structures reduce to the initial edge data structures. The details are given below. It will be seen that this reduction applies to the heavy path representation in general.

Each h-path is represented by an interval tree I. Number the vertices of the h-path consecutively from 0 to $k-1$. Consider a node ν of I. It corresponds to an interval $[i..j) \subseteq [0..k)$; let p

denote its midpoint. ν stores all final edges ℓr that have $i \leq \ell \leq p \leq r < j$. Replace each such edge by edges ℓp and pr; clearly the fundamental cycle of the original edge is the union of the two new fundamental cycles, so we can work with the new edges. Call the path from vertex 0 to p a tree rooted at p, so all edges ℓp are incident to the root. Clearly we can store the edges ℓp in the previously-described data structure for an initial tree. Similarly for the edges pr.

The timing analysis is similar to the previous one: An edge e (tree or nontree) is processed in time $O(\log n)$ in each initial tree data structure; e is in at most $\log n$ nodes of the interval tree if it is a tree edge, and in one node if it is nontree. The assignment of nontree edges to interval tree nodes can be easily done in time $O(m \log n)$ (a time $O(m)$ procedure is described in Section 4.3). The desired time bound follows. Note that various simplifications of this data structure can be made, since the underlying tree of the initial edge data structure is a path.

Theorem 3.1. A Hungarian search on a graphic matroid can be implemented in time $O((m + n \log n) \log n)$ and space $O(m)$. The weighted graphic matroid intersection problem can be solved in time $O(n(m + n \log n) \log n)$ and space $O(m)$.

Proof. The first result follows from the above discussion. The second result follows since the matroid intersection algorithm of [Fra] consists of at most n Hungarian searches. ∎

4. Augment step.

This section presents the Augment Step and finishes the analysis of the graphic weighted intersection algorithm. Note that [Fre] solves cycle and cocycle problems similar to those discussed here, but our algorithms are more efficient because they take advantage of various properties of the matroid intersection algorithm. The section is organized as follows. This introductory portion defines the dynamic-base cycle and cocycle problems which arise in implementing the Augment Step. Section 4.1 solves the dynamic-base cycle problem. Section 4.2 gives a high-level solution to the dynamic-base cocycle problem and completes the timing analysis of the entire algorithm. Section 4.3 supplies the details of several data structures assumed in Section 4.2.

The general matroid algorithm specifies the Augment Step completely except for two operations c and *swap*. To motivate their definition, the Augment Step uses c to grow the depth-first search path P and *swap* to execute the swaps in an augmenting path P. More precisely when e is the last edge of P, an operation $c(e)$ is performed. It returns an edge f that forms a swap e, f. The Augment Step enlarges P to $P, f, \overline{f}$. If this completes an augmenting path, operation $swap(g, h)$ is done for each swap g, h in the path; this enlarges the matching. If the path is not augmenting,

the search continues with the operation $c(\overline{f})$. (Note that e and $\overline{f}$ are "opposites" with respect to matroid ($\mathbf{M}_0$ or $\mathbf{M}_1$) and matching M. This means that in one matroid c is always executed with its argument matched, and in the other matroid the argument is unmatched.) On the other hand if $c(e)$ returns $\emptyset$ then e is a dead-end of the search and the search backs up. Similarly the returned edge f may subsequently be declared a dead-end of the search, in which case $c(e)$ is executed again.

Now we define the operations by stating the *dynamic-base cycle* and *cocycle problems.* Given is a graph G where each edge e has a topological number $t(e)$. Also given is a spanning tree M (the matched edges); initially M is a tree M_0, and M gets changed by the operations. The operations concern elements e and f, where in the dynamic-base cycle (cocycle) problem $e \notin M \cup M_0$, $f \in M \cap M_0$ ($e \in M \cap M_0$, $f \notin M \cup M_0$). There is a set of edges F that contains the possible results for c; in the dynamic-base cycle (cocycle) problem F is initialized to M_0 ($G - M_0$). The problem is to process, on-line, a sequence of operations of two types:

swap(e, f) – execute the swap e, f, i.e., $M \leftarrow M \oplus \{e, f\}$;
$c(e)$ – find an edge $f \in C_M(e) \cap F$ with maximum value $t(f)$, delete f from F and return f.

The *swap* operation must be called with $e \in C_M(f)$ (equivalently $f \in C_M(e)$) so it results in a new spanning tree M. Note that in $c(e)$, a given edge f is returned at most once in the entire sequence of operations. If no such f exists, $c(e)$ returns $\emptyset$.

The depth-first search of the Augment Step is implemented by executing an algorithm for the dynamic-base cycle problem on matroid $\mathbf{M}_0$ and an algorithm for the dynamic-base cocycle problem on $\mathbf{M}_1$. Thus we need only solve these two problems. Our solution depends on the above description of the Augment Step. It also uses property (1) (see Section 2) of consecutive swaps in P.

The careful reader may be puzzled because although the definition of a swap e, f (in Section 2) requires $x(e) = x(f)$, and the Augment Step performs swaps of the form $e, c(e)$, yet the definition of $c(e)$ ignores dual values. Duals can be ignored because of a simplification of the Augment Step: For each distinct dual value x the Augment Step maintains a graph G_x, which has all edges of M contracted except those with dual value x; thus the matched edges in G_x form a tree M_x. The operations $c(e)$ and *swap*(e, f) are both executed in $G_{x(e)}$. (A given depth-first search path or augmenting path may contain a number of swaps in the same graph G_x. If the swaps of an augmenting path are executed according to their order on P then each swap is valid, i.e., it gives a new spanning tree.) At the end of the Augment Step the new matching is just the union of the

matched edges from each contracted graph G_x. This simplification of the Augment Step is valid in any matroid [GX]. *

The graphs G_x are constructed at the start of the Augment Step in time $O(m)$ as follows. We do a depth-first traversal of the tree M and simultaneous depth-first traversals of each tree M_x. Actually M_x and G_x are unknown but get created during this traversal. Initially each G_x has one vertex; it corresponds to the root of M, and is the start vertex of the depth-first search in G_x. When an edge $e = vw$ of M is traversed in the forward direction, reaching vertex w for the first time, a new vertex $\overline{w}$ is created in $G_{x(e)}$; if $\overline{v}$ is the current vertex of the depth-first search of $G_{x(e)}$, edge $\overline{v}\,\overline{w}$ corresponding to e is added to $G_{x(e)}$ and $\overline{w}$ is made the current vertex in $G_{x(e)}$; for each edge f incident to w (in G), an image $\overline{f}$ is added to the adjacency list of the current vertex in the depth-first search of $G_{x(f)}$. (Thus each edge f of G is eventually added to two adjacency lists in $G_{x(f)}$.) When edge $e = vw$ of M is traversed in the reverse direction, the current vertex of the depth-first search in $G_{x(e)}$ is changed from $\overline{w}$ to the previous vertex.

The time bound for the Augment Step depends on the fact that the graphs G_x have a total of m edges and $O(n)$ vertices. This follows since each edge of G appears in a unique G_x, and each edge of M corresponds to a spanning tree edge in a unique G_x.

In the following solutions to the dynamic-base problems it is convenient to use phrases like "the largest edge"; since the only value associated with an edge is its topological number, the meaning is clear.

4.1. Cycle problem.

This section presents the algorithm for the dynamic-base cycle problem. It is simpler and more efficient than the cocycle problem. Some matroid matching problems do not need the cocycle problem, in which case the running time of our algorithm improves slightly.

The dynamic-base cycle problem can be solved in time $O(m + n \log n)$ per Augment Step, using the following data structure. The edges of M are stored in a dynamic tree; in this tree an edge e has key $t(e)$ if $e \in F$, else 0. (As in [ST] the maximum key in a tree path can be found in time $O(\log n)$.) A second data structure does set merging on a universe containing all vertices of the graph. At any time the universe is partitioned into sets called *v_sets*. Each *v_set* is connected in the graph $M - F$; in fact immediately after each augment the *v_sets* are precisely the connected components of $M - F$. (Since $M - F$ is a forest, a *v_set* is always a subtree.) Initially each vertex

* Our definitions of the dynamic-base problems differ from those of [GX] – here we ignore duals and use the candidate sets F. It is easy to prove our definitions are equivalent to [GX].

is a singleton v_set.

The operation $swap(e, f)$, with $e = vw$, cuts edge f in the dynamic tree and links vertices v and w, assigning the new edge vw a key of zero; lastly it merges the v_sets for v and w. The operation $c(e)$, $e = vw$, returns $\emptyset$ if v and w are in the same v_set; otherwise it finds the largest edge f in the dynamic tree path from v to w, changes the key of f to zero, deletes f from F and returns f. If $f = xy$ is subsequently found to be a dead-end, it merges the v_sets containing x and y.

The correctness of this algorithm follows easily from the fact that the c operation does not return edges that were previously returned or swapped into the matching. The efficiency depends on the fact that if $c(e)$ does a dynamic tree operation then $C(e)$ contains an edge $f \in F$. To show this observe that a tree edge $\ell \notin F$ with its ends in different v_sets is in the current depth-first search path P. In fact it precedes e, so $t(\ell) > t(e)$ by (1). This implies $\ell \notin C(e)$. We conclude that an edge in $C(e) - F$ has its ends in the same v_set. Thus if $c(e)$ does a dynamic tree operation, $C(e) \cap F \neq \emptyset$, as desired.

To estimate the time note that there are $O(n)$ dynamic tree operations: A *swap* operation does $O(1)$ dynamic tree operations and removes an edge of M_0 from M; a c operation can do $O(1)$ dynamic tree operations, but only if it returns an edge of M_0. Thus the dynamic tree operations use time $O(n \log n)$. The set merging data structure does $O(m)$ *finds* and $O(n)$ *unions* and hence uses time $O(m\alpha(m, n))$ [T83]. This gives total time $O(m + n \log n)$ as desired. (Note that this time bound holds for each graph G_x, and hence for the entire graph G.)

4.2. Cocycle problem: high-level.

This section gives a high-level description of the algorithm for the dynamic-base cocycle problem. The remaining details are given in Section 4.3. The cocycle algorithm is more involved than the cycle algorithm, and its bound dominates the running time.

We use several different graphs. When the graph of interest H may not be clear, we write *H-vertex* or *H-edge*.

It is convenient to assume that the given graph G has maximum degree at most three. This is achieved by a well-known transformation: Replace a vertex of degree $d > 3$ by a path P of vertices v_i, $i = 1, \ldots, d$. Change each edge incident to v into an edge incident to a distinct v_i. Add all edges of P to M, the given spanning tree of G. The transformed graph (which we still call G) has maximum degree three and spanning tree M. The artificial edges in paths P will never be used as arguments to c, and hence are never arguments to *swap*. So the results for c and *swap* in the

transformed graph are valid for the given graph. The transformed graph has $O(m)$ vertices and edges; the operation $c(e)$ is done for less than n distinct edges e.

Our approach is to maintain a representation of G based on the recently swapped edges, called the wakened graph $\mathcal{W}$. It is defined precisely below. For now we give some relevant terminology. $\mathcal{W}$ is constructed starting at some arbitrary time, called *$\mathcal{W}$-initialization time* or *$\mathcal{W}$-time* for short. $\mathcal{W}$ is used to find and process a number of augmenting paths. The parameter

$$\sigma$$

specifies the maximum number of *swap* operations allowed in one wakened graph. Its value is chosen below. When σ swaps have been processed the current wakened graph is discarded, a new $\mathcal{W}$-time is declared and the next wakened graph is initialized.

At any time, the edges that were matched at $\mathcal{W}$-time are partitioned into sets

$$S \cup W \cup D.$$

Any matched edge at $\mathcal{W}$-time starts out in S, *asleep*. At some point it may be moved to W, becoming *wakened*. A wakened edge will eventually (before the next $\mathcal{W}$-time) move to D, becoming *dead* (it gets *killed*). At the next $\mathcal{W}$-time, all (currently) matched edges are again asleep.

Here is a brief sketch of how these notions correspond to the depth-first search for an augmenting path. At $\mathcal{W}$-time the algorithm wakens at most σ edges (this is done for reasons of efficiency). Thereafter it wakens a subset of the edges that get added to the depth-first search path P. A wakened edge gets added to the wakened graph $\mathcal{W}$. It dies when it is removed from P – either when it is found to be a dead-end in the depth-first search, or when it gets swapped out of the matching in an augment. Note that some matched edges in P do not get wakened. They are not processed in $\mathcal{W}$. Such an edge e eventually becomes either a dead-end or part of the augmenting path. Note that in the former case e does not become "dead" – its classification remains "asleep". In the latter case e is wakened when its swap in the augmenting path is executed; the swap immediately kills e.

Observe this consequence of the sketch: At any time every edge in $S \cup W$ is matched.

Also, let us characterize the notion of a dead-end. Edge e is declared a dead-end of the depth-first search when $c(e)$ returns $\emptyset$, i.e., when $C(e) \cap F = \emptyset$. This condition is equivalent to e being a bridge of the graph $M \cup F$. As the search progresses the set $M \cup F$ only shrinks (edges may leave M, leave F, or go from F to M). Thus e remains a bridge of $M \cup F$. Equivalently, we conclude that the condition $C(e) \cap F = \emptyset$ always holds for a dead-end e.

The algorithm makes certain vertices *active*. Initially all vertices are inactive; once active, a vertex remains so until the next $\mathcal{W}$-time, when it becomes inactive again.

In terms of the depth-first search there are two ways for a vertex to become active. If an edge

that is unmatched at $\mathcal{W}$-time becomes matched in an augment, its ends are activated. Also the ends of any edge wakened by the algorithm at $\mathcal{W}$-time are activated.

We can now give the high-level description of *swap*(e, f). If this operation makes makes the total number of *swaps* more than σ, declare a new $\mathcal{W}$-time and initialize a new wakened graph. (The remaining swaps of the current augmenting path are processed trivially in the reinitialization.) Now consider the opposite case. The matched edge e has been wakened if swap e, f was discovered in the $\mathcal{W}$ graph, but otherwise e is currently asleep. Waken e if it is currently asleep. Then (in any case) remove e from the matching M, kill it, activate the ends of f and add f to M.

We turn to the c operation. It uses another classification of edges, illustrated in Fig. 2(a). Let U be a set of vertices. An edge of G is an *i-edge (with respect to U)* if it contains exactly i vertices of U; hence $i = 0, 1$ or 2. In this definition we will always choose U to be the vertex set of a tree T, and for convenience we refer to "an i-edge for T".

The analysis involves several connecting trees. The main one is the contracted connecting tree

$$\overline{\mathcal{T}} = \overline{\mathcal{C}}(A, S \cup W),$$

where the set of vertices A is defined inductively as follows. At $\mathcal{W}$-time (before any edge is wakened) $A = \emptyset$. Then whenever a vertex v is activated it is added to A. This may create other new vertices in $\overline{\mathcal{T}}$, which are also added to A. (Note that adding these new vertices to A does not change $\overline{\mathcal{T}}$.) *

The first part of the analysis uses the related connecting tree

$$\mathcal{T} = \mathcal{C}(A, S \cup W).$$

Strictly speaking $\mathcal{T}$ is a forest. The forest changes in two ways. First, an edge $e \in W$ may die. This removes e and possibly other edges from $\mathcal{T}$. Second, a swap that matches edge vw activates v and w. This adds edges to $\mathcal{T}$ if v or w was not previously in $\mathcal{T}$.

Now we give a high-level statement of the algorithm for $c(e)$. The algorithm is divided into the Connecting Case and the Unconnecting Case. Recall the task of $c(e)$ is to find the largest edge in $C(e) \cap F$. Four data structures are used to find cocycle edges. Each finds its desired edge in time $O(\log n)$. However the data structures have different set-up times.

Unconnecting Case: $e \notin \mathcal{T}$. Let T be the component of $S \cup W$ containing e. Define i-edges to be with respect to T. Recall that T is a tree of matched edges. Furthermore a matched 1-edge e is either a dead-end or has its ends active (the latter occurs when e was swapped into the matching

* A more parsimonious definition would be to take A as the set of active vertices. Most of the algorithm works unmodified for this definition. However the 2-data structure becomes more difficult to implement, because Lemma 4.2(*iii*) – (*iv*) fail.

in an augment).

We first show that $c(e)$ does not return any 0-edges. $T - e$ consists of two subtrees. The hypothesis $e \notin \mathcal{T}$ implies that at most one subtree contains active vertices. So consider a subtree N with no active vertices. Except for e, any matched edge incident to N is a dead-end, i.e., it is not in any cycle $C(f)$, $f \in F$ (recall the above characterization of dead-ends). Hence if a cycle $C(f)$ contains e then f is incident to N.

Thus $c(e)$ does not return 0-edges, only 1- or 2-edges. We use a separate data structure for each. More precisely, in the Unconnecting Case $c(e)$ works as follows: It uses the *1-data structure* to find the largest 1-edge in $C(e) \cap F$; similarly it uses the *2-data structure* to find the largest 2-edge; the larger of these two edges is deleted from F and returned as $c(e)$. Now we give the ideas underlying the two data structures. Detailed implementations are in Section 4.3. Both data structures are initialized at $\mathcal{W}$-time. Both use $O(m)$ time per wakened graph plus $O(m \log n)$ time per Augment Step. In addition the 2-data structure uses $O(n \log^2 n)$ time per Augment Step.

Consider 1-edges. If neither subtree of $T - e$ contains an active vertex then both subtrees qualify as the above N. In this case $C(e) \cap F$ contains only 2-edges, so the 1-data structure is irrelevant. In the other case one subtree contains active vertices, so N is unique. We have shown an edge $f \in C(e) \cap F$ is incident to N. It is easy to see that conversely, any 1-edge incident to N is in $C(e)$. Thus the 1-edges of $C(e)$ are precisely the 1-edges incident to a vertex of N. This characterization is the basis of the 1-data structure, which finds the largest 1-edge in $C(e) \cap F$.

Next consider 2-edges. A 2-edge is currently in $C(e)$ if and only if it was in $C(e)$ at $\mathcal{W}$-time (since any edge of T was matched at $\mathcal{W}$-time). This observation is the basis of the 2-data structure, which finds the largest 2-edge f in $C(e) \cap F$. In slightly more detail, the 2-data structure finds f as the largest edge in $C(e)$ at $\mathcal{W}$-time, that is currently in F. Then it checks that the ends of f are in the same component of $S \cup W$ as e. If so f is returned. If not, observe that f is not a 2-edge for any other edge e' (since f now joins distinct components of $S \cup W$). Thus f can be deleted from the 2-data structure in this case.

Connecting Case: $e \in \mathcal{T}$. We discuss the Connecting Case by considering a fixed edge ε in $\overline{\mathcal{T}}$. Let ε join vertices ℓ, $r \in A$. Since ε corresponds to a path from ℓ to r in $S \cup W$, it makes sense to refer to a (G-)edge in ε. Also note that ε is created when ℓ or r enters A, i.e., when some swap is executed or when an edge is wakened at $\mathcal{W}$-time. ε is destroyed when one of its edges dies, or it is destroyed by a swap. In the latter case the swap activates a vertex v, which in turn adds a G-vertex w in ε to A, thus replacing ε by edges ℓw and wr in $\overline{\mathcal{T}}$.

We will discuss all operations $c(e)$ that are executed with e in ε, while ε exists as an edge of $\overline{T}$. Obviously any Connecting Case edge e is in some such ε.

Let T be the subtree of $S \cup W - \{\ell, r\}$ that contains all G-vertices in $\varepsilon - \{\ell, r\}$. ($T$ does not contain all edges of ε, and in fact $T = \emptyset$ if ε consists of a single edge ℓr.) Define i-edges to be with respect to T. A 2-edge for T was a 2-edge for T at $\mathcal{W}$-time. Hence 2-edges for $c(e)$ can be found using the above 2-data structure.

In contrast, the 1-data structure cannot be used for 1-edges in the Connecting Case. Instead we use the $\mathcal{W}1$-data structure. It is based on the following characterization. Number the G-vertices in ε from 0 to k, assigning 0 to ℓ and k to r. Deleting the edges of ε creates $k+1$ trees, each containing a unique vertex i in $[0..k]$. Call a G-vertex *type* i if it is in the same tree as i. Then for a tree edge $e = h, h+1$ a 1-edge in $C(e)$ joins a type i vertex to type j, where either $i = 0$ and $h < j < k$ or $i = k$ and $0 < j \leq h$.

The $\mathcal{W}1$-data structure has a larger set-up time than those of the Unconnecting Case. Each time the $\mathcal{W}1$-data structure is used for a new edge ε of $\overline{T}$ there is an additional set-up time of about $\sqrt{m}$. This enables it to process all operations $c(e)$ for all edges $e \in \varepsilon$.

The last data structure is the $\mathcal{W}$-data structure. It has the largest set-up time. It finds the desired edge f for $c(e)$ (regardless of whether f is a 0-, 1- or 2-edge). Each time the $\mathcal{W}$-data structure is used for a new edge e there is an additional set-up time of about $\sqrt{m}$. This set-up corresponds to wakening e.

Further details of these data structures are given below. We now describe how $c(e)$ works for all edges $e \in \varepsilon$. We show that all such executions $c(e)$ are processed by wakening at most one edge of ε.

First observe a property of 0-edges. All edges $e \in \varepsilon$ contain exactly the same 0-edges f in $C(e)$. (This follows since the only matched edges incident to T that are not dead-ends are incident to ℓ or r. The property holds even for a 0-edge ℓr that joins the ends of ε.) Let t_0 be the smallest topological number of an edge in ε. Then any such f has $t(f) \leq t_0$.

Now suppose an edge $e \in \varepsilon$ gets added to the depth-first search path P. To execute $c(e)$, use the 2-data structure to find the largest 2-edge in $C(e) \cap F$. Similarly use the $\mathcal{W}1$-data structure to find the largest 1-edge. (If this is the first c operation for any edge of ε this involves initializing $\mathcal{W}1$ for ε.) Let f be the larger of the two edges found. If $t(f) \geq t_0$ then delete f from F and return it as the value $c(e)$. Process every edge of ε that gets added to P in this manner, until $t(f) < t_0$. (This may complete the processing of ε, if an augmenting path P is found. In this case the above description of *swap* shows that the augment wakens and kills every edge of ε in P. This destroys

ε.)

In general let $c(g)$, $g \in \varepsilon$, be the operation that first encounters the above condition $t(f) < t_0$. (This includes the possibility that no edge f is found in the 2- or $\mathcal{W}1$-data structures.) Waken edge g. Then use the $\mathcal{W}$-data structure to process this operation $c(g)$ and all subsequent operations $c(g)$.

We now show that this actually completes the processing for ε. First observe that as long as $g \in P$, no other edge of ε gets added to P: Let P contain swap g, f. (At different times there may be different edges f.) Let e' be an edge after f in P, in the same matroid ($\mathbf{M}_0$ or $\mathbf{M}_1$) as g, f. Then $t(e') < t(f) \leq t_0$ by (1). Thus $e' \notin \varepsilon$.

Eventually g is removed from P, when it dies (i.e., it is a dead-end or in an augmenting path). Then $g \notin S \cup W$ and ε is no longer an edge of $\overline{T}$. Thus the processing of ε is complete.

Note that if g becomes a dead-end, P may still contain edges e that were in ε. Subsequent calls $c(e)$, if any, are processed using the Unconnecting Case, since now $e \notin \mathcal{T}$.

This completes the high-level description of $c(e)$. We can now prove an important property for the efficiency of the algorithm.

Lemma 4.1. In one wakened graph, $O(\sigma)$ vertices are activated and $O(\sigma)$ edges are wakened.

Proof. Swaps create at most 2σ active vertices and σ wakened edges. The same holds for edges wakened at $\mathcal{W}$-time. Thus it remains only to show that c operations waken $O(\sigma)$ edges.

We have seen that c wakens at most one edge in each edge of $\overline{T}$. Furthermore it eventually destroys the edge of $\overline{T}$. So we need only bound the number of edges ever created in $\overline{T}$. A dead-end does not create a new edge (this relies on the definition of A). Hence only new active vertices create new edges of $\overline{T}$. A new active vertex creates at most two new edges in $\overline{T}$. We have seen there are $O(\sigma)$ active vertices, so the desired bound follows. ∎

Define an *asleep tree* to be a maximal tree of asleep edges. (It is possible for an asleep tree to consist of a single vertex.) The lemma implies there are $O(\sigma)$ asleep trees. We shall see that any asleep tree has $O(m/\sigma)$ vertices. (In fact this is the purpose of the initialization at $\mathcal{W}$-time).

Now we define the *wakened graph* $\mathcal{W}$. It consists of *$\mathcal{W}$-nodes* and *$\mathcal{W}$-edges*. A $\mathcal{W}$-node is the contraction of an asleep tree. (Hence there are $O(\sigma)$ $\mathcal{W}$-nodes.) The $\mathcal{W}$-edges are the G-edges that join distinct $\mathcal{W}$-nodes. More precisely, $\mathcal{W}$ contains a spanning tree of matched G-edges – edges that are wakened, dead-ends or were swapped into the matching. For $\mathcal{W}$-nodes ν, ν', an unmatched $\mathcal{W}$-edge $\nu\nu'$ corresponds to the G-edges joining ν and ν' that are in F (equivalently, they have not been reached in the depth-first search; $\nu\nu'$ exists only if such an edge exists).

The $\mathcal{W}$- and $\mathcal{W}1$-data structures are representations of $\mathcal{W}$. These data structures use time

$$O(m + \sigma^2 \log n)$$

per wakened graph, plus time

$$O(m \log n)$$

per Augment Step. The space is $O(m + \sigma^2)$. We describe the $\mathcal{W}$-data structure; details of the $\mathcal{W}1$-data structure are in the next section. The $\mathcal{W}$-data structure is based on the connecting tree

$$\mathcal{D} = \overline{C}(A, M).$$

Recall M denotes the current matching. (Thus M contains all edges that are asleep, wakened, dead-ends and edges swapped into the matching.)

The implementation maintains asleep trees, and the related connecting tree

$$\mathcal{S} = \overline{C}(A, S).$$

This provides the information needed from the other connecting trees such as $\mathcal{T}$ and $\overline{\mathcal{T}}$. For future reference we state here the properties of asleep trees that are needed in the implementation.

Part (iv) below allows the 1-data structure to be implemented using asleep trees. Recall that in the Unconnecting Case, T denotes the component of $S \cup W$ containing e. When the 1-data structure is used for $c(e)$, precisely one subtree of $T - e$, called N, has no active vertices. The implementation of the 1-data structure using asleep trees executes $c(e)$ as follows: Let τ denote the asleep tree containing e. Let ν be the subtree of $\tau - e$ with no vertices of A. Return the largest 1-edge with respect to τ that is incident to ν and in F.

Part (iii) below refers to the Connecting Case. Recall there ε denotes an edge of $\overline{\mathcal{T}}$, and T denotes a subtree of $S \cup W$. Let $T - \varepsilon$ denote the edges in T that are not in edge ε.

Lemma 4.2.

(i) The edges of $\mathcal{S}$ are precisely the edges of $\overline{\mathcal{T}}$(equivalently, $\mathcal{D}$) that contain only asleep edges.

(ii) Any asleep tree contains a vertex of A.

(iii) In the Unconnecting Case tree N does not contain a wakened edge or a vertex of A. Similarly in the Connecting Case for forest $T - \varepsilon$.

(iv) The implementation of the 1-data structure using asleep trees is correct.

Proof. (i) follows from the definition of asleep tree.

(ii) It suffices to show that at the instant an edge e is wakened, e is in an edge of $\mathcal{S}$. (Note that vertices never leave A.)

If e is wakened in the initialization at $\mathcal{W}$-time its ends are first activated. If e is wakened in the Connecting Case then it is in an edge of $\mathcal{S}$ by (i). Thus it only remains to consider operations

swap(e, f) that waken e. Wakening e creates two new asleep trees, say T, U. Clearly we can assume that U contains a vertex of A, and we need only show that T contains a vertex of A.

If f has an end v in T then *swap*(e, f) activates v, as desired. If there is no such end then $C(f)$ contains a matched edge g incident to T, $g \neq e$. Edge g is either wakened, a dead-end or has an active end in T. The last alternative gives the desired conclusion. We show the first two alternatives cannot hold. Clearly g is not a dead-end (a dead-end is not in cycle $C(f)$). If g is wakened, but its ends are not active, it was wakened when it got added to the depth-first search path. Hence g is in the augmenting path P. But this is impossible: g does not precede e in P, since it would have already been swapped out of the matching. And since the topological numbering is valid [GX], $t(g) \geq t(f)$, so (1) implies that g does not follow e in P.

(iii) First consider the Unconnecting Case. Let $T - e$ consist of subtrees N and X. So X contains an active vertex. A wakened edge g is in $\mathcal{T}$ (whether g was wakened at $\mathcal{W}$-time or during the current search in the Connecting Case). Similarly a vertex $a \in A$ is in $\mathcal{T}$. Thus g or a is joined to the active vertex of X by a path in $\mathcal{T}$. That path does not contain e, since $e \notin \mathcal{T}$. Thus g or a is in X as desired.

The argument for the Connecting Case is essentially the same, using the observation that edges in T that are not in ε are not in $\mathcal{T}$.

(iv) First observe that τ is well-defined – e is asleep, since it is processed in the Unconnecting Case.

We start by showing $\nu = N$: Part (iii) shows that N does not contain a wakened edge, so N is a subtree of $\tau - e$. Furthermore (iii) shows that N does not contain a vertex of A, so $N = \nu$. Note that by (ii), ν is uniquely defined.

It remains to show that we can use 1-edges with respect to τ rather than 1-edges with respect to T. We have already observed that any 1-edge with respect to N is in $C(e)$. So the implementation for asleep trees surely returns a valid cocycle edge f. The 1-edges for τ include all 1-edges for T, plus some 2-edges for T. If f is a 1-edge for T it is the desired edge. If f is a 2-edge it is larger than the desired 1-edge and no harm is done – f is simply discovered in two data structures rather than one. ∎

We proceed to the data structure for the $\mathcal{W}$ graph. Each asleep tree has a distinct *name* in $[1..n]$; this is the index of one of its vertices in A (which exists by Lemma 4.2(ii)). Each vertex is labelled by the name of the asleep tree containing it. The vertices of A are marked. Edges are labelled asleep, wakened or dead as appropriate. The vertices and edges of $\mathcal{S}$ are marked.

Each unmatched $\mathcal{W}$-edge e stores its corresponding G-edges in a priority queue $pq(e)$. The key of a G-edge is its topological number.

In addition each $\mathcal{W}$-node ν maintains its own copy of the tree $\mathcal{D}$ as a maximizing dynamic tree $dt(\nu)$. This copy represents the $\mathcal{W}$-edges incident to ν as follows: Each $\mathcal{W}$-node μ corresponds to a subtree of $dt(\nu)$. (This follows from Lemma 4.2(i) - (ii).) Let u be the name of asleep tree μ. The key of u in $dt(\nu)$, $key(u)$, is the largest topological number of an unmatched G-edge in the $\mathcal{W}$-edge $\nu\mu$. This number is available from the $pq(\nu\mu)$. A vertex of $dt(\nu)$ that is not an asleep tree name has key $-\infty$. Finally, the copies of a given edge ε of $\mathcal{D}$ (in all trees $dt(\nu)$) are linked together.

To estimate the space for these dynamic trees, Lemma 4.1 implies that $|A| = O(\sigma)$. Hence $\mathcal{D}$ has $O(\sigma)$ vertices (from the discussion of connecting trees in Section 3). There are $O(\sigma)$ dynamic trees, implying the desired space bound $O(\sigma^2)$.

Now we describe the algorithms for $\mathcal{W}$. For each algorithm we verify its correctness (if necessary) and also show that the time is within the above bound for $\mathcal{W}$. We start with two primitive operations, activating a vertex and wakening an edge.

Activating a vertex v adds a path P to $\mathcal{C}(A, S)$, from v to a vertex w currently in the tree (by Lemma 4.2(ii)); let w be in an edge from ℓ to r in $\mathcal{S}$. (Possibly $v = w$ or $w = \ell = r$. The processing is simpler in these degenerate cases.) Scan the asleep tree containing v to find w; mark v and w as being in A and mark the edges of P as being in $\mathcal{S}$. In each dynamic tree $dt(\nu)$ cut edge ℓr, create new vertices v, w, and do links to create edges ℓw, wr and vw. (Some of these dynamic tree operations are omitted in the degenerate cases.)

To estimate the time to activate a vertex, scanning the asleep tree uses time $O(m/\sigma)$, since an asleep tree has $O(m/\sigma)$ vertices. There are $O(1)$ dynamic tree operations for each $\mathcal{W}$-node ν. There are $O(\sigma)$ $\mathcal{W}$-nodes and $O(\sigma)$ activations (Lemma 4.1). This gives total time $O(m + \sigma^2 \log n)$ per wakened graph.

Now we describe how to waken an edge e. Let e be in an asleep tree T that corresponds to a $\mathcal{W}$-node. Wakening e splits this $\mathcal{W}$-node into two, creating new $\mathcal{W}$-edges. This is accomplished as follows.

Let $T - e$ consist of new asleep trees T_i, $i = 1, 2$. Scan the vertices of each T_i to find a vertex $t_i \in A$. (Lemma 4.2(ii) shows t_i exists. When e is wakened at $\mathcal{W}$-time, its ends have already been activated; similarly when e is wakened in an operation $swap(e, f)$, the ends of f are already activated.) Mark e wakened. Make t_i the name of T_i and assign it to each vertex of T_i. If e is in an edge ε of $\mathcal{S}$ then remove the edges of ε from $\mathcal{S}$. (This is done since wakening e destroys ε. This occurs when e is wakened in the Connecting Case, or when $swap(e, f)$ is executed and all other

edges of ε are asleep.)

Next update the $\mathcal{W}$-edges. This means splitting each $\mathcal{W}$-edge $T\nu$ into edges $T_i\nu$, $i = 1,2$, and also creating edge T_1T_2 (some of these edges may not exist). For each new $\mathcal{W}$-edge create its associated priority queue. (Discard the data structures for T.)

The time for this is proportional to the number of G-vertices and edges involved, $O(m/\sigma)$. Lemma 4.1 implies the total time for one wakened graph is $O(m)$.

Next update the dynamic trees. Create dynamic trees $dt(T_i)$, $i = 1,2$ with keys set according to the $\mathcal{W}$-edges incident to T_i. Then update the dynamic tree $dt(\nu)$ for each $\mathcal{W}$-node $\nu \neq T_1, T_2$: If T had name t, set $key(t)$ to $-\infty$ in $dt(\nu)$, and set $key(t_i)$ to the maximum topological number in $\mathcal{W}$-edge $T_i\nu$. (Note the wakening does not change the topology of $\mathcal{D}$.)

This procedure does $O(1)$ dynamic tree operations per $\mathcal{W}$-node. This gives $O(\sigma \log n)$ time for each wakened edge. The number of such edges is $O(\sigma)$. Hence the time for the entire wakened graph is $O(\sigma^2 \log n)$.

Next we describe the main algorithms on the $\mathcal{W}$-data structure: initialization at $\mathcal{W}$-time, *swap*, and *c*. For the initialization, first recall a well-known construction [Fre, GT85]. Given is a tree T on n vertices, with maximum degree at most three; also given are parameters d and $s = n/d$. We wish to delete at most d edges from T so that each remaining subtree has $O(s)$ vertices. The *subtree construction algorithm* accomplishes this as follows: Root T at a degree one vertex and traverse T in postorder. To visit vertex v, label v with the size $s(v)$ of its (remaining) subtree, $1 + \sum s(c)$, where c ranges over all children of v with edge vc not deleted. If $s(v) \geq s$, delete the edge from v to its parent (if it exists).

The time for this construction is $O(n)$. Any subtree has size $s(v) < 2s$. At most d edges are deleted, since each deletion creates a subtree of size $s(v) \geq s$. Thus the algorithm works as desired.

We proceed to the initialization at $\mathcal{W}$-time. The object is to have each asleep tree incident to $O(m/\sigma)$ G-edges. This amounts to having $O(m/\sigma)$ G-vertices, by our assumption of bounded degree. To achieve this use the subtree construction algorithm. It deletes at most σ edges e from the tree M. For each such e, instead of deleting it waken e and activate its endpoints.

It is easy to see the time for initialization is $O(m)$.

Next we describe the operation $swap(e, f)$ in the $\mathcal{W}$-data structure. It works according to the high-level description already presented. Thus if a new $\mathcal{W}$-time is not declared proceed as follows: Let $f = uv$. Activate u and v. If e is asleep waken it. Note that e is in an edge ε of $\mathcal{S}$, which is also an edge of $\mathcal{D}$ (see the opening sentence of the proof of Lemma 4.2(ii)). Mark e dead. In each dynamic tree $dt(\nu)$ cut the ends of ε and link u and v together. (The activation makes u and v

vertices of $dt(\nu)$. The swap e, f gives a new spanning tree as long as the swaps of the augmenting path are executed in order.)

This procedure uses time $O(\sigma \log n)$ per swap, giving $O(\sigma^2 \log n)$ total.

Finally we describe $c(e)$ in the $\mathcal{W}$-data structure. We use an additional data structure, a priority queue $\mathcal{W}\_pq$. If e is the last edge that executed $c(e)$ in the $\mathcal{W}$-data structure then $\mathcal{W}\_pq$ contains its candidate cocycle edges. More precisely for each $\mathcal{W}$-node ν, $\mathcal{W}\_pq$ contains the largest edge f_ν of $C(e) \cap F$ that is incident to ν, if it exists.

To motivate the definition of $\mathcal{W}\_pq$, first recall that c may be executed a number of times for the same edge e. Thus we save each candidate edge for $c(e)$, i.e., the above edge f_ν, in a priority queue. Second recall that in between executions of $c(e)$ there may be operations $c(e')$, $e' \neq e$, that use the $\mathcal{W}$-data structure. We do not attempt to maintain separate priority queues for e and e'. Instead there is one priority queue $\mathcal{W}\_pq$ that gets used for the current edge e or e'.

An execution of $c(e)$ works as follows. Let e be in edge ε of $\overline{T}$. If this is the first execution of $c(e)$ in the $\mathcal{W}$-data structure, waken e. In general, if e was not the last edge to execute c in the $\mathcal{W}$-data structure then reinitialize $\mathcal{W}\_pq$ for e, as follows: For each $\mathcal{W}$-node ν, in $dt(\nu)$ cut edge ε; of the two trees created, let U be the one not containing ν; find f_ν as the edge corresponding to the largest key in U; restore $dt(\nu)$ by linking ε, and add f_ν to $\mathcal{W}\_pq$. (Note that the tree U can be found in a variety of ways; one simple way is to use dynamic tree operations.)

Now find the desired cocycle edge for e: If $\mathcal{W}\_pq$ is empty then return $\emptyset$. Otherwise delete the largest edge from $\mathcal{W}\_pq$. Suppose it corresponds to G-edge f in $\mathcal{W}$-edge $\mu\nu$. Delete f from the priority queue for $\mu\nu$ and find the new maximum edge. In $dt(\mu)$ update the key for ν to this edge, and similarly for $dt(\nu)$. Then find the new edge f_μ as above, add it to $\mathcal{W}\_pq$, and similarly for f_ν. Finally if $f \notin F$ (i.e., f has already been returned by some other data structure as a c value), repeat the entire procedure to find the next largest cocycle edge. When $f \in F$ remove f from F and return it as $c(e)$.

Correctness of this algorithm follows from the fact that the information in $\mathcal{W}\_pq$ is correct if $c(e)$ is executed after a previous execution $c(e)$, with no intervening executions $c(e')$ in the $\mathcal{W}$-data structure. This is true because intervening executions $c(e')$ in the other data structures do not waken any edges.

Now estimate the time for $c(e)$. Each edge f_ν is found in $O(1)$ priority queue and dynamic tree operations. Thus given $\mathcal{W}\_pq$, the time to process a cocycle edge f is $O(\log n)$. Since any f is found as a cocycle edge at most once in an Augment Step, the time is $O(m \log n)$ per Augment Step.

Finally consider the time for reinitializing $\mathcal{W}$_pq. One reinitialization uses $O(1)$ dynamic tree operations per $\mathcal{W}$-node. Thus by Lemma 4.1, the time to reinitialize $\mathcal{W}$_pq is $O(\sigma \log n)$.

We prove the time for all reinitializations is $O(\sigma^2 \log n)$ by showing that there are $O(\sigma)$ reinitializations. Observe that the number of reinitializations is at most the number of wakened edges plus the number of $\mathcal{W}$_pq's destroyed. (If an operation $c(e)$ does not waken e but reinitializes $\mathcal{W}$_pq, it is because the last operation $c(e')$ destroyed $\mathcal{W}$_pq.) A priority queue is destroyed by an edge that is wakened. So Lemma 4.1 shows the number of reinitializations is $O(\sigma)$ as desired.

We conclude this section by deriving the time bound for the weighted intersection algorithm. To do this we assume that the data structures of the next section achieve the bounds already claimed.

Theorem 4.1. The graphic matroid weighted intersection problem can be solved in time $O(\sqrt{n}(m + n \log n) \log n \log(nN))$ and space $O(m)$.

Proof. The above discussion has proved the algorithm correct. To derive the time bound, the number of scales is $O(\log(nN))$, so we analyze one scale. First consider the $\mathcal{W}$- and $\mathcal{W}1$-data structures. They use time $O(m + \sigma^2 \log n)$ per wakened graph plus $O(m \log n)$ per Augment Step. Recall that the algorithm described above is executed in graph G_x for each dual variable x. This does not change the time estimates, since the graphs G_x contain a total of m edges and together execute a total of σ swaps.

The time $O(m \log n)$ per Augment Step is within the desired bound, since Lemma 2.1(ii) shows there are $O(\sqrt{n})$ Augment Steps. Consider the time per wakened graph. An Augment Step that does s_i swaps constructs at most $\lceil s_i/\sigma \rceil$ wakened graphs. Thus Lemma 2.1 implies there are $O(\sqrt{n} + (n \log n)/\sigma)$ wakened graphs. Choosing $\sigma = \sqrt{m/\log n}$ gives time $O(m)$ per wakened graph, and total time $O(\sqrt{n}m + n\sqrt{m} \log^{3/2} n)$. Note that the second term is within the desired bound: If $m \leq n \log n$ it is at most $n^{3/2} \log^2 n$; if $m > n \log n$ it is at most $\sqrt{n}m \log n$.

Finally note that the rest of the time is within the desired bound. A scale has $O(\sqrt{n})$ Hungarian searches by Lemma 2.1(ii). The time for a Hungarian search is $O((m + n \log n) \log n)$ by Lemma 3.1. Thus the total time for Hungarian searches is within the bound. The time for 1-data structures is $O(m)$ per wakened graph plus $O(m \log n)$ time per Augment Step. The time for 2-data structures is the same, plus an additional $O(n \log^2 n)$ time per Augment Step. The time for the dynamic-base cycle problem is $O(m + n \log n)$ per Augment Step. All these terms are dominated by previous ones and so are within the bound.

Since the $\mathcal{W}$- and $\mathcal{W}1$-data structures use $O(\sigma^2)$ space, the desired space bound follows. ∎

4.3. Data structures.

This section gives the supporting data structures for the Augment Step – the 1-, $\mathcal{W}1$- and 2-data structures, in that order.

Several of the data structures use a search tree. In this standard data structure [AHU, T83], each element has a *search key* and a *priority key*, both of which are real numbers. A search tree with n elements can execute each of the following operations in $O(\log n)$ time: insert or delete an element; find an element of largest or smallest priority key subject to the constraint that its search key is in a given interval $[\ell..r]$ (alternatively, each end of the interval can be opened or closed).

Since there are a large number of interrelated data structures, it is convenient to use lazy deletion: Specifically the edges that can be returned by $c(e)$ are in the set F. When $c(e)$ returns f, it marks f as no longer in F. When a data structure retrieves a largest edge f as a possible value of $c(e)$, it checks that $f \in F$; if not, it deletes its copy of f and repeats the procedure to find a largest edge.

We begin with the 1-data structure. We show it uses time $O(m)$ per wakened graph, plus $O(m \log n)$ per Augment Step.

Recall Lemma 4.2(iv): For an edge $e \notin T$, if e is in asleep tree T then one of the subtrees of $T - e$, call it N, has no vertices of A. The 1-edges of $C(e)$ are precisely the 1-edges (with respect to T) incident to a vertex of N.

The 1-data structure represents each asleep tree T as a maximizing dynamic tree $dt(T)$. This tree is rooted at a vertex of A. The key of a vertex v is the topological number of the largest 1-edge vw not deleted from the 1-data structure. (Here 1-edge means with respect to T.) Recall that by assumption, v is on $O(1)$ edges.

The 1-data structure is initialized at $\mathcal{W}$-time. In each asleep tree T a vertex $a \in A$ is chosen, and then $dt(T)$ is constructed with root a. Initialization uses $O(m)$ time.

Now we give the algorithms for the dynamic cocycle problem. For $c(e)$, suppose e is in asleep tree T. First find the subtree N of $T - e$ that does not contain a vertex of A, as follows. Let $e = vw$. Cut e in $dt(T)$ and find the end of e in the same new tree as the root of T; say the end is v. Then N is the new tree containing w.

Find the largest 1-edge incident to N as follows. Find a maximum key in N, say for vertex x with corresponding 1-edge $f = xy$. If $f \notin F$ (it has been returned by a different data structure) then discard it, set the dynamic tree key of x to its new largest 1-edge, and repeat the procedure to find the next edge. When the desired edge f has been found, restore $dt(T)$ by linking e and return f.

Estimate the time for $c(e)$ as follows. Each edge f found in c uses $O(\log n)$ time for $O(1)$ dynamic tree operations. A given f is found at most twice in the 1-data structure in an Augment Step (once for each end). This gives time $O(m \log n)$ per Augment Step.

The 1-data structure is updated whenever asleep trees change, i.e., when an edge e wakens. The procedure is essentially the same as wakening e in the $\mathcal{W}$-data structure, so we just sketch it. Two new dynamic trees are constructed for the new asleep trees T_i, $i = 1, 2$. The edges joining T_1 to T_2 are new 1-edges (for both T_i). Previous 1-edges for T now become 1-edges for T_1 or T_2.

The time to waken one edge e is $O(m/\sigma)$, as in the $\mathcal{W}$-data structure.

Note that aside from wakening an edge, there is no need to modify the 1-data structure in a *swap* operation.

We turn to the $\mathcal{W}1$-data structure. We show it uses the time stated for the $\mathcal{W}$- and $\mathcal{W}1$-data structures in Section 4.2. Recall the characterization of 1-edges in $C(e)$ for $\mathcal{W}1$: Suppose e is in edge ε of $\overline{T}$. If $e = h, h+1$ then a 1-edge in $C(e)$ joins a type i vertex to type j, where either $i = 0$ and $h < j < k$ or $i = k$ and $0 < j \leq h$.

The data structure for ε consists of two search trees, collectively storing the 1-edges incident to T. Consider a 1-edge f joining a type i vertex to type j, $i < j$. If $i = 0$ then f is stored in the 0-*search tree* with search key j and priority key $t(f)$. Similarly if $j = k$ then f is stored in the k-*search tree* with search key i and priority key $t(f)$.

This data structure is initialized the first time $c(e)$ is executed for an edge e of ε. The $\mathcal{W}$-nodes are labelled type 0 or k. Thus a vertex in such a node can compute its type in time $O(1)$. Then the 1-edges for T are scanned and classified. Finally the 0- and k-search trees are constructed. (The labels on $\mathcal{W}$-nodes can now be discarded.)

To estimate the time, first observe that T is included in an asleep tree, by Lemma 4.2(*iii*). Thus a total of $O(m/\sigma)$ edges are scanned. Now it is easy to see that the time for one initialization is $O(\sigma + m/\sigma)$. Since there are $O(\sigma)$ edges ε, the time is $O(m + \sigma^2)$ per wakened graph.

Now consider the operation $c(e)$; assume the data structure for ε is initialized. Find the largest cocycle edge in the 0-search tree, and the largest in the k-search tree. For each, execute the lazy deletion procedure to make sure that the edge is in F. The larger of the two edges found is the desired edge.

An edge is retrieved from the $\mathcal{W}1$-data structure in $O(\log n)$ time. Since an edge is retrieved at most once per Augment Step (including retrievals for lazy deletion), the time is $O(m \log n)$ per Augment Step.

Note there is no need to process a *swap* operation in the $\mathcal{W}1$-data structure: An edge ε that

gets processed in the $\mathcal{W}1$-data structure is destroyed by the time the augmenting path is processed.

The last data structure is the 2-data structure. We begin by giving a data structure for the following *cocycle maximum problem.* Given is a graph with a fixed spanning tree T. Every nontree edge e has a real-valued key $t(e)$. We wish to process on-line a sequence of operations of the following types:

max_cocycle(e) – for $e \in T$, return an edge of maximum key in the cocycle $C(e)$;
insert(e, x) – add nontree edge e to the graph, with key x;
delete(e) – remove nontree edge e from the graph.

We first assume that initially the graph contains only T; we discuss different initialization below. We give a data structure that processes a sequence of m *insert* and *delete* operations and c *max_cocycle* operations, on a graph with n vertices, in $O(m \log n + c \log^2 n)$ time and $O(m + n)$ space. (For generality here we do not assume the graph has bounded degree, unlike Section 4.2.)

The data structure is based on the heavy path representation. Thus every nontree edge is converted to its back edges. To do this the tree T is preprocessed for nearest common ancestor operations, in $O(n)$ time; this allows an edge to be converted to its back edges (in an *insert* operation) in time $O(1)$ [HT].

Each given operation translates into corresponding operations on initial edges and final edges. For instance to do *max_cocycle*(e), first do this operation for initial edges, then for final edges, and of the two edges returned choose the one with larger key. We now discuss the data structures for initial and final edges.

First consider initial edges. For each vertex v of T record $pre(v)$, its preorder number, and $hi(v)$, the largest preorder number of a descendant. Each initial tree, with root s, is represented by a search tree storing all initial edges directed to s. An initial edge vs has search key $pre(v)$ and priority key $t(vs)$.

The search trees for all initial trees are initialized to empty in $O(n)$ time and space. An initial edge can be inserted or deleted in time $O(\log n)$. Each initial edge uses space $O(1)$.

To do *max_cocycle*(e), examine each of the $\log n$ vertices s that are roots of initial trees containing e. Examine the corresponding search tree to find the cocycle edge with largest key. (If e joins child c to parent p in T, an initial edge vs is in $C(e)$ if v is a descendant of c, i.e., $pre(v) \in [pre(c)..hi(c)]$.) The largest of the $\log n$ edges found is the largest initial edge in $C(e)$. It is computed in time $O(\log^2 n)$.

Next consider final edges. Use the same approach as Section 3: Each h-path is represented by

an interval tree, and each node of the interval tree has an initial edge data structure. The time and space are the same as for initial edges.

This completes the description of the on-line algorithms for the cocycle maximum problem. We return to algorithms for initialization. We show that if in addition to T we are given m nontree edges then the data structures can be initialized in time $O(m+n)$.

We start by showing that an interval tree I for m subintervals of $[0..n)$ can be initialized in time $O(m+n)$. This amounts to showing that the assignment of subintervals to nodes of I can be found in linear time. We show that that after $O(n)$ preprocessing time, the node of I that contains a given interval $[\ell..r]$ can be found in $O(1)$ time. Recall the description of an interval tree given in Section 3.

Without loss of generality assume n is a power of two, $n = 2^k$. The nodes at height h in I correspond to intervals $[i2^h..(i+1)2^h)$, $0 \leq h \leq k$, $0 \leq i < 2^{k-h}$. Given two integers ℓ, r, $0 \leq \ell < r < n$, write them as $k+1$ bit binary numbers. If the number of common leading bits is a (i.e., for some a bit number i, $i2^{k+1-a} \leq \ell < i2^{k+1-a} + 2^{k-a} \leq r < (i+1)2^{k+1-a}$) then the interval $[\ell..r]$ is stored in the node at height $k+1-a$ corresponding to i. Thus we need only compute the value of a for given numbers ℓ, r.

To do this precompute a $\sqrt{n} \times \sqrt{n}$ table, where the entry for i, j $(0 \leq i, j \leq \sqrt{n})$ is the number of common leading bits. Construct the table in $O(n)$ time, by computing the entry for i, j from a previous entry. To compute a for given integers ℓ, r, in $O(1)$ time determine if they have the same first $k/2$ bits. If so find a by using the table on their last $k/2$ bits; if not use the table on their first $k/2$ bits.

It is easy to do the rest of the initialization for the data structure in linear time: There are $O(n)$ search trees. They all use search keys in the range $[1..n]$. Hence the edges of all search trees can be sorted by search key in time $O(m+n)$, using a two-pass radix sort. Each search tree can be constructed in linear time, given its edges sorted by search key.

We turn to the 2-data structure. It has two parts. The first part of the 2-data structure solves essentially the cocycle maximum problem, with one difference. In the 2-data structure, $c(e)$ may be executed a number of times for the same edge e, possibly with executions for other edges e' interspersed. To handle this efficiently we use a priority queue to save candidates for $c(e)$, as in Section 4.2. The second part of the 2-data structure ensures that the edge returned by $c(e)$ is currently a 2-edge. We discuss the two parts of the 2-data structure in turn. We show the time is $O(m)$ per awakened graph, plus $O((m + n\log n)\log n)$ per Augment Step.

The first part of the 2-data structure contains the data structure for the cocycle maximum

problem. The spanning tree is the tree of matched edges at $\mathcal{W}$-time. The key $t(e)$ of edge e is its topological number. In addition there is a priority queue *2_pq*. Actually it suffices to implement *2_pq* as a linear list of $O(\log n)$ edges.

The portion of the 2-data structure for the cocycle maximum problem is initialized at $\mathcal{W}$-time, using the above initialization procedure for all unmatched edges. The time is $O(m)$.

The priority queue *2_pq* is initialized for a given edge e as follows. The above operation *max_cocycle*(e) is executed. It performs $\log n$ search tree queries in both the initial and final edge data structures, finding $2\log n$ edges in $C(e)$. (As usual lazy deletion is done in finding these edges to ensure that each is in F.) Each of these edges is stored in *2_pq*.

Next consider the c operation. Each time $c(e)$ is executed, it checks that e was the last edge to execute c in the 2-data structure. If not, *2_pq* is reinitialized for e, as above. Then, in general, $c(e)$ is found as a largest edge in *2_pq*. Details of the procedure are similar to the one for $\mathcal{W}$*_pq* in Section 4.2 and so are omitted.

Correctness of the algorithm is clear. We sketch the timing analysis, which is also similar to $\mathcal{W}$*_pq*. Each edge is retrieved from *2_pq* in $O(\log n)$ time, giving $O(m \log n)$ time per Augment Step. In one wakened graph the total number of initializations of *2_pq* is at most twice the number of distinct edges e for which $c(e)$ is executed. Operation $c(e)$ is executed for a given edge e in only one wakened graph of an Augment Step. This gives $O(n)$ initializations per Augment Step. Since an initialization uses time $O(\log^2 n)$, the total time for initialization is $O(n \log^2 n)$ per Augment Step.

We turn to the second part of the 2-data structure. Recall that the 2-data structure is used in both the Unconnecting and Connecting Cases. First consider the Unconnecting Case. In this case a 2-edge has both its ends in the same component of $S \cup W$ as e. To check this, the 2-data structure maintains a label for each asleep tree (i.e., $\mathcal{W}$-node) indicating its connected component in $S \cup W$.

After the largest cocycle edge f is found (in the first part of the 2-data structure) the algorithm checks that the ends of f are in the same component of $S \cup W$ as e. If not, f is deleted (as discussed in Section 4.2) and the next cocycle edge is found. The check adds only $O(1)$ time to the processing of f.

Each time an edge dies the connected components of $S \cup W$ in $\mathcal{W}$ are recomputed. The time to compute connected components is $O(\sigma)$ (Lemma 4.1). Since $O(\sigma)$ edges die the time per wakened graph is $O(\sigma^2)$, which is within the bound of Section 4.2.

Finally we show that the Connecting Case can also be handled by this data structure. Consider

an edge $\varepsilon \in \overline{T}$, joining ℓ to r. The tree T that defines 2-edges for ε is the subtree of $S \cup W - \{\ell, r\}$ containing all G-vertices in $\varepsilon - \{\ell, r\}$. Lemma 4.2(*iii*) shows that T is contained in an asleep tree U. (Recall that when the 2-data structure is used for an edge of ε, no edge of ε is wakened.) An operation $c(e)$ for $e \in \varepsilon$ is executed exactly as in the Unconnecting Case. In particular the test on f uses the label of asleep tree U to check that the ends of f are in the same component of $S \cup W$ as e.

To show this implementation is correct, suppose first that the ends of f are in the same component of $S \cup W$ as e. The argument is similar to Lemma 4.2(*iv*): f is a valid cocycle edge, and is either the desired 2-edge or a larger 1- or 0-edge. In the latter case f is simply discovered in two data structures. (If f is a 0-edge, it has $t(f) \leq t_0$ and the Connecting Case may switch to the $\mathcal{W}$-data structure. Even if f is a 0-edge and $t(f) = t_0$, there is no harm if the Connecting Case does not switch to the $\mathcal{W}$-data structure.)

Finally suppose the the ends of f are in different components of $S \cup W$. The argument of Section 4.2 showing that f can be deleted from the 2-data structure remains valid.

5. Cardinality matching.

This section shows how the weighted intersection algorithm can be applied to find a maximum cardinality intersection on a graphic matroid in time $O(\sqrt{n}(m + n \log n))$. For a planar graph the bound improves to $O(n^{3/2})$.

For arbitrary matroids [GX] shows several simplifications when the weighted intersection algorithm is specialized to cardinality intersection. Obviously no scaling is needed – the algorithm assigns each edge a weight of zero, so there is only one scale. Topological numbers are not needed. Thus the dynamic-base operation $c(e)$ simplifies to the following:

$c(e)$ – find an edge $f \in C_M(e) \cap F$, delete f from F and return f.

Finally, augmenting paths have the following *level property.* Recall that Δ is the total of all dual adjustment quantities so far, and G_x is the contracted graph for dual value x. Then in the Augment Step for Δ, any augmenting path consists of exactly one swap in each of the graphs G_x in M_0 and G_{-x-1} in M_1, for $x = -\Delta, \ldots, -1$. (The level property simply states that, as in the Hopcroft-Karp algorithm for cardinality graph matching, each augmenting path has shortest length, and hence goes from one level of a breadth-first search to the next.)

Consider first the Hungarian search. It simplifies to the (*unweighted*) *static-base cycle* and *cocycle problems*, defined as follows [GS]. Given is a graph with a fixed spanning tree T. There is

a set F, initialized in the cycle problem to all tree edges and in the cocycle problem to all nontree edges. The operation $sc(e)$ has e a nontree edge in the cycle problem and a tree edge in the cycle problem. It returns all edges of $C_T(e) \cap F$ and also deletes these edges from F. The problem is to process a sequence of sc operations, on-line. [GS] solves the static-base cycle problem in time $O(m)$ and the static-base cocycle problem in time $O(m + n \log n)$. The latter is improved to $O(m)$ in [G]. Thus Lemma 2.1(ii) implies that the total time for Hungarian searches in the cardinality algorithm is $O(\sqrt{n}m)$.

We turn to the Augment Step, the dominating part of the algorithm. For graphic matroids, an important consequence of the above properties is that a wakened graph in the cardinality algorithm can process more augmenting paths than in the weighted algorithm. To accomplish this redefine the parameter σ of Section 4 to be the number of augmenting paths that are allowed in one wakened graph. Note that since each augmenting path executes precisely one swap in each contracted graph G_x, the new definition still represents the number of swaps allowed in each G_x.

We show below that in one wakened graph, the total time to process a given contracted graph G_x is $O(m + \sigma^2 \log n)$ and the space is $O(m + \sigma^2)$. Let us first show that this implies the desired result.

Theorem 5.1. The graphic matroid cardinality intersection problem can be solved in $O(\sqrt{n}(m + n \log n))$ time and $O(m)$ space.

Proof. First note that the algorithm of Section 4.1 for the cycle problem of the Augment Step runs within the desired time bound. So we need only analyze the cocycle problem. Define *phase* i to consist of all the Augment Steps where $2^i \leq \Delta < 2^{i+1}$. In phase i set $\sigma = \sqrt{n/2^i}$.

We first show that the entire cocycle algorithm constructs a total of $O(\sqrt{n})$ wakened graphs. Consider any phase i. Lemma 2.1(i) implies that it processes $O(n/2^i)$ augmenting paths. Thus the number of wakened graphs that process σ augmenting paths is $O(\sqrt{n/2^i})$. Summed over all phases this gives $O(\sqrt{n})$ wakened graphs. In addition each Augment Step may contribute another wakened graph. This gives $O(\sqrt{n})$ more wakened graphs by Lemma 2.1(ii), and implies the desired conclusion.

We complete the proof by showing that each wakened graph uses $O(m + n \log n)$ time total. Clearly the term $O(m)$ in the time bound for one contracted graph G_x gives a total contribution within the bound. So consider the second term $O(\sigma^2 \log n)$.

In phase i an augmenting path has length at most $2\Delta \leq 2^{i+2}$, by the level property. Hence a wakened graph in phase i has $O(2^i)$ contracted graphs G_x. Each uses time $O(\sigma^2 \log n)$. This gives

total time $O(n \log n)$ per wakened graph, as desired.

The same reasoning implies the space bound. ∎

Before discussing the supporting data structures note some simplifications in the c routine. First consider the Unconnecting Case. There are no topological numbers, so $c(e)$ can return cocycle edges in any order. It first returns edges found in the 1-data structure, and then the 2-data structure. Also note that at any time the depth-first search path contains at most one edge from any given contracted graph G_x, by the level property. Thus at any time at most one edge e of G_x is in the process of executing $c(e)$. There is no need to worry about another edge e' destroying the data structures for e (as in the 2- and $\mathcal{W}$-data structures of the weighted algorithm.)

Next consider the Connecting Case. Only one G-edge e in a given edge ε of $\overline{T}$ will execute $c(e)$. (This follows from the preceding observation on the depth-first search path, and from the discussion of the Connecting Case in Section 4.2.) Thus there is no need for the $\mathcal{W}1$-data structure: When $c(e)$ is executed for an edge $e \in T$ the algorithm immediately wakens e and processes it in the $\mathcal{W}$-data structure.

Now we discuss the 2-, 1- and $\mathcal{W}$-data structures, in that order.

The 2-data structure is based on an algorithm for the static-base cocycle problem (defined above) for the spanning tree of matched edges at $\mathcal{W}$-time. In addition, as in the weighted case it maintains labels for asleep trees, so the components of $S \cup W$ can be identified.

To process a dynamic operation $c(e)$, execute the static operation $sc(e)$. Use each edge returned by $sc(e)$ that is a 2-edge as a value for $c(e)$, until either an augmenting path is found or the edges are exhausted.

Obviously this algorithm achieves the desired time bound $O(m)$. The correctness of $c(e)$ is clear if e is eventually declared a dead-end. Correctness may not be clear if e gets swapped in an augment – the 2-data structure effectively deletes all edges $f \in C(e)$, so some of them may never get explored. However note that e dies when it gets swapped. Thus edges $f \in C(e)$ are no longer 2-edges, and deleting them from the 2-data structure is correct.

Next consider the 1-data structure. It represents each asleep tree T as tree rooted at an arbitrary vertex of A. Each node has a list of its children and each child has a pointer to its parent.

The 1-data structure is initialized at $\mathcal{W}$-time in the obvious way. Consider the operation $c(e)$, where e joins child q to parent p. N, the subtree of $T - e$ that does not contain a vertex of A, is the subtree of q. Construct a list of all 1-edges incident to N. (Do this by examining the edges vw incident to each vertex $v \in N$. vw is a 1-edge if w is in a different asleep tree.) Use each edge in

the list as a value for $c(e)$, until either an augmenting path is found or the edges are exhausted. In the latter case delete N from the 1-data structure.

To estimate the time for $c(e)$, observe that a given asleep tree T gets processed in the 1-data structure as follows. First operations $c(e)$ for zero or more edges e eventually delete a subtree N. Then, possibly, an operation $c(e)$ eventually returns a swap e, f that becomes part of an augmenting path. In this case e gets wakened when it gets swapped in the augment, thus destroying T. This summary shows that a vertex of v gets visited at most once in the entire processing of T. Thus time $O(|T|)$ is spent on T. There are $O(\sigma)$ asleep trees, each of size $O(m/\sigma)$. This gives total time $O(m)$ per wakened graph.

The 1-data structure is updated whenever asleep trees change, i.e., when an edge wakens. The procedure simply creates the above data structure for the two new asleep trees and labels the vertices in them. The time to waken an edge is $O(m/\sigma)$, giving time $O(m)$ per wakened graph.

Finally consider the $\mathcal{W}$-data structure. It is essentially the same as Section 4.2. As in the weighted case each $\mathcal{W}$-node ν maintains its own copy of $\mathcal{D}$ as a maximizing dynamic tree $dt(\nu)$. The difference is that now, if u is the name of an asleep tree μ, $key(u)$ in $dt(\nu)$ equals one if a $\mathcal{W}$-edge $\nu\mu$ exists, else zero. In addition if $key(u) = 1$ then u points to a list $\ell(u)$ of G-edges in the $\mathcal{W}$-edge $\nu\mu$. (These lists replace the priority queues $pq(e)$ of the weighted case.) As in the weighted case the dynamic trees and edge lists use space $O(m + \sigma^2)$.

Most operations of the $\mathcal{W}$-data structure are similar to the weighted case, Section 4.2. Activating a vertex and wakening an edge both use the same time as Section 4.2, $O(m + \sigma^2)$ per contracted graph of each wakened graph. Initialization at $\mathcal{W}$-time uses the same time, $O(m)$. The operation $swap(e, f)$ uses the same time $O(\sigma \log n)$ per swap, $O(\sigma^2 \log n)$ total.

Now consider the operation $c(e)$. Let e be in edge ε of $\overline{T}$. The first time $c(e)$ is executed for e, waken e. Then, in general, process each $\mathcal{W}$-node ν as follows. In $dt(\nu)$ cut edge ε; of the two trees created, let U be the one not containing ν; in U find a vertex u with $key(u) = 1$; return G-edges in the list $\ell(u)$. (Note that vertex u can be found as a vertex of maximum key in U.) The two possible outcomes for u are that a returned edge leads to an augmenting path, or the edges of $\ell(u)$ are exhausted. In the latter case set $key(u)$ to zero and proceed to find the next vertex u of U. If all these vertices are exhausted then restore $dt(\nu)$ by linking ε and proceed to the next $\mathcal{W}$-node ν; if all of these are exhausted then e is a dead-end.

We show that the total time for c operations in the $\mathcal{W}$-data structure is $O(m + \sigma^2 \log n)$. Charge each G-edge $O(1)$ time to be returned from a list $\ell(u)$. Charge each node ν $O(\log n)$ time for the dynamic tree operations the first time it is examined in $c(e)$. The remaining overhead of

the data structure is $O(\log n)$ time to find each vertex u. (This corresponds to $O(1)$ dynamic tree operations.) Account for this time as follows. There are two possible outcomes for processing u. First suppose all edges of $\ell(u)$ get returned and $key(u)$ set to zero. Clearly u is never examined again in $dt(\nu)$. Vertex u is the name of a $\mathcal{W}$-node μ. Charging the time to μ gives $O(\sigma^2 \log n)$ time total. The second possibility is that a swap for an edge of $\ell(u)$ leads to an augmenting path. Charging the time to this swap gives $O(\sigma \log n)$ time total.

This completes the discussion of the data structures for Theorem 5.1.

To improve the time for planar graphs we solve the cycle problems more efficiently. As noted above, we need only consider the Augment Step, i.e., the dynamic cycle problem.

We classify the edges that are matched at $\mathcal{W}$-time as asleep, wakened and dead, exactly as in the cocycle problem. There is one change, which makes the asleep edges of the cycle problem a superset of those of the cocycle problem: The cycle algorithm wakens an edge f only when $swap(e, f)$ is executed. (Thus f wakens and immediately dies.)

Consider the wakened graph $\mathcal{W}$ for the cycle algorithm. Its vertices are the contractions of asleep trees. There is a spanning tree T of matched edges; it contains the edges that have been swapped into the matching.

The algorithm maintains two trees. The first is T. The second is M_0 (the matched edges at $\mathcal{W}$-time). Both trees use the same data structure: First, each node has a parent pointer and its preorder number. To do this a root vertex is chosen arbitrarily. Additionally in T, each asleep tree chooses its root as the vertex closest to the root of T. Second, set merging is done on the tree using the static tree set-merging data structure [GT85]. The sets represent subtrees of dead-ends as follows. Consider a tree edge vv', where v' is the parent of v. In M_0, v is in the same set as v' only if vv' is a dead-end. In T, let $\mathcal{W}$-edge vv' correspond to G-edge ww', where w (w') is in asleep tree v (v'). Let P denote the path of matched edges from w' to the root of asleep tree v'. Then v is in the same set as v' only if every edge of P is a dead-end.

The data structure for M_0 is initialized at $\mathcal{W}$-time using time $O(n)$. The data structure for T is reinitialized in every *swap* operation, using time $O(\sigma)$. Since at most σ swaps are done in each contracted graph G_x the latter amounts to time $O(\sigma^2)$. Note that $O(n + \sigma^2)$ also bounds the time for all merges in the two set-merging data structures.

It remains only to discuss the operation $c(e)$. It is convenient to define the operation of *path traversal.* Consider a tree whose vertices are numbered in preorder. In addition the vertices are partitioned into sets, so that if child v is in the same set as parent v' then vv' is a dead-end. We are given vertices v_0 and v_1, and wish to traverse the edges that are not dead-ends in the tree path

joining v_0 and v_1. To do this first traverse the appropriate part of the path from v_0 to the root as follows. Initialize x to v_0. In general starting at vertex x, find the root y of the set containing x; if y is not an ancestor of v_1 then traverse the edge yy', set x to y' and repeat; if y is an ancestor of v_1 then stop. When this traversal stops execute the same procedure for v_1.

The operation $c(e)$, for $e = v_0v_1$, is executed as follows. Let v_i be in asleep tree ν_i. Let α denote the nearest common ancestor of ν_0 and ν_1 in T. Let a_i denote the first G-ancestor of v_i in α (in other words, traversing the path of matched edges from v_i to the root of T, a_i is the first vertex in α). The algorithm returns edges of $C(e) \cap F$, first along the path from v_0 to a_0, then the path from v_1 to a_1, and finally the path from a_0 to a_1.

Here is the main idea to process the path from v_i to a_i. Visit successive $\mathcal{W}$-nodes μ in the path from ν_i to α, using path traversal (in tree T). To visit μ, let the first ancestor of v_i in μ be x. Let r be the root of μ. Traverse the path in μ from x to r, using path traversal (in tree M_0). The remaining details, such as doing merges for dead-ends in both T and M_0, and processing the path from a_0 to a_1, are left as a simple exercise.

It is easy to see that the time is $O(1)$ for each edge returned by $c(e)$. This concludes the discussion of the cycle problem, and shows that the time is $O(m + \sigma^2)$ for each contracted graph.

Theorem 5.2. On a planar graph, the graphic matroid cardinality intersection problem can be solved in $O(n^{3/2})$ time and $O(n)$ space.

Proof. Using the above bound of $O(m + \sigma^2)$, the analysis in the proof of Theorem 5.1 shows the time for all cycle problems in the entire intersection algorithm is $O(\sqrt{n}m)$.

Now recall from [GS] that for a planar graph G, the intersection algorithm need not solve the cocycle problem, only the cycle problem: Let G^* be the planar dual graph of G. A fundamental cocycle in G corresponds to a fundamental cycle in G^*. Thus the cocycle problem for G amounts to the cycle problem for G^*. ∎

Acknowledgments.

The first author thanks Mrs. Therese Helmintoller for her pleasant and helpful grammatical assistance.

References.

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

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

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

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

[Fre] G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications", *SIAM J. Comput., 14,* 4, 1985, pp. 781–798.

[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, "Data structures for weighted matching and nearest common ancestors with linking", *Proc. First Annual ACM-SIAM Symp. on Disc. Algorithms,* 1990, pp. 434–443.

[GGT] A.V. Goldberg, M.D. Grigoriadis and R. E. Tarjan, "Efficiency of the network simplex algorithm for the maximum flow problem", *Math. Programming,* to appear.

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

[GT85] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *J. Comp. and System Sci., 30,* 2, 1985, pp. 209–221.

[GT89] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for network problems", *SIAM J. Comput., 18,* 5, 1989, pp. 1013–1036.

[GX] H.N. Gabow and Y. Xu, "Efficient theoretic and practical algorithms for linear matroid intersection problems", Technical Rept. CU-CS-424-89, Comp. Sci. Dept., Univ. Colorado, Boulder, CO 1989; submitted for publication.

[GXa] H.N. Gabow and Y. Xu, "Efficient algorithms for independent assignment on graphic and linear matroids", *Proc. 30th Annual Symp. on Found. of Comp. Sci.,* 1989, pp.

106–111.

[HK] J. Hopcroft and R. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM J. Comput., 2,* 4, 1973, pp. 225–231.

[HT] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors", *SIAM J. Comput., 13,* 2, 1984, pp. 338–355.

[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", Technical 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.

[Me] K. Melhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry,* Springer-Verlag, New York, 1984.

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

[R] A. Recski, "Terminal solvability and n-Port interconnection problem", *Proc. 1979 ISCAS,* 1979, pp. 988–991.

[RI] A. Recski and M. Iri, "Network theory and transversal matroids", *Discrete Applied Mathematics,* 2, 1980, pp. 311–326.

[ST] D.D. Sleator and R.E. Tarjan, "A data structure for dynamic trees", *J. Comp. and System Sci., 26,* 1983, pp. 362–391.

[T79] R.E. Tarjan, "Applications of path compression on balanced trees", *J. ACM, 26,* 4, 1979, pp. 690–715.

[T83] R.E. Tarjan, *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA., 1983.

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

Figure Captions.

Figure 1.

Heavy path representation.

Figure 2.

(a) Subtree T.

(b) Connecting tree $\mathcal{C}(A_0, T)$.

(c) Contracted connecting tree $\overline{\mathcal{C}}(A_0, T)$.

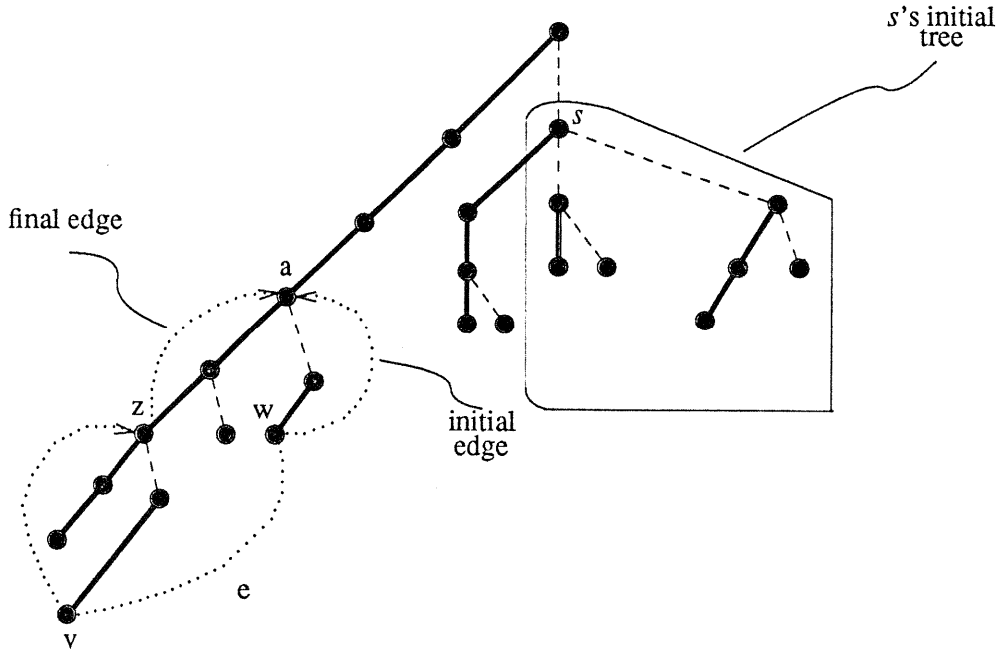
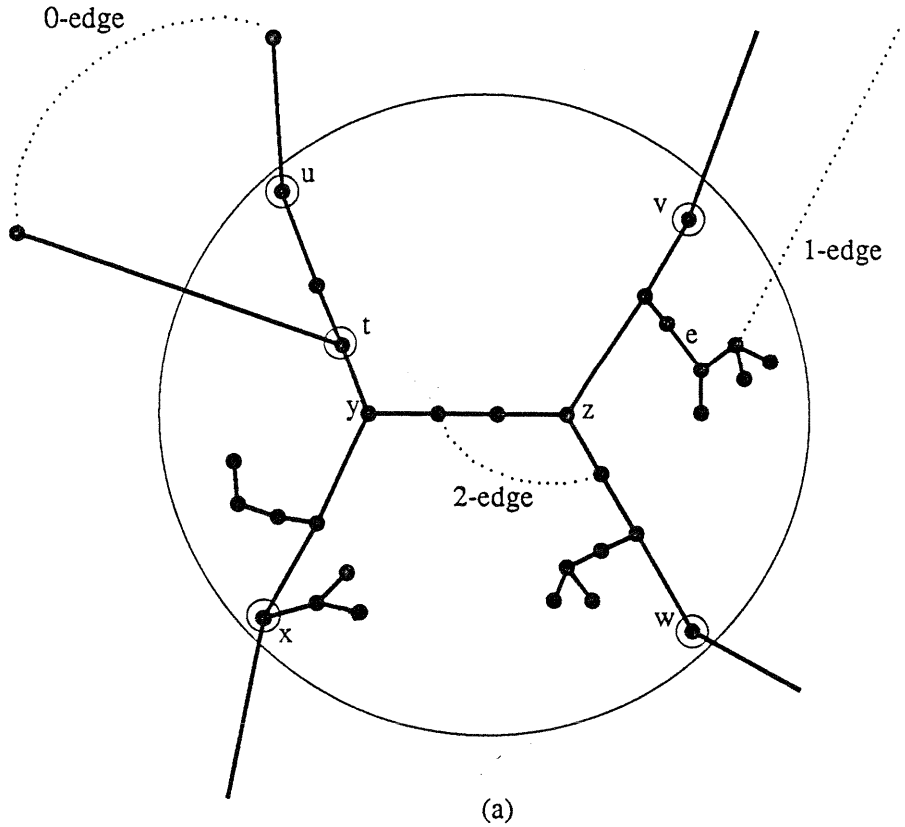
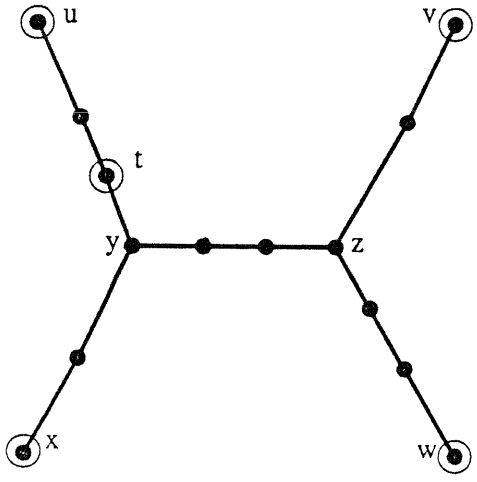


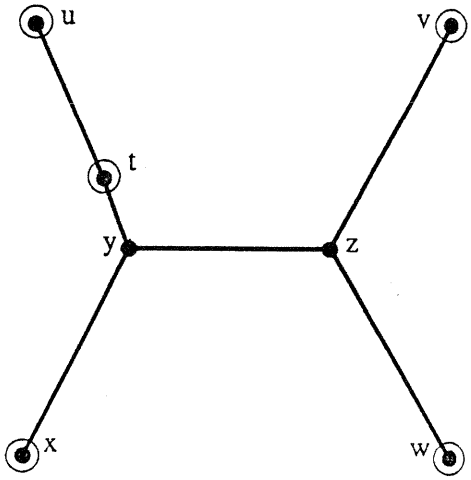
Fig. 1



(a)



(b)



(c)

Fig. 2