Efficient Algorithms for a Family
of Matroid Intersection Problems*

by

Harold N. Gabow †

Robert E. Tarjan ††

CU-CS-214-82 January, 1982

* A preliminary version of this paper appeared in the Proceedings of the 20th Annual Symposium on the Foundations of Computer Science, San Juan, Puerto Rico [GT].

† Department of Computer Science, University of Colorado, Boulder, Colorado 80309. The work of this author was supported in part by the National Science Foundation under grant MCS78-18909.

†† Computer Science Department, Stanford University, Stanford, California 94305, and currently at Bell Laboratories, Murray Hill, New Jersey 07974. The work of this author was supported in part by the National Science Foundation under grant MCS75-22870, by the Office of Naval Research, Contract N 00014-76-C-0688, and by a Guggenheim Fellowship.

Abstract. Consider a matroid where each element has a real-valued cost and a color, red or green; a base is sought that contains q red elements and has smallest possible cost. An algorithm for the problem on general matroids is presented, along with a number of variations. Its efficiency is demonstrated by implementations on specific matroids. In all cases but one, the running time matches the best-known algorithm for the problem without the red element constraint.

On graphic matroids, a smallest spanning tree with q red edges can be found in time $O(n \log n)$ more than what is needed to find a minimum spanning tree. A special case is finding a smallest spanning tree with a degree constraint; here the time is only $O(m+n)$ more than that needed to find one minimum spanning tree.

On transversal and matching matroids, the time is the same as the best-known algorithms for a minimum cost base. This also holds for transversal matroids for convex graphs, which model a scheduling problem on unit-length jobs with release times and deadlines.

On partition matroids, a linear-time algorithm is presented. Finally an algorithm related to our general approach finds a smallest spanning tree on a directed graph, where the given root has a degree constraint. Again the time matches the best-known algorithm for the problem without the red element (i.e., degree) constraint.

Key Words. matroid, matroid intersection problem, swap, graph, minimum cost spanning tree, degree constraint, matching, convex bipartite graph, job schedule, release times and deadlines, partition matroid, linear-time selection, minimum cost directed spanning tree.

1. Introduction.

Matroids offer a model for a wide variety of discrete mathematical structures. This paper investigates a combinational problem from a general matroid point of view, and also from the viewpoint of specific matroids. To state the problem, consider a matroid where each element has a real-valued cost and a color, red or green. We seek a base of the matroid that has smallest possible cost subject to the constraint that it contains exactly q red elements, for a given q.

Our problem can be viewed as a matroid intersection problem. To do this let one matroid be the given one and let a second matroid be a partition matroid induced by the coloring; we seek a minimum cost base of the two matroids. The general matroid intersection problem is polynomially-bounded, but the time bound is of high degree [L2,L1 pp.300-355] Our family of intersection problems can be solved more efficiently because of the simple structure of the second matroid.

Section 2 reviews some relevant notions from matroid theory. Section 3 presents an algorithm for our problem on general matroids, along with several variations. Applications to specific matroids are given in the following sections.

Section 4 investigates graphic matroids. Here the problem is to find a smallest spanning tree (or forest) that contains exactly q red edges. Our general algorithm runs in time $O(m \log \log_{(2+m/n)} n + n \log n)$ (n is the number of vertices, m the number of edges). This is only $O(n \log n)$ time more than the time to solve the problem without the red element constraint. Section 4 also investigates the problem of finding a smallest spanning tree where a given vertex has prespecified degree. This problem arises in the design of certain communication networks. Previous algorithms have been given [GK], the best using time $O(m \log \log_{(2+m/n)} n + n \log n)$ [Gal]. The problem is a special case of our problem, where the edges incident to the given vertex comprise one color. We show this special case can be solved in time $O(m \log \log_{(2+m/n)} n)$; more precisely, the time is the time to find one minimum spanning tree plus $O(m+n)$ extra processing. (So in this case our problem is linear-time equivalent to the problem without the red element (i.e., degree) constraint.)

Section 5 investigates matching matroids and in particular, transversal matroids for convex bipartite graphs. Our problem on the latter class corresponds to the following scheduling problem: A machine executes n unit-length jobs, chosen from a set of m jobs; each job has a cost, release time, deadline, and job class (red or green); find a smallest cost schedule with exactly q jobs of the red class. If all release times are 0, we give an $O(m + n \log n)$ time algorithm. If release times are arbitrary, we give an $O(m \log n + n^2)$ algorithm. In a general matching or transversal matroid, we give an $O(m \log m + n e)$ time algorithm (here m is the number of vertices of the graph, n is the number of edges in a maximum matching (thus $n \leq m$), and e is the number of edges in the graph.) In all cases our time bound equals the best-known bound for finding a minimum cost base of the matroid without the red element constraint.

Section 6 investigates partition matroids. Here the problem is, given a set with a partition, where each element has a cost and color; find a smallest subset containing exactly n_i elements from the i^{th} block of the partition, and exactly q red elements. We give a linear-time (and hence optimal) algorithm for this problem.

Section 7 discusses the problem of finding a smallest directed spanning tree with a prespecified root, where the root has a prespecified degree. This is closely related to our matroid intersection problem. However, since the solution is a base of three matroids rather than two, the general theory does not apply. We present an $O(m \log_{(2+m/n)} n)$ algorithm. Again this bound matches the best-known algorithm for the problem without the degree constraint.

For the algorithms that are not optimal we give a weak type of lower bound: We show that, in a well-defined sense, any algorithm using an approach similar to ours has a nonlinear lower bound. In some cases this bound matches our upper bound.

All of our algorithms easily generalize to the problem where the desired base contains at most (or at least) q red elements. Thus this paper gives evidence that a "q red elements" constraint can often be handled efficiently.

2. Matroid Preliminaries.

This section reviews some basic facts about matroids. It is assumed that the reader is familiar with an introductory treatment of matroids, such as [L1,Ch.7] or [W,Ch.1]. Our terminology comes mainly from the former. Definitions of fundamental concepts such as independent set, base, and circuit can be found in these sources.

We use a convenient shorthand notation for set operations: If S is a set and e an element, then $S + e$ is the set $S \cup \{e\}$, and $S - e$ is $S - \{e\}$. We sometimes use + instead of $\cup$, as in $B - G + R$ (for sets B, G, R). When parentheses are omitted, operators are associated to the left, e.g., $S + e - f$ is $(S+e) - f$.

We use graphic matroids to illustrate the discussion on general matroids. A graphic matroid derives from a graph or multigraph.* The elements of the matroid are the edges of the graph. The independent sets are the forests, and the bases are the spanning forests (or spanning trees, if the graph is connected). Figure 2.1 shows a graph (with solid and dotted edges); Figure 2.2 gives several bases. (We will give a more precise description of these figures shortly.)

Our results follow from one simple property of matroids, which can be included in a set of matroid axioms:

Symmetric Swap Axiom [B, Wp. 15]. If B_1 and B_2 are bases and element $f \in B_1$, then there is an element $e \in B_2$ such that both $B_1 - f + e$ and $B_2 - e + f$ are bases.

The Symmetric Swap Axiom is a strong formulation of the more standard base axiom of a matroid [L1, p. 274]. It can be derived

---

*Throughout this paper graphs are undirected unless explicitly specified otherwise. Also when our algorithms work on multigraphs we say so and then proceed to give the discussion in terms of graphs.

from the independence axiom [L1,p.268] with little difficulty. In Figure 2.2 bases B_1 and B_4 have the symmetric swap of edges 2,5.

Another important notion is the fundamental circuit $C(e,B)$ for an element e and a base B, where $e \notin B$. This is the unique circuit in $B + e$. $C(e,B)$ always exists for $e \notin B$, and $e \in C(e,B)$ [L,p270].

If S is a set of elements, deleting S from M gives the matroid M-S. Its elements are those of M, excluding S. Its independent sets are the independent sets of M that do not intersect S. If S is an independent set of elements of M, contracting S gives the matroid M/S. Its elements are also those of M, excluding S. Its independent sets (respectively, bases) are the sets I of elements of M/S such that $I \cup S$ is an independent set (base) of M. (We restrict ourselves to contracting independent sets for convenience only).

In this paper we investigate the following problem. Given is a matroid M. Each element e has a real-valued cost $c(e)$. Each element is colored either red or green. We seek a minimum cost base that contains exactly q red elements, for a given value q.

In the graphic matroid of Figure 2.1, the red edges are solid and the green edges are dotted. Each edge is labelled by its cost. For convenience we identify an edge by its cost (e.g., edge 2 is the edge of cost 2). These conventions are used in all figures of the paper. Figure 2.2 shows bases B_i, $i = 0, \ldots, 4$, where B_i is the smallest cost base with exactly i red edges.

We make the following conventions. m denotes the number of elements of M, and n denotes the number of elements in a base (i.e., the rank of the matroid).* For convenience we often omit explicit references to element costs. For instance we say an element e is a smallest element of a set S if it has the smallest cost c(e); similarly we refer to largest element, smallest base, etc. Finally, β_i denotes the set of all bases with exactly i red elements and smallest cost possible. Our problem is to find a base in β_q. Note that there are integers ℓ and u such that β_i is nonempty exactly when $\ell \le i \le u$. This is easy to prove using the independence axiom for matroids. It also follows from Theorem 3.1 below.

Our problem can be viewed as a matroid intersection problem. Define a matroid M' on the given elements, where a base of M' contains exactly q red elements and n-q green elements. M' is a partition matroid (see Section 7 for the general definition). Thus our problem is to find a smallest set that is a base in both M and M', or less precisely, a smallest intersection of M and a (special) partition matroid.

---

* Note that in a graphic matroid this convention differs from the usual one for graphs. We take n as the number of edges in a spanning forest, whereas it is usually the number of vertices in the graph. The first quantity is always less than the second, and they differ by one for connected graphs. We only use n in asymptotic estimates, and as will be seen, these two properties allow n to be interpreted either way in these estimates.

3. The General Algorithm.

This section presents an algorithm for our intersection problem on arbitrary matroids; useful variations are also given. The efficiency is illustrated for the case of graphic matroids. Subsequent sections show that the algorithm and its variants are efficient on other matroids.

The results of this section were discovered independently by Dan Gusfield [Gu]. Gusfield investigates the problem of uniformly modifying the costs of red elements. His derivation is concise and elegant, and contains all the results of this section (either explicitly or implicitly). Here we give our own development (different from Gusfield's) so the paper is self-contained. (Also, some ideas of this section for the special case of graphic matroids appear in [Ga1], [U].).

The general matroid intersection problem can be solved by augmenting paths [L1,pp.326-48]. In our problem these paths are particularly simple, as we now show.

Definition 3.1. A swap for a base B is an ordered pair of elements (e,f), where $e \in B$ is green, $f \notin B$ is red and $B - e + f$ is a base. The cost of (e,f) is $c(f) - c(e)$.

We say that (e,f) is a swap for f, for f and B, etc. Following a previous convention, we say (e,f) is a smallest (largest) swap if its cost is as small (large) as possible.

Figure 2.2 shows four swaps executed serially on the matroid of Figure 2.1. A base $B_i \in \beta_i$ derives from $B_{i-1} \in \beta_{i-1}$ by a swap. The following result shows this is true in general.

Theorem 3.1 (Augmentation Theorem). Suppose B is a base in β_{i-1} and

$\beta_i \neq \emptyset$. If (e,f) is a smallest swap for B, then $B - e + f \in \beta_i$.

Proof. It suffices to find a swap (e,f) such that $B - e + f \in \beta_i$. For this implies that (e,f) is a smallest swap for B, and further, any smallest swap for B gives a base in β_i. (Also it implies that the swap of the Theorem always exists).

Choose base $B' \in \beta_i$ such that $|B \cap B'|$ is maximum. Let f be a red element in $B' - B$ (f exists since B' has more red elements than B.) By the Symmetric Swap Axiom, there is an element $e \in B$ such that $B - e + f$ and $B' - f + e$ are bases. Clearly $e \neq f$.

We show e is green by contradiction. If e is red then $B' - f + e$ is a base with i red elements. Thus $c(B' - f + e) \geq c(B')$, so $c(e) \geq c(f)$. Similarly examining base $B - e + f$ shows $c(f) \geq c(e)$. We conclude $c(e) = c(f)$. Thus $B' - f + e$ is a base in β_i having more elements in common with B than B'. This is the desired contradiction.

Now since e is green, $B - e + f$ is a base with i red elements, and $B' - f + e$ is a base with $i - 1$ red elements. The latter implies $c(B' - f + e) \geq c(B)$, or equivalently $c(B') \geq c(B - e + f)$. This inequality shows $B - e + f \in \beta_i$, as desired. $\square$

Note that this proof is easily modified to show that $\beta_i \neq \emptyset$ exactly when $\ell \leq i \leq u$, for two integers ℓ and u (see Section 2). (If β_r, $\beta_s \neq \emptyset$ for $r < s$, show $\beta_{r+1} \neq \emptyset$ by starting with $B \in \beta_r$ and choosing $B' \in \beta_s$ as in the proof).

The Augmentation Theorem implies an algorithm for our problem: Start with a base in β_ℓ. In general, having derived a base in β_i, find a smallest swap and derive a base in β_{i+1}. Repeat this procedure until a base in β_q is derived.

We improve the efficiency of this approach in several ways. First we show that the elements involved in swaps can be drawn from a restricted set. Recall u is the largest index with $\beta_u \neq \emptyset$.

Corollary 3.1. Let B be a base in β_{i-1} and B' a base in β_u. Then there is a smallest swap (e,f) for B, with $e \in B - B'$ and $f \in B' - B$.

Proof. Let (g,h) be a smallest swap for B. We first find a smallest swap (g,f) with $f \in B' - B$. Then we find the desired swap (e,f).

The Symmetric Swap Axiom applied to h, $B - g + h$ and B' shows there is an element $f \in B'$ such that $B - g + f$ and $B' - f + h$ are bases. f is red. For otherwise since h is red, $B' - f + h$ is a base with u + 1 red elements, which is impossible.

Since f is red, $f \neq g$. Since $B - g + f$ is a base, we have $f \in B' - B$ and (g,f) is a swap for B. For (g,f) to be a smallest swap we must have $c(h) \geq c(f)$. To see this, note $B' - f + h$ has u red elements, so $c(B' - f + h) \geq c(B')$, and $c(h) \geq c(f)$. Thus (g,f) is as desired.

If $g \notin B'$, then take e = g, and swap (e,f) gives the Corollary. Otherwise find e by applying the Symmetric Swap Axiom to g, $B - g + f$, and B'. The argument is analogous to the one above and is left to the reader. □

Another useful fact is that swaps get progressively more expensive. Corollary 3.2 is a formulation of this fact that is used in Section 4.2; Corollary 3.3 is another formulation.

Corollary 3.2. Let B be a base containing a green element e. Let (e,f) be a smallest swap involving e, and set $B' = B - e + f$. For any green element $g \in B - e$, let (g,h), (g,h') be smallest swaps for g and bases B, B', respectively. Then $c(g,h) \leq c(g,h')$.

Proof. Clearly we may assume (g,h') is not a valid swap for B. Now apply the Symmetric Swap Axiom to g, B and $B-e+f-g+h'$, to show that $B-g+f$ and $B-e+h'$ are bases. The former shows (g,f) is a swap (for B), whence $c(f) \geq c(h)$. The latter shows (e,h') is a swap, whence $c(h') \geq c(f)$. Thus $c(h') \geq c(h)$, as desired. □

Corollary 3.3. The cost of a base in β_i is a convex function of i.

Proof. We need only show that the cost of an optimum swap (e_i,f_i) is nondecreasing with i. This follows from the previous result. □

This Corollary shows how to solve a modified version of our problem, where the desired base is the smallest one with at least (or at most) q red elements. To do this first find a minimum cost base. If it satisfies the red element constraint, it is the desired base. Otherwise the desired base is the smallest one with exactly q red elements.

The time for this procedure is the time to find a minimum base plus the time to solve the unmodified problem. The latter always dominates. Thus the time estimates given in Sections 4-7 also hold for the modified problem.

Returning to Corollary 3.1, we can find the desired base B_q as follows. First find B_ℓ and B_u, bases in β_ℓ and β_u, respectively. Then repeatedly swap a green element of B_ℓ for a red element of B_u, until B_q is derived.

In this approach each swap must have the smallest cost possible. The bulk of the time is spent searching for these smallest swaps. Searching is complicated by the fact that each time a swap is executed, a new base is derived. This changes the set of valid swaps, and necessitates new searching. To cut down on the searching we derive an alternate characterization of the swaps involved. This allows us to find the swaps efficiently, although in a different order.

We call the desired sequence of swaps the "swap sequence." Actually it is convenient to use this term in a slightly more general context.

Definition 3.2. Let B be a base and R an independent set of red elements. Let $B_0 = B$. Suppose for $i = 1, \ldots, r$, $B_i = B_{i-1} - e_i + f_i$, where (e_i, f_i) is a smallest swap for B_{i-1} that has $f_i \in R$; further, no swap for B_r and an element of R exists. Then (e_i, f_i), $i=1, \ldots, r$, is a swap sequence (for B and R).

When $B \in \beta_\ell$ and R is the set of red elements in a base of β_u, then the swap sequence is the one we seek. Figure 2.2 shows a swap sequence for the example matroid.

The following idea is the key to our characterization of the swap sequence. In Figure 2.1-2, consider 3, the smallest red element. The smallest swap for 3 and B_0 is (2,3). Although this is not the first swap of the swap sequence (as one might guess), it *is* in the swap sequence. It is not hard to see why: a green element in the circuit $C(3,B_0)$ cannot give a better swap than (2,3); hence $C(3,B_0)$ is preserved until swap (2,3) is made.

To state the result precisely, let h be a smallest element of R. Let (g,h) be a smallest swap for h and B. In the matroid $M - g/h$, take a swap sequence for base $B - g$ and red elements $R - h$. Insert (g,h) as the $j + 1$st/ swap, where the swap sequence begins with j swaps strictly smaller than (g,h). Call the resulting sequence S.

Lemma 3.1. S is a swap sequence for B and R.

Proof. First note that S is well-defined, i.e., in matroid $M - g/h$, $B - g$

and $R-h$ are sets that have a swap sequence: $B-g$ is a base of $M-g/h$, since $B-g+h$ is a base of M; similarly $R-h$ is independent.

Now let S be the sequence (e_i,f_i), $i=1, \ldots, r$. (So for $i = j+1$, $(e_i,f_i) = (g,h)$.) Let $B_0 = B$, and for $i = 1, \ldots, r$, let $B_i = B_{i-1} - e_i + f_i$. We must show that (e_i,f_i) is a smallest swap for B_{i-1}, for $i = 1, \ldots, t$. The argument divides into three cases: the first j swaps, the j + 1st swap (g,h), and the remaining swaps.

Consider the first j swaps. We show that for $i = 1, \ldots, j$, if B_{i-1} is a base and circuit $C(h,B_{i-1}) = C(h,B)$, then (e_i,f_i) is a smallest swap for B_{i-1}, and $C(h,B_i) = C(h,B)$. Clearly this implies the desired conclusion, by induction on i.

Base B_{i-1} is the result of executing swaps $(e_1,f_1), \ldots, (e_{i-1},f_{i-1})$ on base B, in matroid M. In matroid $M-g/h$, executing these same swaps on base $B - g$ gives $B_{i-1} - g$. Now we show a useful proposition:

For elements $e, f \notin \{g,h\}$, suppose (e,f) costs less than (g,h). Then (e,f) is a swap for B_{i-1} (in M) if and only if it is a swap for $B_{i-1} - g$ (in $M-g/h$).

Notice that, in our induction on i, i goes from 1 to j. However in this proposition, we allow $i = j+1$. The proof given below still applies, and the proposition is useful in the next case, the j + 1st swap.

The proposition is equivalent to showing that in M, $B_{i-1} - e + f$ is a base if and only if $B_{i-1} - g - e + f + h$ is a base. Since (e,f) costs less than (g,h), and $c(f) \geq c(h)$, it follows that $c(e) > c(g)$. Thus, by the definition of g, $e \notin C(h,B) = C(h,B_{i-1})$. Now suppose $B_{i-1} - e + f$ is a base, call it A. Then $A + h$ contains the circuit C(h,B), and $A - g + h = B_{i-1} - g - e + f + h$ is a base, as desired. Conversely, suppose $B_{i-1} - g - e + f + h$ is a base, call it A'. Then $A' + g$ contains C(h,B), and $A' - h + g = B_{i-1} - e + f$ is a base, as desired. This proves the proposition.

Now (e_i, f_i) is a swap for $B_{i-1} - g$ (in $M - g/h$), costing strictly less than (g,h). The proposition shows (e_i, f_i) is a swap for B_{i-1}. Also, from the proof of the proposition, $e_i \notin C(h, B_{i-1})$, and so $C(h,B_i) = C(h,B_{i-1}) = C(h,B)$. It remains to show that (e_i, f_i) is a smallest swap for B_{i-1}. Suppose, on the contrary, that a swap (e,f) costs less. If $e, f \notin \{g,h\}$, then the proposition shows (e,f) is a swap for $B_{i-1} - g$. But this contradicts the definition of (e_i, f_i). Thus $e = g$ or $f = h$. Since $C(h,B_{i-1}) = C(h,B)$, the smallest swap involving g or h is (g,h). (Recall h has smallest cost). But (g,h) costs more than (e_i, f_i). These contradictions show (e_i, f_i) is a smallest swap for B_{i-1}. This completes the analysis of the first j swaps.

For the j + 1st swap, we must show that (g,h) is a smallest swap for B_j. From the induction made for the first j swaps, $C(h,B_j) = C(h,B)$. Thus (g,h) is a swap for B_j. The proposition of that induction (valid for $i=j+1$) shows that if there is a smaller swap for B_j than (g,h), there is also a smaller one for $B_j - g$ (in $M - g/h$). The latter is false by supposition. Hence (g,h) is a smallest swap for B_j, as desired.

Finally consider the remaining swaps. We show by induction that for $i = j+2, \ldots, r$, (e_i, f_i) is a smallest swap for B_{i-1}. Base B_{i-1} is the result of executing swaps $(e_1, f_1), \ldots, (e_{i-1}, f_{i-1})$ on B. In matroid $M - g/h$, executing these same swaps, except for $(e_{j+1}, f_{j+1}) = (g,h)$, on $B - g$, gives $B_{i-1} - h$. It is easy to see that (e,f) is a swap for B_{i-1} (in M) if and only if it is a swap for $B_{i-1} - h$ (in $M - g/h$). Thus (e_i, f_i) is a smallest swap for B_{i-1}, as desired. $\square$

The Lemma can be used iteratively to find a complete swap sequence. The following definition is useful.

Definition 3.3. Let B be a base and let R be a set of red elements such that R plus the red elements of B form an independent set. Order the elements of R as h_i, $i = 1, \ldots, r$, so that the cost is nondecreasing. Let $H_0 = B$. Suppose g_i and H_i, $i = 1, \ldots, r$, are such that $H_i = H_{i-1} - g_i + h_i$ and (g_i, h_i) is a smallest swap for h_i and H_{i-1}. Then (g_i, h_i), $i = 1, \ldots, r$, is a restricted swap sequence (for B and R).

The term "restricted swap sequence" derives from the fact that we have restricted the order in which the red elements get swapped into the base. Also note that the initial condition given on R is for notational convenience only. One consequence is that $|R|$ becomes the length of a swap sequence. Figure 3.1 shows a restricted sequence.

Note that a restricted swap sequence for B and R always exists, i.e., for each element h_i, there is a swap (g_i, h_i): The circuit $C(h_i, H_{i-1})$ contains a green element, since the red elements of B and R are independent. Now g_i exists as a largest green element of $C(h_i, H_{i-1})$.

In Figure 3.1, for i = 1, 2, 3, H_i is not a base in the optimum set β_i. However a restricted sequence does give the desired swaps:

Corollary 3.4. A restricted swap sequence for B and R can be rearranged to form a swap sequence for B and R.

Proof. The proof is by induction on r. The base case r = 0 is vacuous. For r > 0, Lemma 3.1 shows (g_1, h_1) is in a swap sequence, where the remaining swaps form a swap sequence for $B - g_1$ and $R - h_1$ in matroid $M - g_1/h_1$. It is easy to check that (g_i, h_i), $i = 2, \ldots, r$ is a restricted swap sequence for $B - g_1$ and $R - h_1$. By induction these swaps rearrange to a swap sequence. The desired conclusion follows. □

To actually form the swap sequence, order swaps (g_i, h_i) of a restricted sequence as follows: Sort (g_i, h_i), $i = 1, \ldots, r$ so that their cost is nondecreasing and for swaps (g_i, h_i) of equal cost, the index i is increasing.

Corollary 3.5. A restricted swap sequence, ordered as above, is a swap sequence.

Proof. Argue as in Corollary 3.4. □

For some matroids Corollaries 3.4 and 3.5 give the best way to solve our problem. For other matroids a divide-and-conquer approach can be more efficient. We modify the results for this approach as follows: Choose B and R as in Definition 3.3. Let R_1 contain the $\lfloor \frac{r}{2} \rfloor$ smallest elements of R, $R_1 = \{h_i \mid i = 1, \ldots, \lfloor \frac{r}{2} \rfloor\}$. Let R_2 contain the remaining red elements, $R_2 = \{h_i \mid i = \lfloor \frac{r}{2} \rfloor + 1, \ldots, r\}$. Let G_1 be a set of $\lfloor \frac{r}{2} \rfloor$ green elements such that $B - G_1 + R_1$ is a smallest base whose red elements are exactly the red elements of $B \cup R_1$. Let G_2 be a subset of the remaining green elements of B such that $B - (G_1 \cup G_2) + R$ is a smallest base whose red elements are exactly the red elements of $B \cup R$.

Intuitively we expect that a restricted sequence swaps elements of G_1 with elements of R_1, and similarly for G_2 and R_2. This is correct except for slight complications due to equal-cost elements. So in the matroid $M - R_2/G_2$, let S_1 be a restricted swap sequence for base $B - G_2$ and red elements R_1. Let $G_1' = \{g \mid (g,h) \text{ is in } S_1\}$. (As indicated above, it is not necessarily true that $G_1 = G_1'$). In $M - G_1'/R_1$, let S_2 be a restricted swap sequence for $B - G_1'$ and R_2. Let $S_1 S_2$ be the sequence formed by concatenating S_2 onto the end of S_1.

Corollary 3.6. $S_1 S_2$ is a restricted swap sequence and thus can be ordered to form a swap sequence for B and R.

Proof. We start by checking that the construction is well-defined. First note the set G_1 exists: There is some set of green elements G with $B - G + R_1$ a base (e.g., $\{g_i | i = 1, \ldots, \lfloor \frac{r}{2} \rfloor\}$ in a restricted sequence). G_1 is a smallest set G. Similarly we see G_2 exists. To check that the sequence S_1 exists, note that in $M - R_2/G_2$, $B - G_2$ is a base. Further, R_1 plus the red elements of $B - G_2$ is independent, since $B - G_1 + R_1$ is a base of M containing G_2. To check that S_2 exists, note that in $M - G'_1/R_1$, $B - G'_1$ is a base, since $B - G_1' + R_1$ is a base of M. Further, R_2 plus the red elements of $B - G_1'$ is independent, since $B - (G_1 \cup G_2) + R$ is a base of M containing R_1 and these red elements.

We prove the Corollary by induction on r. The case r = 0 is vacuous. Assume r > 0. Let $S_1 S_2 = (g_i, h_i)$, $i = 1, \ldots, r$. In M, define H_i, $i = 1, \ldots, r$ by $H_i = H_{i-1} - g_i + h_i$. Now we must show that (g_i, h_i) is a smallest swap for h_i and H_{i-1}, $i = 1, \ldots, r$.

First consider the case $i = 1, \ldots, \lfloor \frac{r}{2} \rfloor$. In matroid $M - R_2/G_2$, swaps $(g_1, h_1), \ldots, (g_i, h_i)$ derive the base $H_i - G_2$. Thus H_i is a base of M, and (g_i, h_i) is a swap for H_{i-1}.

Now let (g, h_i) be a smallest swap for h_i and H_{i-1}. We show (g_i, h_i) costs no more, so it too is a smallest swap, as desired. First suppose $g \notin G_2$. Then (g, h_i) is a swap for $H_{i-1} - G_2$ in $M - R_2 / G_2$. Now the definition of (g_i, h_i) implies it costs no more than (g, h_i).

On the other hand, suppose $g \in G_2$. Consider (in matroid M) bases $B_1 = B - G_1 + R_1$ $B_2 = H_{i-1} - g + h_i$. Note $g \in B_1$. The Symmetric Swap Axiom applied to g, B_1, and B_2 shows there is an element $g' \in B_2$ such that $B_1 - g + g'$ and $B_2 - g' + g$ are bases. Note that $g' \in G_1$, since the only elements of B_2 that are not in $B_1 - g$ are in G_1. This implies, first of all, that

$c(g') \geq c(g)$, since $B_1 - g + g'$ is a base (recall the definition of B_1). It also implies $c(g_i) \geq c(g')$, since $B_2 - g' + g$ is a base containing G_2 (recall the definition of g_i). These inequalities imply $c(g_i) \geq c(g)$. Thus (g_i, h_i) costs no more than (g, h_i), as desired.

Now consider the case $i = \lfloor \frac{r}{2} \rfloor + 1, \ldots, r$. In $M - G_1'/R_1$, swaps $(g_{\lfloor \frac{r}{2} \rfloor +1}, h_{\lfloor \frac{r}{2} \rfloor +1}), \ldots, (g_{i-1}, h_{i-1})$ derive the base $H_{i-1} - R_1$. (Here it is necessary that we use matroid $M - G_1'/R_1$, rather than $M - G_1/R_1$.) It is easy to see (g, h_i) is a swap for H_{i-1} (in M) if and only if it is a swap for $H_{i-1} - R_1$ (in $M - G_1'/R_1$). This gives the desired result. □

The divide-and-conquer approach derives its efficiency from doing the computation on smaller and smaller matroids. Toward this end the following facts are useful.

Lemma 3.2. (a) Choose bases $B_\ell \in \beta_\ell$ and $B_u \in \beta_u$. Let matroid $M' = M - (\overline{B_\ell \cup B_u})/(B_\ell \cap B_u)$. Then a swap sequence for $B_\ell - B_u$ and the red elements of $B_u - B_\ell$, in M', is a swap sequence for B_ℓ and the red elements of B_u, in M.

(b) Choose $B_\ell \in \beta_\ell$. Then there is a base $B_u \in \beta_u$ such that in M' (defined as in (a)), $B_\ell - B_u$ is a base of all green elements and $B_u - B_\ell$ is a base of all red elements.

Proof. (a) Clearly $B_\ell - B_u$ and the red elements of $B_u - B_\ell$ have a swap sequence in M'; let it be (e_i, f_i), $i = 1, \ldots, r$. In M, let the sets derived from these swaps be A_i, i.e., $A_0 = B_\ell$ and for $i = 1, \ldots, r$, $A_i = A_{i-1} - e_i + f_i$. The first i swaps, which derive A_i in M, derive $A_i - (B_\ell \cap B_u)$ in M'. Thus A_i is a base of M, and (e_i, f_i) is a swap for A_{i-1}. To see that (e_i, f_i) is a smallest swap, Corollary 3.1 shows there is a smallest swap (e,f) with $e \in A_{i-1} - B_u$ and $f \in B_u - A_{i-1}$. Since e is green, $e \in B_\ell - B_u$; similarly $f \in B_u - B_\ell$. Thus (e,f) is a swap for $A_{i-1} - (B_\ell \cap B_u)$ in M'. This implies that (e_i,f_i) costs no more than (e,f), as desired.

(b) B_ℓ and B_u satisfy the desired condition if $B_\ell \cap B_u$ consists of the green elements of B_u and the red elements of B_ℓ. To find an appropriate B_u for B_ℓ, apply the Augmentation Theorem to B_ℓ until a base of β_u is derived. □

Now we give the divide-and-conquer algorithm. The procedure $A(M,q)$ below is called with M a matroid whose elements have costs and colors, and q an integer. It finds a base B with exactly q red elements and smallest cost possible. (It halts if no such base exists.)

The heart of A is the recursive procedure $S(N,B,R)$. It is called with N a matroid, B a base of all green elements, and R a base of all red elements. S finds a restricted swap sequence for B and R.

procedure $A(M,q)$;

  begin

  procedure $S(N,B,R)$;

1. **begin** **if** $|R| = 1$ **then** make (g,h) a swap, where $B = \{g\}$, $R = \{h\}$

   else **begin**

2. let R_1 be the set of $\left\lfloor \frac{|R|}{2} \right\rfloor$ smallest red elements, and $R_2 = R - R_1$;

3. let $B - G_1 + R_1$ be a smallest base whose red elements are exactly R_1;

4. $S(N-R_2/B-G_1, G_1, R_1)$; **comment** find the swaps for R_1;

5. $S(N-G_1/R_1, B-G_1, R_2)$; **comment** find the swaps for R_2;

  end **end** S;

6. let B_ℓ be a base with the minimum number of red elements, and smallest cost possible;

7. let B_u be a base with the maximum number of red elements, containing all red elements of B_ℓ, containing only green elements of B_ℓ, and smallest cost possible;

8. **if** $q < \ell$ **or** $u < q$ **then** **halt** **comment** no base with q red elements exists;

9. $S(M-(\overline{B_\ell \cup B_u})/(B_\ell \cap B_u), B_\ell - B_u, B_u - B_\ell)$;

10. let W contain the $q - \ell$ smallest swaps found by S;

    let G contain the green elements of swaps of W;

    let R contain the red elements of swaps of W;

11. $B := B_\ell - G + R$ **comment** B is the desired base

  end A;

Figure 3.2 illustrates the algorithm on Figure 2.1. The initial call to S finds the base of Figure 3.2 (a), resulting in recursive calls on the graphs of Figure 3.2 (b). The algorithm eventually finds the swaps of Figure 2.2.

For the algorithm to work correctly we must break ties in cost consistently, as in Corollary 3.5. We shall see below (Lemmas 4.1, 6.1) that in most applications of the algorithm there is sufficient time to sort. Thus we use the simple rule of Corollary 3.5: Assume the red elements have been sorted and indexed in nondecreasing order. Then in line 2, choose the $\left\lfloor \frac{|R|}{2} \right\rfloor$ elements of R with smallest index. In line 10, if there is a tie for the $q-\ell^{th}$ smallest swap, choose the one whose red element has smaller index for W. (An alternate approach to tie-breaking is given in Section 4.2).

Theorem 3.2. Procedure A finds a smallest base with exactly q red elements, if one exists.

Proof. We first check that procedure S is correct: when called with B a base of all green elements and R a base of all red elements, S finds a restricted swap sequence. We prove this by induction on $|R|$. The case $|R| = 1$ is handled correctly by line 1.

For $|R| > 1$, lines 2-5 find the restricted swap sequence $S_1 S_2$ of Corollary 3.6: The entrance conditions on B and R imply that the set G_2 of Corollary 3.6 is $B - G_1$ in the algorithm. By induction the recursive call of line 4 constructs S_1. It is easy to see that the set G_1' of Corollary 3.6 is G_1 in the algorithm. By induction the recursive call of line 5 constructs S_2. Thus S works correctly.

Now we show that A is correct. Lemma 3.2 (b) shows bases B_ℓ and B_u of lines 6-7 exist. In line 9, S finds a restricted swap sequence for

B_ℓ and B_u, by Lemma 3.2 (a). Line 11 finds the desired base B, by Corollary 3.5 and the Augmentation Theorem. □

Now we examine the efficiency of algorithm A. We do not derive a general time bound, since more accurate bounds can be given for specific matroids. We start by discussing three properties of the matroid that are desirable for an efficient implementation.

First, the divide-and-conquer approach of A depends on the ability to contract and delete efficiently. Specifically, these operations are needed in lines 4, 5, and 9.

Second, A needs an efficient algorithm for finding a minimum-cost base. This algorithm can be used for lines 3, 6 and 7. For instance line 6 is done as follows: Delete all red elements and find a smallest base B_1. Then contract B_1 and find a smallest base B_2. Finally set $B_\ell = B_1 \cup B_2$. Lines 3 and 7 are done in a similar way. (Another way to do lines 3, 6 and 7, without contracting or deleting, is to modify the cost function so the desired base has minimum cost).

The third desired property concerns the greedy algorithm. This algorithm, which finds a smallest base on any matroid, works as follows [L1,pp.275-77]: It is given a list of all elements, sorted so the cost is nondecreasing. It prunes the list to the desired base, by scanning it from beginning to end, deleting any element that forms a circuit with previous (undeleted) elements of the list.

Specific matroids often have algorithms that are faster than the greedy one. These of course are the method of choice for lines 6 and 7. However the greedy algorithm is particularly suited for line 3. Even though line 3 is repeated many times in the recursion, the sort required by the greedy algorithm need only be done once. The details are as follows.

Procedure S is called with bases B and R given as sorted lists. Line 3 finds the desired base by running the greedy algorithm on the list of elements of R_1 followed by the list of elements of B. The sorted lists for the recursive calls of lines 4 and 5 are easily constructed. Thus in procedure S, no sorting is done for the greedy algorithm. Instead, linear-time list manipulation is done inside S, and one sort is done before the first call to S.

This brings us to the third desirable property of the matroid: The greedy algorithm runs faster than other minimum-cost base algorithms if the elements are given in sorted order.

We illustrate the efficiency of algorithm A by deriving a time bound for graphic matroids. Here the problem is to find a smallest spanning forest with q red edges.

Graphic matroids have the three properties that allow efficient implementation of A. First, contraction and deletion are efficient, each requiring time $O(m+n)$. Here we assume the graph is represented by adjacency lists. To contract a set of edges, form adjacency lists for the new graph, using a linear connectivity algorithm [AHU]. Further, for each edge in the contracted graph, record the edge in M (the original graph input to A) that it derives from. This is necessary so that when swaps are formed in line 1 of A, the swaps consist of edges of M.

Note that contractions may introduce parallel edges in the graph. However it is easy to see that in A, there are at most two parallel edges, one of each color, between two given vertices. Actually, we can allow the input graph M to contain such parallel pairs.

Concerning the second property needed to implement A, there are efficient algorithms for finding a minimum cost spanning forest, in time $O(m \log \log_{(2+m/n)} n)$ time [CT,T2,Y]. And for the third property, the greedy algorithm (i.e., Kruskal's algorithm for minimum spanning trees) is even faster, $O(m\alpha(m,n))$, if the edges are already sorted [AHU, pp.172-6].

Theorem 3.3. Procedure A finds a smallest spanning forest with q red edges (if one exists) in time $O(m \log\log_{(2+m/n)} n + n\alpha(n,n)\log n)$ and space $O(m+n)$.

Proof. We first estimate the time for procedure S. S is called with N a graph consisting of two spanning forests, B, containing all green edges, and R, containing all red edges; both B and R are given as lists of edges, sorted so cost is nondecreasing. n, the rank of the matroid, is the number of edges in each of B and R. Line 2 uses $O(n)$ time. Line 3, using the greedy algorithm, is $O(n\alpha(n,n))$. The deletions and contractions in lines 4 and 5 are $O(n)$. So if $t(n)$ is the time required by S on a graph of rank n, there are constants c_1, $c_2 > 0$ so that

$$t(n) \le c_1 n\alpha(n,n) + t(\lfloor \tfrac{n}{2} \rfloor) + t(\lceil \tfrac{n}{2} \rceil), \text{ for } n > 1;$$
$$t(1) = c_2.$$

It follows by induction that $t(n)$ is $O(n\alpha(n,n)\log n)$.

The space needed by S is $O(n)$. For suppose each recursive call stores the lists B and R. If $S(n)$ is the space required on a graph of rank n, there are constants c_1, $c_2 > 0$ so that

$$S(n) \le c_1 n + S(\lceil \tfrac{n}{2} \rceil), \text{ for } n > 1;$$
$$S(1) = c_2.$$

The desired bound on S follows.

Now suppose A is called on a graph of rank n, with m edges. Lines 6-7, using an efficient minimum spanning tree algorithm, are $O(m \log\log_{(2+m/n)} n)$. Line 9 is $O(n\alpha(n,n)\log n)$. (This includes the time to sort the edges in the spanning forests $B_\ell - B_u$ and $B_u - B_\ell$.)

Line 10 is $O(n)$, using a linear selection algorithm [BFPRT,SPP]. The desired time bound for A follows. The space is obvious. □

The next section improves this time bound by eliminating the factor $\alpha(n,n)$ in the second term. Subsequent sections apply A to other matroids. The efficiency can be estimated by computations similar to Theorem 3.3.

Sections 4-6 all use algorithms that are variants of A. One simple variant is to replace procedure S by a procedure that computes a restricted swap sequence directly from Definition 3.3. This approach, coupled with data structures and algorithms that capitalize on special features of the matroid, gives our best algorithms.

4. Graphic Matroids

This section discusses our problem on graphic matroids. As mentioned before, the problem here is to find a smallest spanning forest with q red edges. First we use the "dynamic tree" data structure of Sleator and Tarjan [ST] to solve the problem in $O(m \log \log_{(2+m/n)} n + n \log n)$ time. Then we discuss a special case of the problem - finding a smallest spanning tree with a degree constraint. We show this case is linear-time equivalent to finding an (unconstrained) minimum spanning tree.

4.1. Spanning Forests

This section shows that for graphic matroids a restricted swap sequence can be rapidly computed from the definition. It also gives a lower bound to show that no implementation of the swap sequence approach can be faster.

The dynamic tree data structure allows a number of operations, including the following:

- find max (v) - return an edge of maximum cost on the tree path from v to the root;
- evert (v) - modify the tree so that v is the root;
- cut (v,w) - delete the tree edge (v,w);
- link (v,w,x) - link the trees containing v and w by adding edge (v,w), setting the cost of (v,w) to x.

The time for a series of m such operations is $O(n+m \log n)$ [ST].

It is easy to see how to use these operations to compute a restricted swap sequence. We maintain the tree T so that any red edge in T has a modified cost equal to $S-1$, where S is the smallest (original) cost of any edge. Then to compute the smallest swap for a red edge $h = (v,w)$, make v the root, by evert(v); find the edge g in the smallest swap (g,h), by find max(w); and execute the swap, by cut(g), link(v,w, S-1). It is easy to see that the n swaps of a restricted swap sequence are found in time $O(n \log n)$.

Now we sketch a complete algorithm for our intersection problem on graphic matroids, based on algorithm A. First find a smallest spanning forest B_ℓ with the minimum number of red edges (line 6 of A). To do this, use an efficient minimum spanning tree algorithm, with one modification: When the costs of edges of different colors are compared, always declare the red edge to be larger. The time is $O(m \log \log_{(2+m/n)} n)$. [T2]. Similarly compute base B_u (line 7). Then form a new graph by contracting $B_\ell \cap B_u$ and deleting all edges but $B_\ell \cup B_u$. Find a restricted swap sequence in this graph, proceeding as described above. Finally as in lines 10-11, find the desired base B.

Theorem 4.1. A smallest spanning forest with q red edges can be found, if it exists, in time $O(m \log \log_{(2+m/n)} n + n \log n)$ and space $O(m+n)$. □

Now we give lower bounds that indicate how close to optimal our algorithm is. For these we define two timing functions: tree(m,n) is the time required by an optimal algorithm to find a minimum spanning tree on a graph of m edges and rank n; sort(n) is the time required to sort n numbers. Note tree and sort are both defined for algorithms on Random Access Machines. The comparison tree model offers strong evidence that sort is $\Theta(n \log n)$ [F], and also some evidence that tree is $\theta(m \log \log_{(2+m/n)} n)$ [CT, T2].

It is clear that any algorithm for a smallest spanning tree with q red edges requires time $\Omega(\text{tree}(m,n))$ (let all edges be green and take q=0). Also from the above discussion, our algorithm uses time $O(\text{tree}(m,n) + n \log n)$. So the algorithm is at most $O(n \log n)$ above optimal.

We can show that among algorithms using the swap sequence approach, ours is optimal to within a constant factor. To be precise, say that an algorithm "uses the swap sequence approach" if, given a graph consisting of two spanning trees, one green and one red, it finds the swaps (e_i, f_i) that constitute the swap sequence. Note the algorithm need not determine the correct order of the swaps. For example, procedure A uses the swap sequence approach.

Lemma 4.1. Any algorithm using the swap sequence approach requires time $\Omega(\text{sort}(n))$.

Proof. We give a procedure that sorts n arbitrary numbers. It works by constructing a rank n+1 graph, calling the algorithm, and then processing the swaps given by the algorithm to find the sorted order. Excluding the time for the algorithm, the procedure uses $O(n)$ time. Since $c_1 n \le \text{sort}(n) \le \text{sort}(n-1) + c_2 n$ this suffices to prove the Lemma.

Consider n numbers $x_1, \ldots, x_n$. Without loss of generality assume all numbers are positive (otherwise increase the numbers by a constant). Let M be the largest of these numbers plus one. The graph for $x_1, \ldots, x_n$ is shown in Figure 4.1. Formally, the graph has vertices v, w, and u_i, $i = 1, \ldots, n$; green edges (v, u_i) costing $-x_i$ and red edges (w, u_i) costing x_i, $i = 1, \ldots, n$; green edge (v,w) costing 0 and red edge (v,w) costing M.

To specify the swap sequence of this graph, let the given numbers in nondecreasing order be $y_1, \ldots, y_n$. Then identifying each edge by its cost, the swap sequence is $(0, y_1)$, $(-y_1, y_2), \ldots, (-y_{n-1}, y_n)$, $(-y_n, M)$.

The procedure constructs the graph and calls the algorithm to find the swaps of the swap sequence. Then it encodes the swaps into an array S, by setting $S(i) = j$ if $((v,u_i),(w,u_j))$ is a swap, and $S(0) = j$ if $((v,w),(w,u_j))$ is a swap. Finally it outputs the sequence $S(0), S^2(0), \ldots, S^n(0)$. It is easy to see this gives the indices of the numbers in sorted order. Furthermore, the time spent before and after the algorithm is $O(n)$, as desired. $\square$

Corollary 4.1. Any algorithm that finds a smallest spanning tree with q red edges using the swap sequence approach requires time $\Omega(\text{tree}(m,n) + \text{sort}(n))$. $\square$

4.2. Spanning Trees with a Degree Constraint.

Now we turn to a special case of our problem on graphic matroids that has some practical significance. An important question in network design is how to link a central computer, having a limited number of communication channels, to a collection of peripheral computing sites. Some versions of this problem, such as the Capacitated Tree Problem, are NP-complete [P]. Closely related but tractable is the problem of finding a smallest spanning tree such that a given vertex v has a specified degree. Polynomial algorithms have been presented for this problem [GK]; the most efficient uses time $O(m \log \log_{(2+m/n)} n + n \log n)$ [Ga1]. We show here that the problem is equivalent to finding a minimum spanning tree. More precisely, we give an algorithm for the problem that finds one minimum spanning tree and then does $O(n)$ postprocessing.

An exact statement of the degree-constrained spanning tree problem is as follows: Given a connected graph with real-valued edge costs, find a spanning tree with smallest possible cost such that the degree of a given vertex v is exactly p. We solve the following problem: Given a connected graph with edge costs and colors, such that all green edges are incident to v, find a smallest spanning tree with q red edges. This problem includes the degree-constrained spanning tree problem if we take $q = n-p$, but it allows red edges incident to v. (Note that in both problems the restriction to connected graphs is for convenience only.)

The problem is simplified by making the desired tree unique. This can be done in a variety of ways. Here we take the desired tree to be lexicographically minimum. That is, assume the edges of the graph are indexed from 1 to m. For any set of edges, form a vector by arranging its edge indices in increasing order, i.e., for $\{e_{i_1}, e_{i_2}, \ldots, e_{i_k}\}$ where $i_1 < i_2 < \ldots < i_k$, form $(i_1, i_2, \ldots, i_k)$. Now among all smallest spanning trees with q red edges, our algorithm finds the one whose vector is lexicographically minimum.

Note that this tree is the unique smallest spanning tree with q red edges with respect to a certain cost function c'. To define c', set $\epsilon = \frac{1}{2m}(\min\{|c(S)-c(T)| \mid S$ and T are sets of edges with $c(S) \neq c(T)\} \cup \{1\})$, and set $c'(e_i) = c(e_i) - \epsilon^i$. It is easy to check that for any two sets S, T, $c'(S) < c'(T)$ if and only if $c(S) < c(T)$ or $c(S) = c(T)$ and S is lexicographically smaller than T. This implies the desired property of c'.

Note also that no two swaps have the same cost c': It is easy to check that $c'(e_i,e_j) < c'(e_k,e_\ell)$ if either $c(e_i,e_j) < c(e_k,e_\ell)$ or $c(e_i,e_j) = c(e_k,e_\ell)$ and the vector for $\{e_i,e_\ell\}$ is smaller than the vector for $\{e_j,e_k\}$.

In the lemmas that follow, we assume the cost function c has been changed to c'. Hence no two edges or swaps have the same cost. In the algorithm, we can use c' without explicitly calculating it, by using the characterization of c' in terms of c specified above.

We make two more assumptions. First, the green edges form a spanning tree. Second, some of the red edges form a spanning tree, and the red edges not in the tree are incident to v.

(Note that by preprocessing as in procedure A, the general case reduces to the case where the green and red edges are spanning trees. We shall see that our algorithm may introduce extra red edges incident to v.)

Now let T_i, $i = 0, \ldots, n$ be the smallest spanning tree with exactly i red edges. Let $T_i = T_{i-1} - e_i + f_i$, $i = 1, \ldots, n$, so that (e_i, f_i), $i = 1, \ldots, n$, is the swap sequence. As noted above, T_i, e_i and f_i are unique.

Our approach is to start with T_0 and repeatedly find sets of edges that are in the first q swaps. We do not find the swaps themselves. The following simple concept is central in deducing the edges in swaps.

Definition 4.1. For a green edge $e = (v,w)$, let $\mu(e)$ be the smallest red edge incident to w. Equivalently, $(e,\mu(e))$ is the smallest swap for e and T_0.

Note $\mu(e)$ always exists, since the red edges span. Let $M = \{(e,\mu(e)) \mid e \text{ is green}\}$. The M-swaps for a graph are shown in Figure 4.2. The next four lemmas show how the M-swaps allow us to deduce edges in T_q.

Lemma 4.2. Suppose $(e,\mu(e))$ is not a valid swap for some optimal tree T_i, $0 \le i \le n$. Then $\mu(e) \in T_i$.

Proof. Choose i as small as possible such that $(e,\mu(e))$ is not valid. Clearly $i > 0$. It suffices to prove the lemma for T_i, since this implies $\mu(e) \in T_j$ for $j > i$.

First note that either $e \notin T_i$ or $\mu(e) \in T_i$. For otherwise $e \in T_i$ and $\mu(e) \notin T_i$. Since $(e,\mu(e))$ is not valid, the cycle $C(\mu(e),T_i)$ does not contain e. This implies $C(\mu(e),T_i)$ does not pass through v. But then $C(\mu(e),T_i)$ consists entirely of red edges, contradicting the fact that the red edges not incident to v are acyclic.

Thus without loss of generality $e \notin T_i$. Hence T_i derives from T_{i-1} by doing the smallest swap for e. $(e,\mu(e))$ is valid for T_{i-1}, by definition. Thus Corollary 3.2 implies it is the smallest swap for e, and $\mu(e) \in T_i$. □

<u>Lemma 4.3</u>. If $(e,\mu(e))$ is among the q smallest swaps of M, then $\mu(e) \in T_q$.

<u>Proof</u>. Among the first q swaps (e_i,f_i), $i = 1, \ldots, q$ of the swap sequence, there is one with $c(e_i,\mu(e_i)) \geq c(e,\mu(e))$. Corollary 3.2 implies $c(e_i,f_i) \geq c(e,\mu(e))$.

Now if $(e,\mu(e))$ is a valid swap for T_{i-1}, it is the smallest swap, and thus $(e,\mu(e)) = (e_i,f_i)$. (Recall no two swaps gave the same cost.) Hence $\mu(e) \in T_q$. Otherwise, if $(e,\mu(e))$ is not valid, Lemma 4.2 implies $\mu(e) \in T_q$. □

In Figure 4.2, the lemma implies edges 6 and 8 are in T_3, as shown in Figure 4.3. Notice the lemma does not give $q = 3$ distinct red edges in T_q. In general, the lemma need only give $\left\lceil \frac{q}{2} \right\rceil$ red edges, since a red edge can be the μ - value of two green edges.

<u>Lemma 4.4.</u> If $(e,\mu(e))$ is not among the 2q-1 smallest swaps of M, then $e \in T_q$.

<u>Proof</u>. We argue by contradiction. Suppose $e \notin T_q$. Then some swap (e_i,f_i), $i = 1, \ldots, q$, in the swap sequence has $e_i = e$. From Corollary 3.2, $c(e,\mu(e)) \leq c(e_i,f_i)$. Clearly none of the 2q-1 swaps of M that are strictly smaller than $(e,\mu(e))$ is valid for T_{i-1}. These swaps contain at least q distinct red edges, and Lemma 4.2 shows they are all in T_{i-1}. Since $i - 1 < q$, this is the desired contradiction. □

This Lemma is illustrated by edge 2 in Figures 4.2-3.

Lemmas 4.3-4 allow us to deduce edges in T_q. Now we show how these lemmas can be iterated. Let F be any set of edges in T_q. Form a multigraph H as follows. First form G/F. In a slight abuse of notation, let v denote the vertex of G/F that contains the original vertex v of G. If G/F contains parallel edges, they are incident to v (since two parallel edges not incident to v give a cycle of red edges in G-v, a contradiction.) Now for every set of parallel edges (all incident to v) of the same color, delete all but the edge of smallest cost. The resulting multigraph is H. (This is illustrated in Figure 4.4 (a).) We show that T_q corresponds to an optimal tree in H. Let p be the number of red edges in F.

Lemma 4.5. In the multigraph H, the edges T_q-F form the smallest spanning tree with q-p red edges.

Proof. First note that all edges of T_q-F are in H, i.e., none are deleted. For if an edge $e \in T_q$ is parallel to an edge f, clearly $f \notin T_q$, since T_q is acyclic. If f has the same color as e but smaller cost, then the spanning tree T_q-e+f contradicts the definition of T_q. So e is not deleted.

Now it is clear that T_q-F is a spanning tree of H containing q-p red edges. Further, it is the smallest such tree. For if S is a smaller tree, then S+F is a spanning tree of G that contradicts the definition of T_q. □

Figure 4.4 shows H and T_3-F, the smallest spanning tree with one red edge. Note that in H, the smallest spanning tree with no red edges is {1,3,9}. This tree corresponds to a spanning tree of G with two red edges, {1,2,3,6,8,9}, but this is not T_2 = {1,2,3,6,7,9}.

The last result allows us to iterate Lemmas 4.3-4 until all edges of T_q are deduced. This is the method followed by procedure B given below.

On entry to B, the graph (a multigraph) consists of two spanning trees, one green and the other red. All green edges are incident to v; red edges may or may not be incident to v. A spanning tree containing q red edges is to be found. (By assumption, such a tree exists.)

B adds edges to a list F until it is the desired tree. q is maintained as the number of red edges to be added.

```
    procedure B;
    begin
1.  while the graph has more than one vertex do
        begin
2.      for each green edge e = (v,w) do
            μ(e):= the smallest red edge incident to w;
3.      let M = {(e,μ(e))|e is a green edge};
        let M1 contain the q smallest swaps of M;
        let M2 contain all but the 2q-1 smallest swaps of M;

4.      add the red edges of swaps of M1 to F; decrease q by the number
           of edges added;
5.      add the green edges of swaps of M2 to F;
6.      contract the edges of F;
7.      for each vertex of the graph w ≠ v do
           delete all edges from w to v except the smallest green and
             smallest red edge (if it exists);
end     end  B:
```

When B is called with $q = 3$ in Figure 4.2, the first iteration forms the graph of Figure 4.4 (a); the second iteration adds the edges of Figure 4.4 (b) to F, completing the tree T_3.

Note that in lines 2, 3, and 7, ties in cost c are broken lexicographically, consistent with the cost function c' described at the start of this section.

Also in line 6, it is only necessary to contract the edges added to F in the current iteration. As usual, we do the contraction by forming adjacency lists for the new multigraph. Further, for each edge in the current graph we record the edge in the original graph from which it derives. This way we can maintain F as a list of edges in the original graph.

Lemma 4.6. Suppose B is called with the entry conditions (given above B) satisfied. Then B adds edges to F that form the smallest spanning tree with q red edges. B runs in $O(n)$ time and space.

Proof. We first prove correctness. We show that every time line 1 is reached, the following conditions hold:

(i) The desired tree consists of F plus the smallest spanning tree with q red edges in the current graph.

(ii) The green edges form a spanning tree, with each green edge incident to v.

(iii) The red edges consist of a spanning tree and zero or more edges incident to v.

The entry conditions show (i)-(iii) hold initially. So suppose lines 2-7 are executed. The edges added to F in lines 4-5 are in the desired tree, by Lemmas 4.3-4. (Note if $q = 0$, line 5 adds all green edges to F, as desired). Lines 6-7 form the graph H of Lemma 4.5. Hence after line 7, (i) holds. (ii) is obvious. To see (iii), first note that the red edges not incident to v in the new graph are acyclic (otherwise the original graph has a red cycle missing v). Further, the red edges span the new graph. (iii) follows. This completes the induction.

Finally note that every time through the loop, if $q > 0$ the iteration decreases q; if $q = 0$, the loop halts. Thus B eventually halts. Now (i) shows F is as desired.

Now we estimate the efficiency of B. First observe that one iteration of lines 2-7 takes time linear in the number of edges (in the current graph). Line 3 uses a linear-time selection algorithm [BFPRT,SPP].

Define the following quantities, for $i \geq 1$: q_i is the value of q immediately before the i^{th} iteration of the *while* loop; g_i (r_i) is the number of green (red) edges in the graph immediately before the i^{th} iteration. The following relations hold, for $i \geq 1$:

(1) $q_{i+1} \leq q_i/2,$

(2) $g_{i+1} \leq 2q_i - 1,$

(3) $r_i \leq 2g_i - 1.$

(1) holds because the i^{th} iteration adds the red edges of M_1 to F. (2) holds because the i^{th} iteration adds the green edges of M_2. (3) follows from inductive assertions (ii) - (iii).

Now the i^{th} iteration uses time $O(g_i + r_i)$. For $i = 1$ this is $O(n)$, by (ii) - (iii). Otherwise (2) - (3) show that for $i \geq 1, r_{i+1}$ and g_{i+1} are both $O(q_i)$. (i) shows $q_i \leq q_1/2^{i-1} \leq n/2^{i-1}$. Hence all iterations after the first take time at most a constant times $\sum_{i=1}^{\infty} \frac{n}{2^{i-1}} = O(n)$.

This gives the desired time bound.

The space bound is obvious. (Note that we only maintain adjacency lists for the current graph.) □

The main routine for finding a degree-constrained spanning tree is similar to procedure A: First find trees T_ℓ and T_u (lines 6-7 of A). Then check q for feasibility (line 8). If q is feasible, decrease its value by ℓ, and place the ℓ red edges of T_ℓ in F. Then delete all edges besides $T_\ell \cup T_u$, and contract $T_\ell \cap T_u$. After this step the green edges and the red edges both are spanning trees. Finally call B.

<u>Theorem 4.2</u>. The above algorithm finds a smallest spanning tree with a degree constraint, in time $O(m \log\log_{(2+m/n)} n)$ and space $O(m)$.

<u>Proof</u>. T_ℓ and T_u are found by using a minimum spanning tree algorithm on the graph with edge costs appropriately modified. The time for this is $O(m \log\log_{(2+m/n)} n)$ [CT, Y, T2]. The new graph can be constructed from the $O(n)$ edges in $T_\ell \cup T_u$ in time $O(n)$. B is $O(n)$. The time bound follows. □

Actually the algorithm can be implemented with only one call to a general minimum spanning tree algorithm. First find T_u, the spanning tree with the greatest number of red edges and smallest cost possible. Use the general algorithm for this. Then find T_ℓ, the spanning tree with the fewest number of red edges, containing all green edges of T_u, only red edges of T_u, and smallest cost possible. Note that after the cost function has been appropriately modified, this can be done by finding a minimum spanning tree on a graph consisting of a spanning tree T_u and edges incident to one vertex v. This requires time O(n), by an algorithm originally due to Spira and Pan: Use procedure B, with lines 2-5 replaced by one step that adds the smallest edge incident to each vertex in the graph to F. (Also, ignore colors in line 7.) Details are in [SP, p. 377]. This gives the following result:

Corollary 4.2. A smallest spanning tree with a degree constraint can be constructed by modifying costs (in time $O(m)$), finding one minimum spanning tree, and doing O(n) post-processing. □

Note that finding a smallest spanning tree with a degree constraint requires at least the time to find a minimum spanning tree. (We can find a minimum spanning tree for a graph by adding a vertex v with one edge, and finding a smallest spanning tree with one edge incident to v.) Hence the two problems are linear-time equivalent.

5. Matching Matroids.

This section discusses our intersection problem on several types of matching matroids. First the matroids are defined. Then an $O(m + n \log n)$ algorithm is given for simple scheduling matroids. An $O(m \log n + n^2)$ algorithm is given for general scheduling matroids. Finally, the time for general transversal and matching matroids is shown to be $(m \log m + n e)$.

We begin with the definitions. A matching matroid is derived from a graph G and a subset of the vertices J. The elements of the matroid are the vertices of J. The independent sets are subsets of J that can be covered by a matching of G. A transversal matroid is a matching matroid where G is bipartite and J is one of the two vertex sets of G (i.e., each edge goes from J to $\bar{J}$).

Two special types of transversal matroids are of particular interest. A (general) scheduling matroid derives from a convex bipartite graph. More precisely, the vertices of $\bar{J}$ can be indexed from 1 to $|\bar{J}|$ so that each vertex of J is adjacent to consecutive vertices a, a+1, ..., b of $\bar{J}$ (and no others). We think of the vertices of J as jobs and the i^{th} vertex of $\bar{J}$ as the time period from i-1 to i. Thus a scheduling matroid corresponds to the following situation: A processor runs for $|\bar{J}|$ units of time. There are $m = |J|$ jobs, each requiring one unit of processing time. Each job j has a release time r_j and a deadline d_j, both integers, $0 \le r_j < d_j \le |\bar{J}|$. If job j is chosen for execution, it must be started no earlier than time r_j and finished no later than d_j. In this matroid a base is a set of jobs of maximum cardinality that can be executed on the processor so that each job meets its constraints. A schedule is a base, together with a specification of when each job gets executed.

A simple scheduling matroid [L1 pp.265-6, 278] corresponds to a convex bipartite graph where each vertex of J is adjacent to vertices 1,2,...,b, for some b. In the scheduling interpretation, each release time is 0, so release times can be ignored.

In the following discussion we use interval notation in two ways. The first is for intervals of time. Suppose a processor executes n unit-length jobs, one after another. We say that the i^{th} job is executed in the time interval $[i-1,i)$. (Thus the i^{th} job is no longer executing at time instant i). The second use employs interval notation for sets of integers. Thus if r and d are integers, $[r,d)$ denotes the integers in the set of real numbers usually denoted $[r,d)$, i.e., $[r,d) = \{r,\ldots,d-1\}$. The context will always make our use of interval notation unambiguous.

5.1. Simple Scheduling Matroids.

This section discusses the following scheduling problem. Given is a set of m unit-length jobs. Each job has an integer deadline, a real-valued profit and a job class G or R. The profit for a job is earned if and only if it is completely executed by its deadline. Find a maximum profit schedule containing exactly q jobs in class R.

This problem is essentially equivalent to our intersection problem on simple scheduling matroids. Note that we will give an algorithm that finds a schedule, not just an (unordered) base. Also, as usual, without loss of generality we find a minimum (not maximum) cost base (schedule). An example problem is shown in Figure 5.1. Jobs are identified by their cost and listed underneath their deadline.

We represent the matroid by the following data structure. For each integer time t between 1 and the largest deadline, there is a list $D_t = \{j | \text{job } j \text{ has } d_j = t\}$. The list heads are in an array so that given t, the first job in D_t can be found in O(1) time. It is easy to construct this data structure in linear time from any reasonable specification of the matroid.

We start with a simple normalization: We can always assume that the processor runs from time 0 to n, and exactly n jobs are executed.

Lemma 5.1. Let M be a simple scheduling matroid. Then there is a simple scheduling matroid M', where M' has rank $n' = \max\{d_j | j \text{ is a job of } M'\}$, $m' < (n')^2$, and for some set of jobs I of M, I plus a base of M' gives a base of M (i.e., $M' = M/I$). Further if the jobs have costs, then I plus a minimum cost base of M' gives a minimum cost base of M. M' can be found in time O(m).

Proof. First we prune M to a matroid M' with all the desired properties except the upper bound on m. To do this, initialize M' to M and I to $\emptyset$. Then set $n' = \max\{d_j | j \text{ is a job of } M'\}$. Now let $J_t = \{j | j \text{ is a job of } M' \text{ and } d_j > t\}$. Suppose there is a time t such that $|J_t| = n' - t$. Then choose t as large as possible; remove the jobs of J_t from M' and add them to I; then repeat the process (start by redefining n'). Otherwise if there is no time t, halt.

The correctness of this procedure follows from the observation that the deleted sets J_t are included in any base of M; also, the final matroid M' has rank n'. It is easy to implement this procedure in linear time. (Note that if we seek a schedule rather than just a base, the assignment of I to time periods can also be recorded in linear time).

In the second modification to M', for each d, $1 \le d \le n'$, remove all but d smallest jobs that have deadline d. (If fewer than d jobs have deadline d, do not remove any. If jobs do not have costs, the choice of jobs to remove is arbitrary). It is easy to see that this does not change the minimum cost of a base, and achieves the bound $m' \le \frac{n'(n'+1)}{2}$. Further this can be done in the required time by using a linear-time selection algorithm [BFPRT,SPP]. □

For the rest of the discussion we assume the matroid has been preprocessed so M has the properties given above for M'.

Now we discuss finding a minimum cost schedule. The greedy algorithm can be implemented in time $O(m \log m + n)$, or if the jobs are already sorted, $O(m\alpha(m,n)+n)$. This is a simple exercise using the UNION-FIND data structure [HS, pp.161-8].

The time bound for unsorted jobs is improved to $O(m+n \log n)$ by the following algorithm. Recall $D_t = \{j | d_j = t\}$. The algorithm also uses A, a priority queue of jobs.

```
    procedure C;
    begin
1.  A: = ∅;
2.  for t: = n to 1 by -1 do
        begin
3.      A: = A ∪ D_t;
4.      remove the smallest job from A (if one exists), and schedule
          it in time interval [t-1,t);
    end end C;
```

Lemma 5.2. Procedure C finds a minimum cost schedule for a simple scheduling matroid in time $O(m + n \log n)$ and space $O(m)$.

Proof. We start by showing that if j is a smallest job with deadline n, there is a minimum schedule with j executed in $[n-1,n)$. Suppose a minimum cost schedule executes jobs $k_1,\ldots k_n$, in that order. If $j \notin \{k_1,\ldots,k_n\}$, then the schedule $k_1,\ldots,k_{n-1},j$ has the desired property (since $c(j) \le c(k_n)$). Otherwise if $j = k_i$, the schedule $k_1,\ldots,k_{i-1},k_{i+1},\ldots,k_{n-1},j$ has the desired property.

Now it is easy to prove by induction on n that C finds a minimum cost schedule. To verify the time bound for C, implement A and D_t, $t = 1,\ldots,n$, as mergeable heaps, e.g., 2-3 trees [AHU,pp.152-55]. Then the operations of union and removing the smallest are $O(\log m)$. The time bound follows. □

It now follows that procedure A runs in time $O(m + n\alpha(n,n) \log n)$ on simple scheduling matroids. The analysis is similar to that for spanning trees. Note that contraction is easy to do on a simple scheduling matroid: Suppose a set containing k_i jobs of deadline i, $i = 1,\ldots,n$, is to be contracted. Then a job whose original deadline is d gets a new deadline $d - \sum_{i=1}^{d} k_i$.

Now we give an algorithm for the intersection problem on simple scheduling matroids that runs in $O(m + n \log n)$ time. It works by computing a restricted swap sequence from the definition. We start by characterizing the valid swaps for a base B. For time $t = 1,\ldots,n$, let b_t be the number of jobs in B with deadline at most t. The slack in B at time t is $t-b_t$; B is tight at time t if $t = b_t$. (Thus B is a base if and only if it has non-negative slack at times $1,\ldots,n-1$ and is tight at n.)

Lemma 5.3. Let B be a base, with jobs $g \in B$, $h \notin B$. Choose t as small as possible so that $t \geq d_h$ and B is tight at t. Then B-g+h is a base if and only if $d_g \leq t$.

Proof. Let $B' = B-g+h$. B' is a base if and only if for $s = 1, \ldots, n$, $b'_s \leq s$. We consider three cases.

If $d_g > t$, then $b'_t = b_t + 1 = t + 1$, so B' is not a base. If $d_g \leq d_h$, then for any s, $s = 1, \ldots, n$, $b'_s \leq b_s$, so B is a base. Finally suppose $d_h < d_g \leq t$. If $s < d_h$ or $s \geq d_g$, then $b'_s = b_s$. If $d_h \leq s < d_g$, then $b'_s = b_s+1$; the choice of t shows $b'_s \leq s$. Hence B' is a base. □

The Lemma implies that the following procedure can be used to find the best swap (g,h) for a base B and red job h.

procedure D;

begin

1. find the smallest time $t \geq d_h$ where B is tight;
2. find the largest green job $g \in B$ with $d_g \leq t$;
3. make (g,h) a swap; $B: = B - g + h$;

end D;

Note that time t exists, since B is a base. The given job g exists if there is any swap for h, and (g,h) is a largest swap, by the Lemma.

We will find a restricted swap sequence by iterating D. We use the following data structure for the base B: Take any balanced tree with n leaves, e.g., a complete binary tree. Number the leaves from left to right as $1, \ldots, n$. The leaves descending from a node s form an interval of integers $[\ell, r]$. Node s represents the corresponding time interval, $[\ell-1, r)$. Node s has three data fields: G(s) is the largest green job in B with a deadline in $[\ell,r]$. $S(s)$ is an integer used to compute slacks; more precisely for any integer time t, if P is the path from leaf t to the root, then the slack of B at t is $\sum_{u\in P} S(u)$. $M(s)$ is the smallest sum $\sum_{u\in P} S(u)$, where P is a path from s to a leaf in $[\ell,r]$. (Thus for instance the root has M-value 0.)

Figure 5.2 illustrates this data structure for Figure 5.1. Each node s is labelled with the values $G(s)$, $S(s)$, $M(s)$. Figure 5.2(a) shows the data structure for the initial base of all green jobs; Figure 5.2(b) shows it after the first swap (7,1) has been made.

Besides the balanced tree, the green jobs of B are organized in lists: For each integer time t, $1 \le t \le n$, there is a list $L(t)$ of all the green jobs in B with deadline t; $L(t)$ is sorted so that cost is nonincreasing.

Using this data structure, it is easy to implement D in time $O(\log n)$ [AHU,pp.145-152]. For example, the update to B in line 3 is done as follows:

```
     comment set B to B - g + h;
3.1  remove g from L(d_g);
     if L(d_g)≠∅ then G(d_g):= the first job in L(d_g)
                 else G(d_g):=0 comment 0 is a dummy job with cost - ∞;
3.2  increase S(d_g) and M(d_g) by 1;
     for each node s on the path from leaf d  to the root do
         begin
         if s has a right brother r then increase S(r) and M(r) by 1;
         if s ≠ d_g then begin
                let G(s) be the job with the largest cost in {G(s')|s'is
                  son of s};
                M(s):= S(s) + min{M(s')|s' is a son of s};
         end   end;
```

3.3 decrease $S(d_h)$ and $M(d_h)$ by 1;

for each node s on the path from leaf d_h to the root **do**

begin

if s has a right brother r **then** decrease $s(r)$ and $M(r)$ by 1;

$M(s) := S(s) + \min\{M(s') | s' \text{ is a son of } s\}$;

end;

The remaining details of procedure D are left to the reader. Now suppose we are given bases B and R of all green and all red jobs respectively. A restricted swap sequence for B and R can be found by sorting the jobs in B and in R, constructing the data structure for B, and iterating procedure D n times. This gives the following result.

Lemma 5.4. A restricted swap sequence for two bases can be found in time $O(n \log n)$ and space $O(n)$. □

The complete algorithm for our intersection problem follows procedure A: First find bases B_ℓ and B_u (lines 6-7). Then check q for feasibility (line 8). Find a restricted swap sequence as in line 9, only using the procedure given above. As in lines 10-11, form the desired base B by making the q smallest swaps. Finally construct a schedule for B in linear time, as follows: Schedule jobs of B from the first time slot to the last, always choosing the job with smallest deadline to be executed next.

Theorem 5.1. For simple scheduling matroids, a smallest schedule with q red jobs can be found in time $O(m+n \log n)$ and space $O(m)$. □

Note that our time bound is the same as the best known method for finding a minimum cost base. (The time needed to do the latter is clearly a lower bound on the time for our intersection problem.) We can also show that our algorithm is the best possible implementation of the swap sequence approach. This is done below in Corollary 6.1.

5.2. General Scheduling Matroids.

Now we treat the general scheduling problem, where each job j has a release time r_j and a deadline d_j. Again we seek a smallest schedule with q red jobs.

We start by normalizing the matroid.

Lemma 5.5. Let M be a general scheduling matroid. Then there is a general scheduling matroid M', where M' has rank $n' = \max\{d_j | j \text{ is a job of } M'\}$, $m' < (n')^3$, and a base of M' is a base of M. If the jobs have costs, a minimum cost base of M' is minimum for M. M' can be found in time $O(m)$.

Proof. First we find M' satisfying all conditions except the upper bound on m'. Initialize M' to M. Without loss of generality, for $n' = \max\{d_j | j \text{ is a job of } M'\}$, $n' > m'$; also some job has release time 0. Now do the following:

1. let $\ell = \min\{t | t \text{ is an integer and } t > |\{j | \text{job } j \text{ has } r_j < t\}|\}$;
2. let $h = \min\{t | t \text{ is an integer, } t \geq \ell-1 \text{ and some job } j \text{ has } r_j = t\} \cup \{n'\}$;
3. contract the time interval $[\ell-1,h)$, i.e., set all deadlines that are in $[\ell-1,h)$ to $\ell-1$, and decrease all release times and deadlines that are h or more by $h-\ell+1$;

Note that in line 1 ℓ is well-defined since the time n' is in the set. In line 2, $h \geq \ell$, since line 1 implies no job has release time $\ell - 1$. Thus line 3 contracts a nonempty time interval.

Now we show that any schedule can be transformed so that it does not use the time slots that are contracted in line 3. Given a schedule S for M, let S' be a schedule executing the same jobs as S but as early as possible; more precisely, if $[t_i-1,t_i)$, i=1, ..., k are the times when jobs are executed and $t_1 < t_2 \ldots < t_k$, then the vector $(t_1,t_2,\ldots,t_k)$ is lexicographically minimum. Let u be the highest time, $u \leq \ell$, where $[u-1,u)$ is idle in S'. Note u exists by line 1. We claim $u = \ell$.

To see this, suppose on the contrary that $u < \ell$. The ℓ-u jobs that S' schedules in $[u,\ell)$ have release times at least u (else S' could be improved). Line 1 implies there are at least u jobs with release times less than u. This gives at least $(\ell-u) + u = \ell$ jobs with release times less than ℓ, a contradiction.

Hence $u = \ell$. This implies that no job with release time less than ℓ is scheduled after time ℓ (otherwise S' can be improved). Now it is easy to see from line 2 that no job is scheduled in $[\ell-1,h)$. Thus the contraction in line 3 does not change the bases of the matroid.

Now to insure that $n' \leq m'$, repeat the above three steps until the desired condition holds. (Note that each repetition decreases the value n'). However in line 1, always consider only new jobs, i.e., if k is the previous value ℓ-1, the new value of ℓ is $\ell = \min\{t | t \text{ is an integer and } t-k > |\{\text{job } j \text{ has } k \leq r_j < t\}|\}$.

The above procedure can be implemented in time O(m), if the jobs are sorted by release time. (In line 3, it is only necessary to modify a job's release time and deadline once). Further, we can keep track of the modifications, so that a schedule in M' can be converted to a schedule in M.

Finally we modify M' to achieve the upper bound on m'. For every nonempty set of the form $\{j|r \le r_j < d_j \le d\}$, delete all but the d-r smallest jobs from M'. Using bucket sorts and linear-time selection, this step is O(m). □

In the remainder of the discussion we assume M has been modified as in the Lemma.

The greedy algorithm on scheduling matroids can be implemented in time $O(m \log m + n^2)$. Lipski and Preparata [LP] obtain a bound of $O(m\,n)$ using matchings and augmenting paths. Their method can be modified to achieve the above time bound. Here we give a method based on Glover's algorithm for matching convex graphs [Gl], also called the earliest deadline rule for scheduling [LF,J].

The earliest deadline rule finds a schedule for a given set of jobs if one exists [LF,J]. Call a job j <u>available</u> for time t if it is not scheduled in [0,t-1) and $r_j < t \le d_j$. The <u>earliest deadline rule</u> is as follows:

```
for t: = 1 to n do
  if some job is available for t then
    in [t-1,t), schedule a job that is available for t and
      has smallest possible deadline;
```

We call any schedule that can be constructed by this rule an <u>earliest deadline schedule</u>. By convention, during any time interval [t-1,t) that the machine is idle we say it is executing a dummy job 0, where $d_0 = n + 1$.

The greedy algorithm works by iterating the following step: Given an unscheduled job x and scheduled jobs S, if S + x can be scheduled then add x to S. (The jobs x are considered in order of nondecreasing cost).

We maintain an earliest deadline schedule for S. To gain efficiency, we delete certain "tight" time intervals, where the schedule cannot change. A time interval $[r,d)$ is <u>tight</u> if $d-r = |\{j|j$ is currently scheduled and $r \le r_j \le d_j \le d\}|$. The following data structure keeps track of time intervals $[t-1,t)$ that are not in tight intervals and hence not deleted.

The time intervals $[t-1,t)$ that are not in tight intervals, where t is an integer, $0 \le t \le n+1$, are maintained in a doubly-linked list T in ascending order. A node p on T has three fields. TIME(p) is the value t for interval $[t-1,t)$, SUCC(p) is a pointer to the next node on T, if it exists, and PRED(p) is a pointer to the preceding node, if it exists. Thus TIME(p) < TIME(SUCC(p)) if SUCC(p) exists. (Note that the first and last nodes of T, with TIME fields 0 and n + 1, are dummies).

In addition, the times 0, ..., n are partitioned into disjoint sets. For each time t on T, there is a set $S_t = \{s|0 \le s \le n$, and t is the lowest time in T with $s<t\}$. These sets are manipulated by UNION and FIND instructions [AHU]. UNION(S_s,S_t) merges set S_s into S_t, thereby eliminating S_s. FIND(s) has a value that is a pointer to a node of T: if $s \in S_t$, then TIME(FIND(s)) = t.

The algorithm also uses a list C of changes to the current schedule. The algorithm attempts to construct an earliest deadline schedule for S + x. All changes needed in the current schedule are recorded in C. If S + x can be scheduled, then the changes of C are made; otherwise they are not made.

```
    procedure E;  comment given is an earliest deadline schedule for a
                  set of jobs, and an unscheduled job x.  E adds x to the
                  earliest deadline schedule, if this is possible.  Otherwise
                  it may delete a tight interval from T;
    begin
1.  C: = Ø; u: = x;p: = FIND(r_x); t: = r: = TIME(p);
2.  while u ≠ 0 and t ≤ d_u do
       begin
3.     let j be the job scheduled in [t-1,t) comment j may be 0;
4.     if d_u < d_j then
5.        begin in list C, schedule u in [t-1,t); u: = j end;
6.     p: =  SUCC(p); t: = TIME(p);
       end;
7.  if u = 0 then update the schedule by making the changes in C
    else begin
8.     q: = PRED(p); s: = TIME(q); r: = r-1,
9.     while s > r do
         begin
10.      let j be the job scheduled in [s-1,s);
11.      if r_j < r then r: = r_j;
12.      UNION(s,t);
13.      q: = PRED(q); s: = TIME(q);
   end end end E;
```

Lemma 5.6. The greedy algorithm, using procedure E, finds a minimum cost schedule in time $O(m \log m + n^2)$ and space $O(m)$. (The time is $O(m\,\alpha(m,n) + n^2)$ if the jobs are given in sorted order).

Proof. Assume that x is the next job to be considered by the greedy algorithm and that an earliest deadline schedule has been constructed for the correct subset of jobs larger than x. Further, assume that list T and sets S_t are correct. This means that if s and t are consecutive times on T, then integers $[s,t)$ form a set S_t and $[s,t-1)$ is a tight interval. We show that E processes x correctly, i.e., an earliest deadline schedule for S + x is constructed if possible; otherwise T and S_t are updated correctly. Clearly this suffices to show that the greedy algorithm itself works correctly.

We begin by analyzing the loop of lines 1-6. Let C be the schedule derived from the current one by making the changes in list C. We show by induction that each time line 2 is reached, one of the following alternatives holds:

(i) $u = 0$, in which case C is an earliest deadline schedule for S + x.

(ii) $t > d_u$, in which case it is impossible to schedule all jobs of S + x.

(iii) $u \neq 0$ and $t \leq d_u$, in which case C is an earliest deadline schedule for S + x over the time interval $[0,t-1)$; the jobs remaining to be scheduled in $[t-1,n)$ are u plus the jobs in $[t-1,n)$ in the current schedule.

For the base case, u = x. On the time interval $[0,r_x)$ it is clear that C is an earliest deadline schedule for S + x. Thus if $t = r_x + 1$, then alternative (iii) holds at line 2. Otherwise suppose $[r_x,r_x+1)$ is in a tight interval $[s,t-1)$ (for some s). If $t > d_x$, then x and all jobs in $[s,t-1)$ have deadline $\leq t-1$. Thus the earliest deadline rule does not schedule one of these jobs, and (ii) holds. Otherwise if $t \leq d_x$, then since all jobs currently in $[s,t-1)$ have deadline less than d_x, (iii) holds.

For the inductive step, assume (iii) holds when line 2 is reached. We show that one of (i)-(iii) holds after lines 3-6 are executed. It is clear that lines 3-5 update C and u correctly for the interval $[0,t)$. Now if $u = 0$, (i) holds. Otherwise, reasoning as above, (ii) holds if $t > d_u$ and (iii) holds if $t \le d_u$. This completes the induction.

It is easy to see that E is correct if the loop halts with $u = 0$. Otherwise, if $u \ne 0$, from (ii) we need only show that lines 8-13 update the schedule correctly.

Let r_0 be the value of r computed in line 8. Consider the jobs of the original schedule in $[r_0,t-1)$. These jobs all have deadline $\le t-1$. For let the tight interval ending at $t-1$, if any, be $[s,t-1)$ (for some s). Jobs in $[s,t-1)$ have deadline $\le t-1$. Lines 4-5 show that jobs in the remainder of $[r_0,t-1)$ have deadline $\le d_u < t$.

Take $\mathcal{s} \le t-1$ as high as possible such that a job with deadline $> t-1$ (or job 0) is scheduled in $[\mathcal{s}-1,\mathcal{s})$. (If no such $\mathcal{s}$ exists, take $\mathcal{s} = 0$.) The earliest deadline rule implies all jobs in $[\mathcal{s},t-1)$ have release times $\ge \mathcal{s}$. Hence $[\mathcal{s},t-1)$ is tight. The preceding paragraph shows $\mathcal{s} \le r_0$. Now take $\mathcal{r}$ as large as possible such that

$$\mathcal{r} = \min\{r_0\} \cup \{r_j \mid \text{job } j \text{ is scheduled in } [\mathcal{r},t-1)\}.$$

Clearly $\mathcal{r} \ge \mathcal{s}$ and $[\mathcal{r},t-1)$ is tight. Further, if $[\mathcal{r}',t')$ is tight, with $\mathcal{r}' < \mathcal{r} \le t' \le t-1$, then $[\mathcal{r}',t-1)$ is tight.

It is an easy matter to check that the loop of lines 9-13 halts with $r = \mathcal{r}$, and further that it updates T and the sets S_t for the tight interval $[\mathcal{r}',t-1)$. Thus the greedy algorithm works correctly.

Now we estimate the time for the greedy algorithm. The initial sort of m jobs by cost is $O(m \log m)$. The set merging instructions do m FINDs (line 1) and $\le n$ UNIONs (line 12); thus the time for set merging is $O(m\alpha(m,n))$. We show that the remainder of the processing is $O(n^2)$.

Exactly n jobs get scheduled. For each of these jobs, lines 1-7 are executed in $O(n)$ time. Thus $O(n^2)$ time is spent on the scheduled jobs. Now we show that $O(m)$ time is spent on the jobs x that do not get scheduled. For these jobs, it is clear from previous remarks that each of the times t processed in lines 2-6 it also processed (as s) in lines 9-13. We charge an iteration of lines 9-13 to the time processed, s. Since there is at most one iteration for a given s, the charge to a given s is $O(1)$. The desired bound follows. □

The techniques used in E can be applied to find a restricted swap sequence. Consider a given earliest deadline schedule and an unscheduled red job h. To find a largest swap (g,h) for h, select d as small as possible and r as large as possible so

$$d = \max\{d_h\} \cup \{d_j \mid j \text{ is scheduled in } [r_h,d)\},$$
$$r = \min\{r_h\} \cup \{r_j \mid j \text{ is scheduled in } [r,d)\}.$$

We claim that g is a largest green job scheduled in $[r,d)$.

To see this, first note that $[r,d)$ is tight, as in Lemma 5.6. Thus any valid swap has its green job in $[r,d)$. Next note that swap (g,h) is valid. For if g is in $[r_h,d)$, an earliest deadline schedule with g swapped for h is constructed by a procedure similar to E (as usual assume we start with an earliest deadline schedule):

1. replace g by the 0 job;
2. s: = r_h; u: = h:
3. *while* u ≠ 0 *do*
   begin
4. let j be the first job scheduled in $[s,d)$ with $d_j > d_u$; let j be scheduled in $[t-1,t)$;
5. in $[t-1,t)$ schedule u; u: = j; s: = t;
   end;

Otherwise if g is in $[r,r_h)$, the following algorithm constructs an earliest deadline schedule with g swapped for h:

1. let g be scheduled in $[s-1,s)$;
2. **while** $s \le r_h$ **do**
   begin
3. let j be the first job scheduled in $[s,d)$ with $r_j \le s-1$; let j be scheduled in $[t-1,t)$;
4. in $[s-1,s)$ schedule j; $s := t$;
   end;
5. add h to the schedule, using lines 2-5 of the previous algorithm;

Thus to compute a restricted swap sequence for red jobs $h_1, \ldots, h_n$ and base H_0, iterate the following steps: For h_i, compute r and d as above; find the largest green job g_i in $[r,d)$; find an earliest deadline schedule for H_i, as above. Since one iteration is $O(n)$, the time for the restricted swap sequence is $O(n^2)$.

Now we sketch the complete algorithm for our intersection problem. We modify procedure A. First find a smallest base B_ℓ with the minimum number of red jobs (line 6 of A). To do this use the greedy algorithm, with all green jobs considered before any red job. Similarly compute B_u (line 7). The test of line 8 is as in A. The restricted swap sequence is found as above. Then, as in lines 10-11, the desired base B is found. Finally, the desired schedule is formed. This last step can be done using the earliest deadline rule in time $O(n \log\log n)$ [EKZ] or more efficiently in time $O(n\alpha(n,n))$[LP].

<u>Theorem 5.2</u>. For general scheduling matroids, a smallest schedule with q red jobs can be found in time $O(m \log n + n^2)$ and space $O(m)$. □

Although we cannot prove a matching lower bound for this problem, note that the time is the same as for the greedy algorithm.

5.3. Matching and Transversal Matroids.

We close Section 5 by briefly examining the general case of matching and transversal matroids. First we need some notation. Recall that n denotes the number of vertices in a base; m denotes the number of vertices in the set J of vertices in the matroid. We need one more parameter to specify the size of the matroid: e denotes the number of edges in the graph.

The greedy algorithm on matching matroids can be implemented in time $O(m \log m + n e)$: The time to sort the vertices is $O(m \log m)$. To test if a vertex can be added to an independent set, we use the method of augmenting paths. Using the techniques of [Ga2,KM] for cardinality matching, the test can be done in time $O(e)$. As in the general scheduling algorithm, we delete vertices that are reached in unsuccessful tests (these are "Hungarian vertices" [Ga2]). This makes the unsuccessful tests use a total of $O(m+e)$ time, while the successful ones use $O(n e)$. The time bound follows.

Again using augmenting paths, a restricted swap sequence can be computed from the definition, in time $O(n e)$. This completes a sketch of the following result:

Theorem 5.3. For matching and transversal matroids, a smallest base with q red elements can be found in time $O(m \log m + n e)$ and space $O(m+e)$. $\square$

As usual our time bound matches the best known bound for finding a minimum cost base.

6. Partition Matroids.

This section discusses our intersection problem on partition matroids. First we prove an $\Omega(n \log n)$ lower bound for the swap sequence approach. This bound extends to scheduling and matching matroids. Then we give an $O(m)$ algorithm for partition matroids.

We start by defining the matroids. A partition matroid derives from a set E of m elements that is partitioned into disjoint subsets E_i, $i=1, \ldots, b$, and positive integers n_i, $i=1, \ldots, b$. The elements of the matroid are those of E; a base is a set containing exactly n_i elements of E_i, $i=1, \ldots, b$. [L1, p. 272] If $b = 1$, the matroid is called uniform.

The greedy algorithm on partition matroids finds a minimum cost base in time $O(m \log m)$. This is easily improved to $O(m)$ by using a linear-time selection algorithm.

A restricted swap sequence can be constructed from the definition in time $O(n \log n)$. Thus following procedure A our intersection problem can be solved in time $O(m+n \log n)$. Before improving this, we show that it is the best one can do using the swap sequence approach. Recall from Section 4 that sort (n) is the time needed for an algorithm to sort n numbers. Now consider an algorithm on partition matroids. Say it "uses the swap sequence approach" if, given a uniform matroid consisting of two bases, one green and the other red, the algorithm finds the swaps in a swap sequence for the two bases. (As usual, the algorithm need not find the order of the swaps.)

Lemma 6.1. Any algorithm on partition matroids that uses the swap sequence approach requires time $\Omega(\text{sort}(n))$.

Proof. As in Lemma 4.1, it suffices to give a procedure that sorts n numbers by calling the algorithm and doing $O(n)$ extra processing.

Consider n arbitrary numbers $x_1, \ldots, x_n$. The corresponding uniform matroid consists of n green elements costing 1, ..., n and n red elements costing $x_1, \ldots, x_n$. If the numbers in nondecreasing order are $y_1, \ldots, y_n$, it is easy to see that the swap sequence is $(n,y_1), (n-1,y_2), \ldots, (1,y_n)$.

The procedure forms the matroid and calls the algorithm. Then it computes the array S, where S(i) is the element paired with i as a swap. Last, it outputs S(n), S(n-1), ..., S(1) as the sorted order.

Only O(n) time before and after the algorithm is used. □

Note that a uniform matroid is a special case of a simple scheduling matroid (where all jobs have deadline n). Thus for the obvious definition of "swap sequence approach" the following corollary holds:

Corollary 6.1. Any algorithm on simple scheduling matroids that uses the swap sequence approach requires time $\Omega(\text{sort}(n))$. The same holds for general scheduling, transversal and matching matroids.

Now we present the linear algorithm. It is a straightforward adaptation of an efficient algorithm for selecting a q^{th} smallest element of a multiset X + Y. Here X and Y are sets of n real numbers and X + Y is the multiset $\{x+y \mid x \in X, y \in Y\}$. We use the selection algorithm of Jefferson, Shamos and Tarjan [S,pp.256-8], which runs in time O(n log n). The well-known algorithm of [JM] also uses time O(n log n) but apparently does not adapt to our problem.

In a partition matroid it is easy to describe a swap sequence for B and R: Choose some consistent rule to break ties in cost. Then the swaps in a swap sequence are $\bigcup_{E_i} \{(e_j,f_j) \mid 1 \le j \le n_i$, e_j is the j^{th} largest green element of $E_i \cap B$ and f_j is the j^{th} smallest red element of $E_i - B$, if both exist$\}$. A swap sequence is formed by sorting this set in order of nondecreasing cost.

To get a linear algorithm we cannot find the swaps individually. Instead we repeatedly select a small number of swaps, all known to be either among the q smallest swaps or not among them. In the first case the swaps are executed; in the second case the elements of the swaps are deleted. More precisely, if $q \le \lfloor \frac{n}{2} \rfloor$, we find swaps that are larger than the median (i.e., the $\lfloor \frac{n}{2} \rfloor^{th}$ smallest swap); these swaps are deleted. If $q > \lfloor \frac{n}{2} \rfloor$ we find swaps that are smaller than the median and execute them.

Now we sketch the complete algorithm. It follows lines 6-9 of procedure A. First we find B_ℓ, a smallest base with as few red elements as possible. Note that for each block E_i, $B_\ell \cap E_i$ consists of the n_i smallest green elements, or if only $h < n_i$ green elements exist, all green elements plus the n_i-h smallest red elements. We can find B_ℓ in O(m) time using linear-time selection. The desired base B is initialized to B_ℓ.

Similarly we find B_u in O(m) time and check q for feasibility. Then we pass to the matroid $M - \overline{(B_\ell \cup B_u)}/(B_\ell \cap B_u)$: Elements not in $B_\ell \cup B_u$ are deleted. An element $e \in B_\ell \cap B_u$ is contracted as follows. e is deleted from the matroid; n and n_i, where $e \in E_i$, are decreased by 1; if e is red, q is decreased by 1. After the matroid is modified, there are 2n elements, forming bases of all green and all red elements.

The last step is finding the q smallest swaps (in the reduced matroid), and executing them on B. This gives the desired base. The procedure for this, outlined above, is now stated in precise form. For conciseness, we treat the two cases $q \le \lfloor \frac{n}{2} \rfloor$ and $q > \lfloor \frac{n}{2} \rfloor$ together: The changes for the second case are always given in parentheses, as in $q \le (>) \lfloor \frac{n}{2} \rfloor$. In the algorithm below, recall that for a block E_i, the j^{th} smallest swap consists of the j^{th} largest green element of $E_i \cap B_\ell$ and the j^{th} smallest red element of $E_i - B_\ell$.

procedure F;
begin
while $q > 0$ **do**
  begin
  let $q \le (>) \frac{n}{2}$;
  for each nonempty block E_i **do**
    begin
    $k_i := \left\lceil \frac{3}{4} n_i \right\rceil$; $h_i := \left\lfloor \frac{n}{4} \right\rfloor + 1$;
    let (e_i, f_i) be the $k_i{}^{th}$ smallest (largest) swap for E_i;
    end;
  let (e_p, f_p) be the smallest (largest) swap such that for
    $I = \{i \mid c(e_i, f_i) \le (\ge)\ c(e_p, f_p)\}, \sum_{i \in I} k_i > \left\lfloor \frac{n}{2} \right\rfloor$;
  for each nonempty block E_i **do**
    if $c(e_i, f_i) \ge (\le)\ c(e_p, f_p)$ **then**
      begin
      let $\{(e_j, f_j) \mid j \in J\}$ be the h_i largest (smallest) swaps for E_i;
      for $j \in J$ **do** delete e_j and f_j from E_i:
      if $q > \left\lfloor \frac{n}{2} \right\rfloor$ **then**
        begin
        for $j \in J$ **do** $B := B - e_j + f_j$;
        $q := q - h_i$;
        end;
      $n_i := n_i - h_i$; $n := n - h_i$;
end **end** **end** F;

To insure the algorithm works correctly we must break ties consistently. Assume all elements have been indexed. If two swaps have equal cost, we say the one whose red element has smaller index is smaller.

Theorem 6.1. For partition matroids, a smallest base with q red elements can be found in $O(m)$ time and space.

Proof. From the remarks above, it suffices to show that procedure F is correct and uses $O(n)$ time and space.

We start with correctness. First note that in line 6, the swap (e_p, f_p) exists, since $\sum_{i=1}^{b} k_i \geq \left\lceil \frac{3}{4} n \right\rceil > \left\lfloor \frac{n}{2} \right\rfloor$.

Now suppose $q \leq \left\lfloor \frac{n}{2} \right\rfloor$. The number of swaps with cost $\leq c(e_p, f_p)$ is $> \left\lfloor \frac{n}{2} \right\rfloor$, by line 6. Thus any swap with cost $\geq c(e_p, f_p)$ is not among the q smallest, and can be deleted. Note that the h_ith largest swap is the $n_i + 1 - h_i = \left\lceil \frac{3}{4} n_i \right\rceil$th smallest. Thus all swaps deleted in lines 7-14 have cost $\geq c(e_i, f_i) \geq c(e_p, f_p)$ as desired.

Next suppose $q > \left\lfloor \frac{n}{2} \right\rfloor$. Reasoning as above, the number of swaps with cost $\geq c(e_p, f_p)$ is $> \left\lfloor \frac{n}{2} \right\rfloor$. Thus any swap with cost $\leq c(e_p, f_p)$ can be executed, and lines 7-14 execute the correct swaps.

Hence each iteration of F deletes or executes the correct swaps. Since n always decreases, eventually the loop halts with B as the desired base.

Now we discuss the timing. We start by observing that one iteration of lines 2-14 is $O(n)$ (where n is its current value, i.e., the number of elements currently in a base): First note that we maintain a list of nonempty blocks E_i for use in lines 3, 6 and 7. Let b be the number of blocks in this list. Clearly $b \leq n$. Line 5 uses a linear-time selection algorithm [BFPRT,SPP]. This makes lines 3-5 $O(b+n) = O(n)$. Line 6 is $O(b) = O(n)$ if a linear-time weighted median algorithm [JM] is used. Similarly lines 7-14 are $O(n)$.

Next we show that each iteration of lines 2-14 reduces n by at least $\frac{n}{16}$. In a given iteration, let D index the blocks where deletions are made, i.e., $D = \{i \mid \text{in line 8, } c(e_i,f_i) \geq (\leq)\ c(e_p,f_p)\}$. Then n decreases by $\sum_{i\in D} h_i \geq \sum_{i\in D} \frac{n_i}{4}$. Note that $\sum_{i\notin D} k_i \leq \left\lfloor \frac{n}{2} \right\rfloor$ by line 6. Thus

$$\sum_{i\in D} n_i \geq \sum_{i\in D} k_i = \sum_{i=1}^{p} k_i - \sum_{i\notin D} k_i \geq \frac{3}{4} n - \frac{n}{2} = \frac{n}{4}.$$

These inequalities imply n decreases by at least $\frac{n}{16}$ as desired.

Finally let n_0 be the value of n in the initial iteration of lines 2-14. In the j^{th} iteration $n \leq (\frac{15}{16})^j n_0$. Since $\sum_{j=0}^{\infty} (\frac{15}{16})^j n_o = O(n_o)$, we see that the total time is $O(n_0)$ as desired. □

7. Root-Constrained Directed Spanning Trees.

This section gives an algorithm that finds a smallest directed spanning tree rooted at a given vertex v, such that v has a given degree q. The time is $O(\min(m \log n, n^2))$, the same as for the unconstrained problem. First we review the unconstrained minimum directed spanning tree algorithm. Then we present our algorithm. Finally we show a lower bound on the efficiency of any algorithm using our approach.

Let G be a directed graph with n vertices and m edges. An edge $e = (x,y)$, directed from x to y, has head y and tail x; we also use the notation $h(e) = y$, $t(e) = x$. As usual when we contract a set of edges C to form G/C, we designate vertices and edges by their names in G. If e is an edge of both G and G/C, we write $h(e,G)$ or $h(e,G/C)$ when it is necessary to specify the graph that e is considered to be in.

All paths and cycles we consider are directed. A directed spanning tree rooted at v is a set of edges such that v is the head of no edge, and every other vertex w is the head of exactly one edge and is on a path from v to w. (This is also called an outtree; our methods easily adapt to intrees.) We sometimes abbreviate "directed spanning tree rooted at v" to "spanning tree". Since we are only concerned with trees rooted at v, we can assume that the given graph G has no edges directed to v. This implies that v is not in any directed cycle.

Now we review Karp's derivation of the minimum directed spanning tree algorithm [K]. We specialize the derivation from the general case of branchings to the case of directed spanning trees rooted at v. An edge is called critical if no edge with the same head has smaller cost. A critical cycle (spanning tree) is a cycle (spanning tree) with all edges critical. If a critical spanning tree exists it is a minimum spanning tree. However, the critical edges do not necessarily give a spanning tree: a subgraph composed of one critical edge directed to each vertex except v may contain cycles.

Let C be a critical cycle. The reduced graph for C is the graph G/C, with the cost function of G modified as follows: If g is an edge of G with $t(g) \notin C$ and $h(g) \in C$ and g' is the edge of C with $h(g') = h(g)$, then in the reduced graph g costs $c(g) - c(g')$. All other edges of G/C have their cost unchanged.

The following result, proved in [K], is central.

Lemma 7.1. Let C be a critical cycle. In the reduced graph for C, let T be a minimum spanning tree rooted at v. Let e be the edge of T with $h(e,G/C) = C$.* Let e' be the edge of C with $h(e',G) = h(e,G)$. Then in G, $T + C - e'$ is a minimum spanning tree rooted at v. □

The result implies an efficient algorithm for a minimum directed spanning tree, due to Edmonds [E]. The algorithm works in two phases. Phase I finds critical edges and places them in a set F. In the process it finds and reduces critical cycles C. Phase I halts with the edges of the desired tree contained in F and the reduced cycles C. Phase II extracts the desired tree from F and the C's, using Lemma 7.1. It halts with F as the desired tree.

* e exists since by assumption G has no edges directed to v, whence $v \notin C$.

```
Phase I:
F: = Ø;
while some vertex w ≠ v in the current graph is not the head of
    an edge in F do
    begin
    let e be a smallest edge of the current graph with h(e) = w;
    if t(e) does not descend from h(c) in F then add e to F
    else begin
        let C be the (critical) cycle formed by e and the path from
            h(e) to t(e) in F;
        reduce C;
        F: = F - C;
  end  end;
Phase II:
while the graph is a reduced graph G/C for some cycle C do
    begin
    let e be the edge of F with h(e,G/C) = C;
    let e' be the edge of C with h(e',G) = h(e,G);
    F: = F + C - e';
    end;
```

This algorithm, with the correct choice of data structures, is $O(\min(m \log n, n^2))$ [CFM,T].

Now we consider the root-constrained directed spanning tree problem. Given is a directed graph G. Each edge has a real-valued cost and a color; the green edges are directed from v, and the remaining edges are red (red edges may be directed from v). We seek a smallest spanning tree rooted at v with exactly q green edges.

Unlike the case of undirected spanning trees, the directed spanning trees do not form a matroid. Instead they are the intersection of a graphic and a partition matroid. Because of this, the Augmentation Theorem fails for directed spanning trees. This is illustrated by the directed graph in Figure 7.1. Figure 7.2 gives the optimal trees T_2, T_3 and T_4 (with 2, 3, and 4 red edges, respectively), all of which are unique. These trees do not differ by a simple swap of edges.

To overcome this difficulty we broaden the notion of swap so that several swaps may be used to derive one tree from another. More precisely, if T is a directed spanning tree rooted at v, (e,f) is a <u>swap</u> for T if edge $e \in T$ is green, $f \notin T$ is red, and $h(e) = h(f)$. As usual the <u>cost</u> of (e,f) is $c(f) - c(e)$.

(e,f) is a <u>complete swap</u> if $T - e + f$ is a spanning tree. This holds if and only if t(f) does not descend from h(f) in T. Otherwise, if $T - e + f$ is not a spanning tree, (e,f) is an <u>incomplete swap</u>.

Now we give our algorithm for the root-constrained directed spanning tree problem. Let the given cost function be c. Let $\mu = 1 + \min\{|c(e) - c(f)| \mid$ e is green, f is red, $h(e) = h(f)\}$. Define a new cost function c' as follows:

$$c'(e) = c(e) + \mu, \text{ if } e \text{ is green,}$$
$$= c(e), \text{ if } e \text{ is red.}$$

Using cost c', execute Phase I of Edmonds' algorithm. This finds a critical tree F in a reduced graph; F contains all the green edges, by choice of c'.

Next execute procedure G, given below, to make swaps until F contains q green edges. Finally, execute Phase II to derive the tree corresponding to F in the original graph.

```
procedure G;
    begin
    while F has more than q green edges do
        if no swap exists then halt comment the desired tree does
            not exist;
        else begin
            let (e,f) be a smallest swap;
            if t(f) does not descend from h(f) in F then F: = F - e + f
            else begin
                let C be the cycle formed by f and the path from
                    h(f) to t(f) in F;
                reduce C;
                F: = F - C;
    end end end G;
```

Suppose G is called on the graph of Figure 7.1 with the cost function shown and q = 2. Figure 7.3 illustrates the swaps done by G. An incomplete swap is illustrated by showing the reduced graph; a complete swap is illustrated by showing the new tree F (in the reduced graph).

Now we show that the algorithm is correct. The desired tree remains the same if the given cost function c is changed by adding a constant to all green edges. Call such a function d. We will exhibit a d such that the following is true:

(1) Every time the algorithm adds a red edge to F, it is critical (for d).

(2) Every reduction is made for a critical cycle (for d).

(3) Every green edge in F at the end of G is critical (for d) in the final graph.

Using Lemma 7.1, it is easy to show that these conditions imply the algorithm finds a minimum directed spanning tree rooted at v for the cost function d. Also it is clear that the algorithm finds a tree with q green edges. Together these properties show the algorithm is correct.

Now consider the first step of the algorithm, Phase I. d, c, and c' are identical on the red edges. Thus the first step satisfies (1) - (3). Hence it remains only to show (1) - (3) for procedure G.

We begin by introducing some notation. If b is any cost function, call an edge b-critical if it is critical when b is the cost function; similarly for a tree or cycle. Let p be the number of iterations of the loop of lines 1-8 of G. Let the swaps done by G be (e_i,f_i), $i = 1, \ldots, p$. For $i = 0, 1, \ldots, p$ let G_i be the graph at the end of the i^{th} iteration; let c_i be the cost function at the end of the i^{th} iteration; if swap (e_i,f_i) is incomplete, let C_i be the critical cycle that is reduced. Hence c_i is the cost function in G_i: (e_i,f_i) is a smallest swap for c_{i-1}; if C_i exists then $G_i = G_{i-1}/C_i$. (Although cycles C_i are indexed from 1 to p, C_i exists only for incomplete swaps (e_i,f_i)).

We start by showing the red edges of F are critical with respect to other red edges.

Lemma 7.2. In graph G_i, let f and g be red edges, with $f \in F$, $g \notin F$, and $h(f) = h(g)$. Then $c_i(g) \geq c_i(f)$.

Proof. The proof is by induction on i. For $i = 0$ the Lemma holds because Phase I satisfies (1) - (2) (and d = c on red edges).

For the inductive step, suppose the Lemma holds for G_{i-1} and consider swap (e_i,f_i). If this swap is complete, line 4 adds f_i to F. The Lemma holds for f_i, since (e_i,f_i) is a smallest swap. It continues to hold for other red edges $f \in F$. Hence it holds for all red edges f.

Next suppose (e_i,f_i) is incomplete. In G_i, if f is a red edge of F then $h(f,G_i) \neq C_i$, since $h(e_i,G_i) = C_i$. By induction the Lemma holds for f. □

In Figure 7.3 the cost of swaps is increasing. This illustrates the following result.

Lemma 7.3. The cost of swaps is nondecreasing, i.e., for $i=1,\dots,p-1$,

$$c_{i+1}(e_{i+1},f_{i+1}) \geq c_{i-1}(e_i,f_i).$$

Proof. First suppose (e_i,f_i) is a complete swap. Line 4 does not create new swaps, nor does it change the cost of existing swaps. The desired inequality follows, since (e_i,f_i) has smallest cost.

Next suppose (e_i,f_i) is incomplete. If (e_{i+1},f_{i+1}) is valid before (e_i,f_i) is executed, its cost is not changed by lines 5-7 (even if $h(f_{i+1},G_{i-1} \in C_i)$. Again the desired inequality follows. Thus suppose (e_{i+1},f_{i+1}) is not valid before (e_i,f_i). It is easy to see that both $h(e_{i+1},G_{i-1})$, $h(f_{i+1},G_{i-1}) \in C_i$. Since only one green edge of F has its head in C_i, $e_{i+1} = e_i$. So when C_i is reduced $e'_{i+1} = f_i$. Also $c_{i-1}(f'_{i+1}) \leq c_{i-1}(f_{i+1})$ by Lemma 7.2. Thus $c_i(f_{i+1}) = c_{i-1}(f_{i+1}) - c_{i-1}(f'_{i+1}) \geq 0$, and $c_i(e_{i+1}) = c_{i-1}(e_i) - c_{i-1}(f_i)$. So $c_i(e_{i+1},f_{i+1}) \geq c_{i-1}(f_i) - c_{i-1}(e_i)$, which is the desired inequality. □

Lemma 7.3 is the analog of Corollary 3.3 for matroids. As with the latter result, Lemma 7.3 implies that the inequality version of our problem (i.e., finding a smallest spanning tree where the root has degree at least or at most q) can be solved as efficiently as the equality version.

Now we derive the analog of Lemma 7.2 for green edges.

Lemma 7.4. In graph G_i, let e and g be green edges with $e \in F$, $g \notin F$, and $h(e) = h(g)$. Then $c_i(g) \geq c_i(e)$.

Proof. In G_i, if $h(e) = h(g)$ and $c_i(g) \geq c_i(e)$, then $c_j(g) \geq c_j(e)$ for all $j \geq i$. This is true because any reduction after the i^{th} iteration decreases the cost of g and e by the same amount. Now it is easy to see that the Lemma follows from this special case: $g = e_k$ in a complete swap (e_k,f_k); C_i, $i > k$, is the first cycle to be reduced that contains $h(g,G_k)$; and $e = e_i$ in the incomplete swap (e_i,f_i).

Lemma 7.3 shows $c_{i-1}(e_i,f_i) \geq c_{k-1}(e_k,f_k)$. Thus $c_i(e) = c_{i-1}(e_i) - c_{i-1}(f_i) = - c_{i-1}(e_i,f_i) \leq - c_{k-1}(e_k,f_k) = c_{k-1}(g) - c_{k-1}(f_k)$. Note that $c_{k-1}(g) = c_{i-1}(g)$ and $c_{k-1}(f_k) = c_{i-1}(f_k)$, by the hypotheses of the special case. Hence $c_i(e) \geq c_{i-1}(g) - c_{i-1}(f_k) = c_i(g)$ as desired. □

Now we show that the algorithm satisfies properties (1)-(3) with respect to the cost function d defined as follows:

$$\begin{aligned} d(g) &= c(g) + c_{p-1}(e_p,f_p), &&\text{if } g \text{ is green,} \\ &= c(g) &&\text{, if } g \text{ is red.} \end{aligned}$$

Define similar functions d_i, $i = 0, \ldots, p$ as follows:

$$\begin{aligned} d_i(g) &= c_i(g) + c_{p-1}(e_p,f_p), &&\text{if } g \text{ is green,} \\ &= c(g) &&\text{, if } g \text{ is red.} \end{aligned}$$

Suppose the original cost function c is changed to d. It is easy to see that d_0 is the cost function at the end of Phase I, and d_i, $i=1,\ldots,p$ is the cost function at the end of the i^{th} iteration. This follows by induction. (Observe that in a reduction if $c(g)$ changes to $c(g) - c(g')$, then g' is in the cycle, hence red, so $c(g') = d(g')$.)

Also define R_i, $i=0,\ldots,p$ as the forest of red edges in G_i that are in F. Now it is easy to see that Lemma 7.5 below is a statement of properties (1) - (2), and Lemma 7.6 is property (3).

Lemma 7.5. For $i=0,\ldots,p$, R_i is d_i-critical in G_i. Further for $i=1$, $\ldots,p$, if swap (e_i,f_i) is incomplete and contracts cycle C_i, then C_i is d_{i-1}-critical.

Proof. We use induction on i. For the base case, note that F is initially c_0-critical and contains all the green edges. Since c_0 and d_0 are identical on red edges, R_0 is d_0-critical in G_0.

Now suppose $1 \leq i \leq p$. The i^{th} iteration does swap (e_i,f_i). We start by observing that $R_{i-1} + f_i$ is d_{i-1}-critical. R_{i-1} is

d_{i-1}-critical by induction. Consider edge f_i. If f is a red edge with $h(f) = h(f_i)$, then $c_{i-1}(f) \geq c_{i-1}(f_i)$ by the choice of (e_i,f_i). Thus $d_{i-1}(f) \geq d_{i-1}(f_i)$ as desired. Next suppose e is a green edge with $h(e) = h(f_i)$. Lemma 7.4 shows $c_{i-1}(e) \geq c_{i-1}(e_i)$. Lemma 7.3 shows $c_{p-1}(e_p,f_p) \geq c_{i-1}(e_i,f_i)$. Thus $d_{i-1}(e) = c_{i-1}(e) + c_{p-1}(e_p,f_p) \geq c_{i-1}(e_i) + c_{i-1}(e_i,f_i) = c_{i-1}(f_i) = d_{i-1}(f_i)$ as desired.

Now suppose (e_i,f_i) is a complete swap. Then $R_i = R_{i-1} + f_i$, and $d_i = d_{i-1}$ (since $c_i = c_{i-1}$). Thus the above paragraph shows R_i is d_i-critical, as desired.

Next suppose (e_i,f_i) is incomplete. The contracted cycle C_i is contained in $R_{i-1} + f_i$ and thus is d_{i-1}-critical. It remains to show that R_i is d_i-critical. An edge $f \in R_i$ is in R_{i-1} and does not have its head in C_i (since $h(e_i,G_{i-1}) \in C_i$). Hence no costs are modified at vertex $h(f,G_{i-1})$ and f remains critical in G_i. □

Lemma 7.6. At the end of procedure G, every green edge of F is d_p-critical in G_p.

Proof. Consider a green edge $e \in F$ in G_p. If g is a green edge with $h(g,G_p) = h(e,G_p)$, then $c_p(g) \geq c_p(e)$ by Lemma 7.4, whence $d_p(g) \geq d_p(e)$. If f is a red edge with $h(f,G_p) = h(e,G_p)$, then (e,f) is a swap in $G_p = G_{p-1}$. Since it costs no less than (e_p,f_p), $c_p(e,f) \geq c_p(e_p,f_p)$. Adding $c_p(e)$ to both sides gives $d_p(f) \geq d_p(e)$. □

Since properties (1) - (3) are established, the algorithm is correct. (It is easy to see that if procedure G halts in line 2, there is no spanning tree rooted at v with q green edges.)

It is not difficult to implement the algorithm in time $O(\min(m \log n, n^2))$ using the data structures of [T,CFM]. This involves using one UNION-FIND structure to represent vertices of the reduced graph, another to represent trees in the forest F, and priority queues to keep track of the costs of all edges with a given head [T]. The cycles are represented by a tree structure [CFM]. In addition, procedure G uses a priority queue of swaps (for line 3).

Theorem 7.1. A smallest root-constrained directed spanning tree can be found in time $O(\min(m \log n, n^2))$ and space $O(m+n)$.

Proof. Correctness follows from Lemmas 7.1-6 and the accompanying discussion. Further details concerning the time bound are left to the reader. □

We conclude this section with a lower bound for our approach. Say that an algorithm on a directed graph "uses the swap sequence approach" if, given a graph as shown in Figure 7.4 (a), it determines the complete swaps found by procedure G (with $q = 1$). Note that the algorithm need not determine the incomplete swaps; as usual the algorithm need not determine the order of the complete swaps. Also, although swaps may be for reduced graphs, they are specified by the edges of the original graph.

Lemma 7.7. Any algorithm on a directed graph that uses the swap sequence approach requires $\Omega(\text{sort }(n))$ time.

Proof. As in Lemma 4.1 we show that with only $O(n)$ extra processing, the algorithm, called on an $n+2$ vertex graph, can sort n numbers.

Consider n arbitrary numbers $x_i. \ldots, x_n$. Without loss of generality assume that the numbers are distinct and positive. (To achieve the former condition break ties by using the indices of the numbers.) These numbers correspond to the directed graph shown in Figure 7.4 (a). More precisely, the graph contains vertices v, w, and u_i, $i = 1, \ldots, n$. For each i, $i = 1, \ldots, n$, there is a green edge (v,u_i) costing $-x_i$, and red edges (w,u_i)

and (u_i,w) both costing x_i; there is also a green edge (v,w) costing 0.

Let the given numbers in increasing order by $y_i, \ldots, y_n$. Now we show that the swaps found by G are

(4) $((v,w), (u_1,w)), ((v,u_1), (w,u_1)), ((v,u_1),(u_2,w)), \ldots, ((v,u_i), (w,u_i)),$

$((v,u_i), (u_{i+1}, w)), \ldots ,((v,u_{n-1}), (w,u_{n-1})), ((v,u_{n-1}), (u_n, w)).$

To do this we prove inductively that for $i = 1, \ldots, n$ the first $2i-1$ swaps are as in (4) and give the graph shown in Figure 7.4 (b). There vertices $u_1, \ldots, u_{i-1}$ and w are contracted into a new vertex, which we call w; the edges directed to this vertex have their cost modified so that (u_j,w) costs $x_j - x_{i-1}$, for $j \geq i$; the spanning tree consists of edges (v,u_j), $j \geq i$ and (u_i,w).

For the base case $i = 1$, note that in the original graph the smallest swap at vertex w is $((v,w), (u_1,w))$ costing x_1, and the smallest swap at u_i is $((v,u_i), (w,u_i))$ costing $2x_i$. Hence the first swap is as in (4). Since this is a complete swap no contraction or cost modification occurs. This gives Figure 7.4 (b) for $i = 1$, where we take $x_0 = 0$.

Now assume the first $2i-1$ swaps are as in the induction hypothesis. We do the inductive step by analyzing the next two swaps. For the $2i^{th}$ swap, the smallest swap at u_j $(j \geq i)$ is $((v,u_j), (w,u_j))$ costing $2x_j$, and there are no swaps at w. Hence the $2i^{th}$ swap is $((v,u_i), (w,u_i))$ as in (4). This incomplete swap contracts vertex u_i into w, changing the cost of (u_j,w) to $x_j - x_i$, $j > i$; the new green edge (v,w) gets cost $-2x_i$.

For the $2i + 1$st swap, the smallest swap at w is $((v,w),$ $(u_{i+1},w))$ costing $x_{i+1} + x_i$. (Note that (v,w) derives from the original edge (v,u_i), so this swap is also designated as $((v,u_i), (u_{i+1},w)))$. The smallest swap at u_j $(j > i)$ is still $((v,u_j), (w,u_j))$ costing $2x_j$. Hence the 2 $i+1^{st}$ swap is $((v,u_i), (u_{i+1},w))$, as in (4). Since this is a complete swap it is easy to see that the rest of the inductive assertion holds. This completes the induction and establishes (4).

Now we describe the procedure. Given numbers $x_1, \ldots, x_n$ construct the graph of Figure 7.4 (a) and call the algorithm to find the complete swaps. (These are the odd swaps of (4)). Then set an array S so that $S(i) = j$ if $((v,u_i), (u_j,w))$ is a complete swap; further, $S(0) = j$ where $((v,w), (u_j,w))$ is a complete swap. (4) implies that the sequence $S(0)$, $S^2(0), \ldots, S^n(0)$ gives the indices of the numbers in sorted order. Since besides the algorithm only $O(n)$ time is used, this gives the desired result. □

Lemma 7.7 can be extended to incorporate a lower bound on the time to find a minimum directed spanning tree, as in Corollary 4.1.

References

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

[B] T. H. Brylawski, "Some properties of basic families of subsets," Discrete Math. 6, (1973), pp. 333-341.

[BFPRT] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time bounds for selection," J. Comput. Systems Sci. 7, (1973), pp. 448-461.

[CFM] P. M. Camerini, L. Fratta, and F. Maffioli, "A note on finding optimum branchings," Networks, to appear.

[CT] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees," SIAM J. on Computing 5, (1976), pp. 724-741.

[E] J. Edmonds, "Optimum branchings," J. Res. NBS 71B, (1967), pp. 233-240.

[EKZ] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," Math. Systems Theory 10, 1977, pp.99-127.

[F] N. Friedman, "Some results on the effect of arithmetics on comparison problems," Proc. 13th Annual Symp. on Switching and Automata Theory, College Park, Maryland, 1972, pp. 139-143.

[Ga1] H. N. Gabow, "A good algorithm for smallest spanning trees with a degree constraint," Networks, 8 (1978), pp. 201-208.

[Ga2] H. N. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matching on graphs," J.ACM 23 (1976), 221-234.

[GK] F. Glover and D. Klingman, "Finding minimum spanning trees with a fixed number of links at a node," Report No. 74-5, Research Report CS #169, Center for Cybernetic Studies, University of Texas at Austin, Austin, Texas, 1974.

[Gl] F. Glover, "Maximum matching in a convex bipartite graph," Naval Research Logistics Quarterly 14, (1967), pp. 313-316.

[GT] H. N. Gabow and R. E. Tarjan, "Efficient algorithms for simple matroid intersection problems," Proc. 20th Annual Symp. on Foundations of Comp. Sci., San Juan, Puerto Rico, 1979, pp.196-204.

[Gu] D. Gusfield, "Matroid optimization with the interleaving of two ordered sets," preprint.

[HS] E. Horowitz, and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Potomac, Maryland, 1978.

[J] J. R. Jackson, "Scheduling a production line to minimize maximum tardiness," Research Rept. 43, 1955, Management Sci. Research Project, Univ. of Calif. at Los Angeles.

[JM] D. B. Johnson and T. Mizoguchi, "Selecting the K^{th} element in X+Y and $X_1+X_2 \ldots +X_m$," SIAM J. Comput. 7, (1978), pp. 147-153.

[K] R. M. Karp, "A simple derivation of Edmonds' algorithm for optimum branchings," Networks 1, (1971), pp. 265-272.

[KM] T. Kameda and I. Munro, "A O(VE) algorithm for maximum matching of graphs," Computing 12, 1974, pp. 91-98.

[L1] E. L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, New York (1976).

[L2] E. L. Lawler, "Matroid intersection algorithms," Math. Programming 9, (1975), pp. 31-56.

[LF] T. Lang and E. B. Fernández, "Scheduling of unit-length independent tasks with execution constraints," Inf. Proc Letters 4, (1976), pp. 95-98.

[LP] W. Lipski, Jr., and F. P. Preparata, "Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems," preprint.

[P] C. Papadimitriou, "The complexity of the capacitated tree problem," Networks 8, (1978), pp. 217-230.

[S] M. I. Shamos, "Geometry and statistics: Problems at the interface," in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub ed., Academic Press, N. Y., 1976, pp. 251-280.

[SP] P. M. Spira and A. Pan, "On finding and updating spanning trees and shortest paths," SIAM J on Computing 4, (1975), pp. 375-380.

[SPP] A. Schönhage, M. Paterson, and N. Pippenger, "Finding the median," J. Comptr. Syst. Sci. 13, (1976), pp. 184-199.

[ST] D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," Proc. 13th Annual ACM Symp. on Th. of Computing, Milwaukee, Wisc., 1981, pp. 114-122.

[T1] R. E. Tarjan, "Finding optimum branchings," Networks 7, (1977), pp. 25-35.

[T2] R. E. Tarjan, "Minimum spanning trees," unpublished manuscript, 1981.

[U] R. A. Ubelmesser, "Finding smallest spanning trees with one degree constraint," M. S. Thesis, Dept. of Computer Science, University of Colo., Boulder, Colorado, 1978.

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

[Y] A. C. Yao, "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees," Information Processing Letters 4, (1975), pp. 21-23.

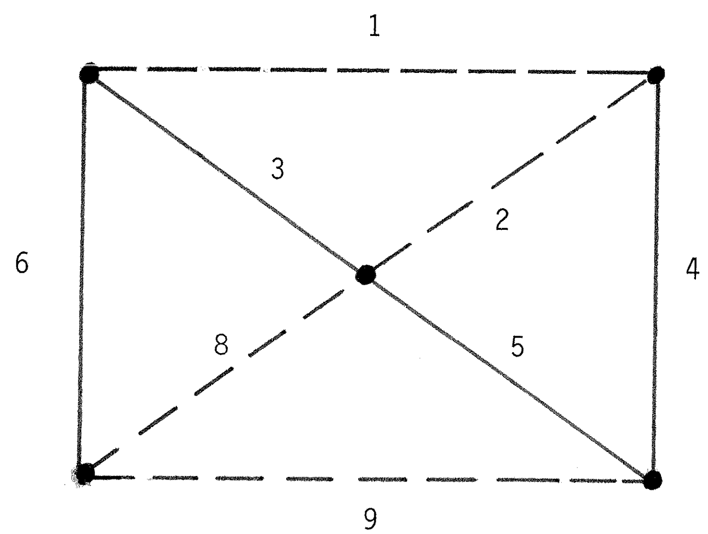


Figure 2.1.
Graphic matroid with colors and edge costs.

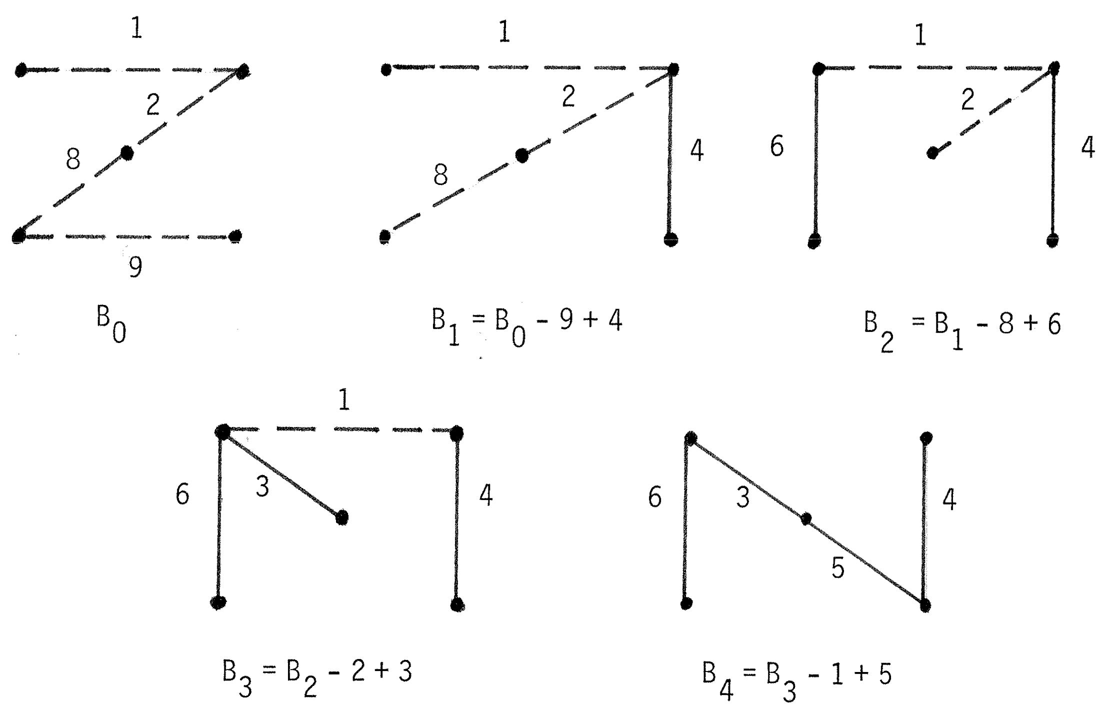


Figure 2.2.
Optimum bases and swap sequence.

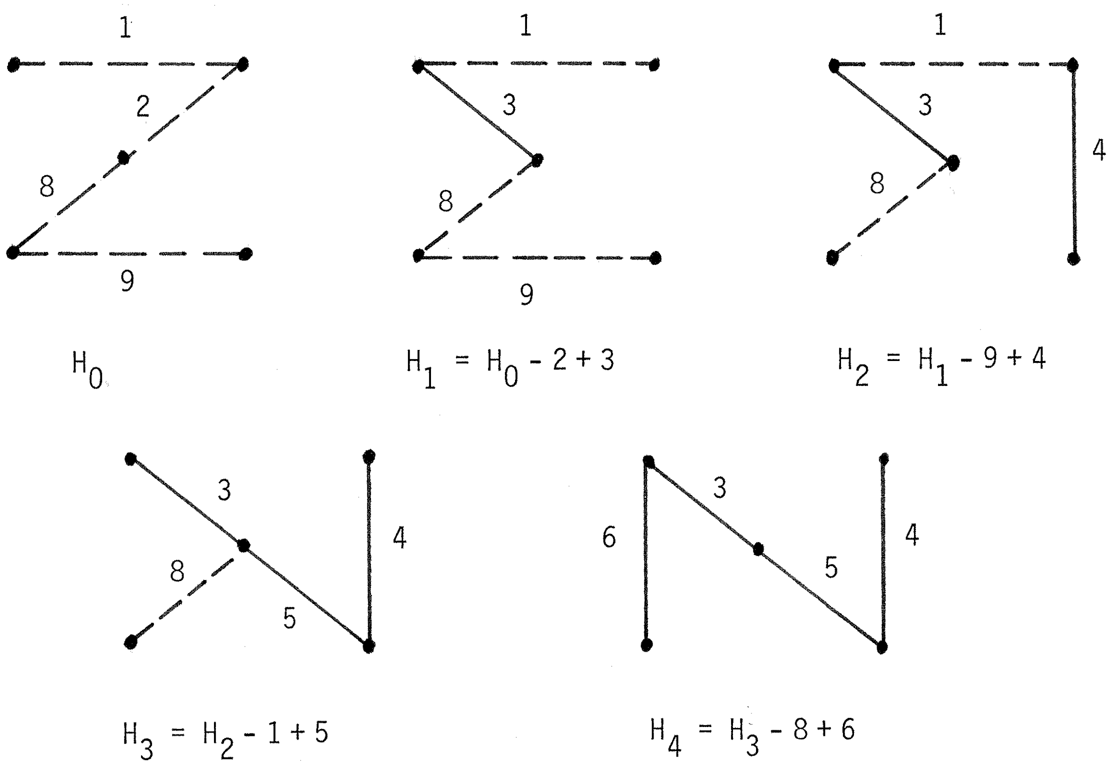
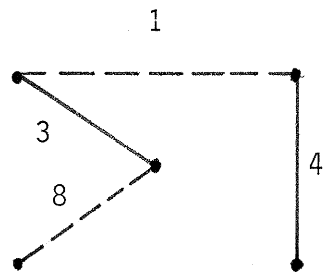
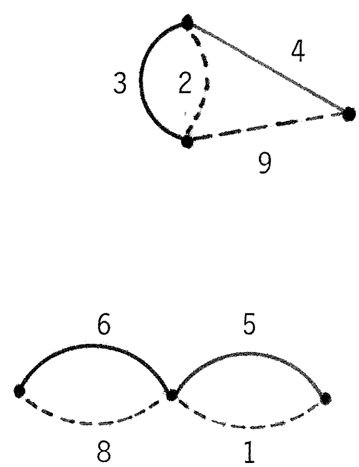


Figure 3.1
Restricted swap sequence.



(a)



(b)

Figure 3.2.

Algorithm A.

(a) $B - G_1 + R_1$

(b) The two recursive calls.

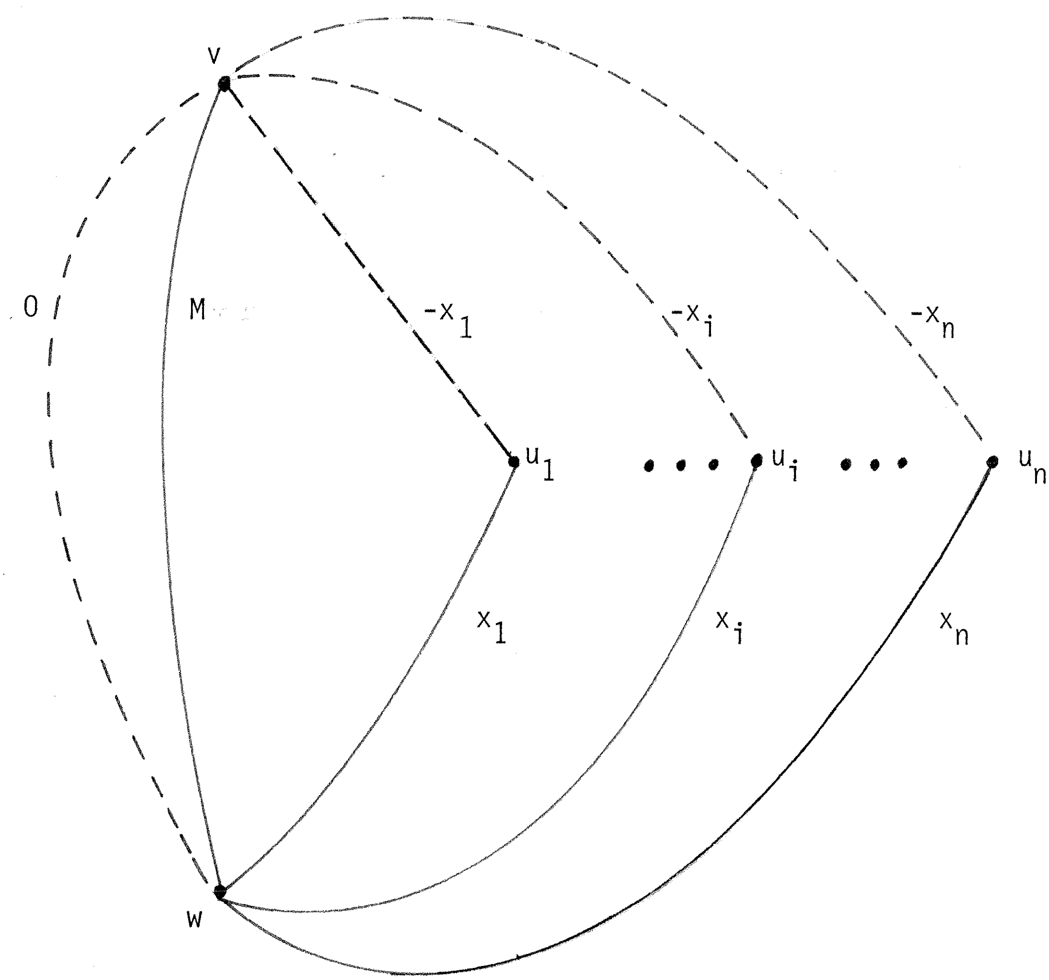


Figure 4.1. Lower bound graph.

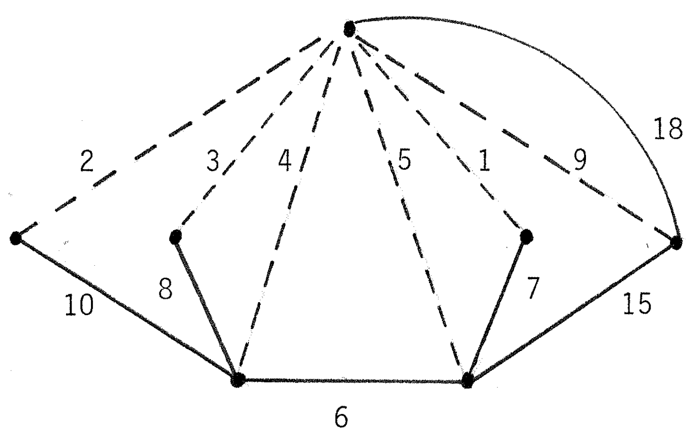


Figure 4.2.

A graph with M = {(5,6), (4,6), (3,8), (1,7), (9,15), (2,10)} and F = {6,8,2}.

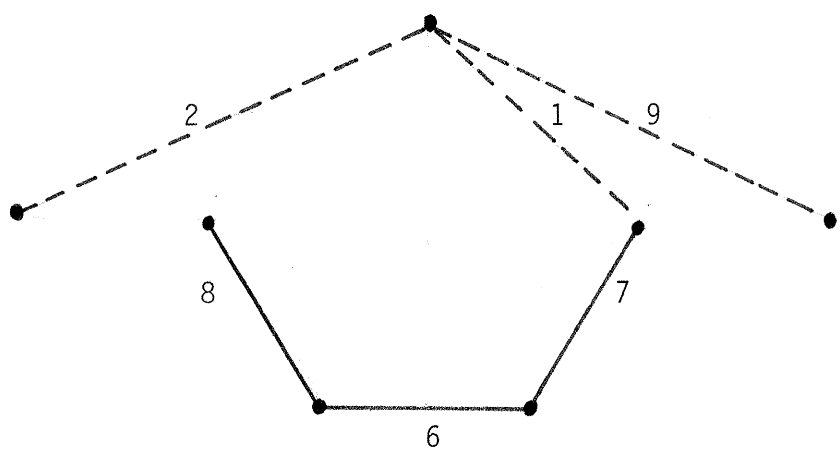
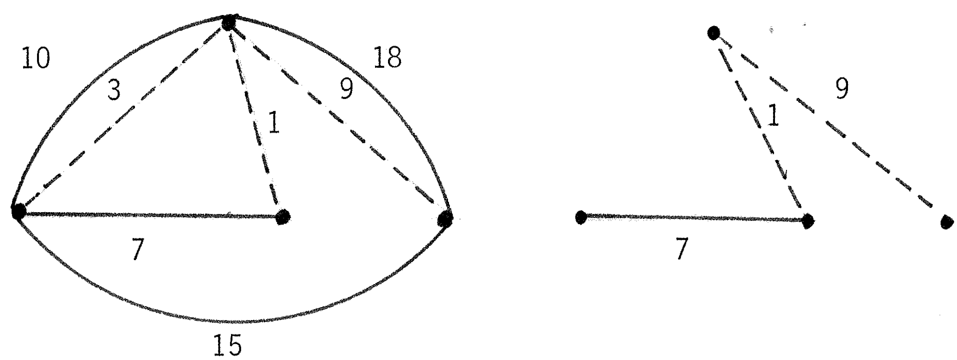


Figure 4.3.

T_3 for Figure 4.2.



(a) H

(b) $T_3 - F$

Figure 4.4.

Derived graph for Figure 4.2.

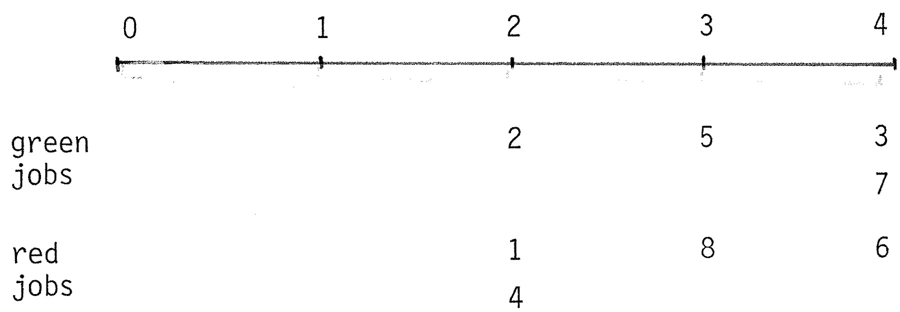


Figure 5.1.
Simple job scheduling matroid, with swap sequence (7,1), (5,6), (2,4),(3,8).

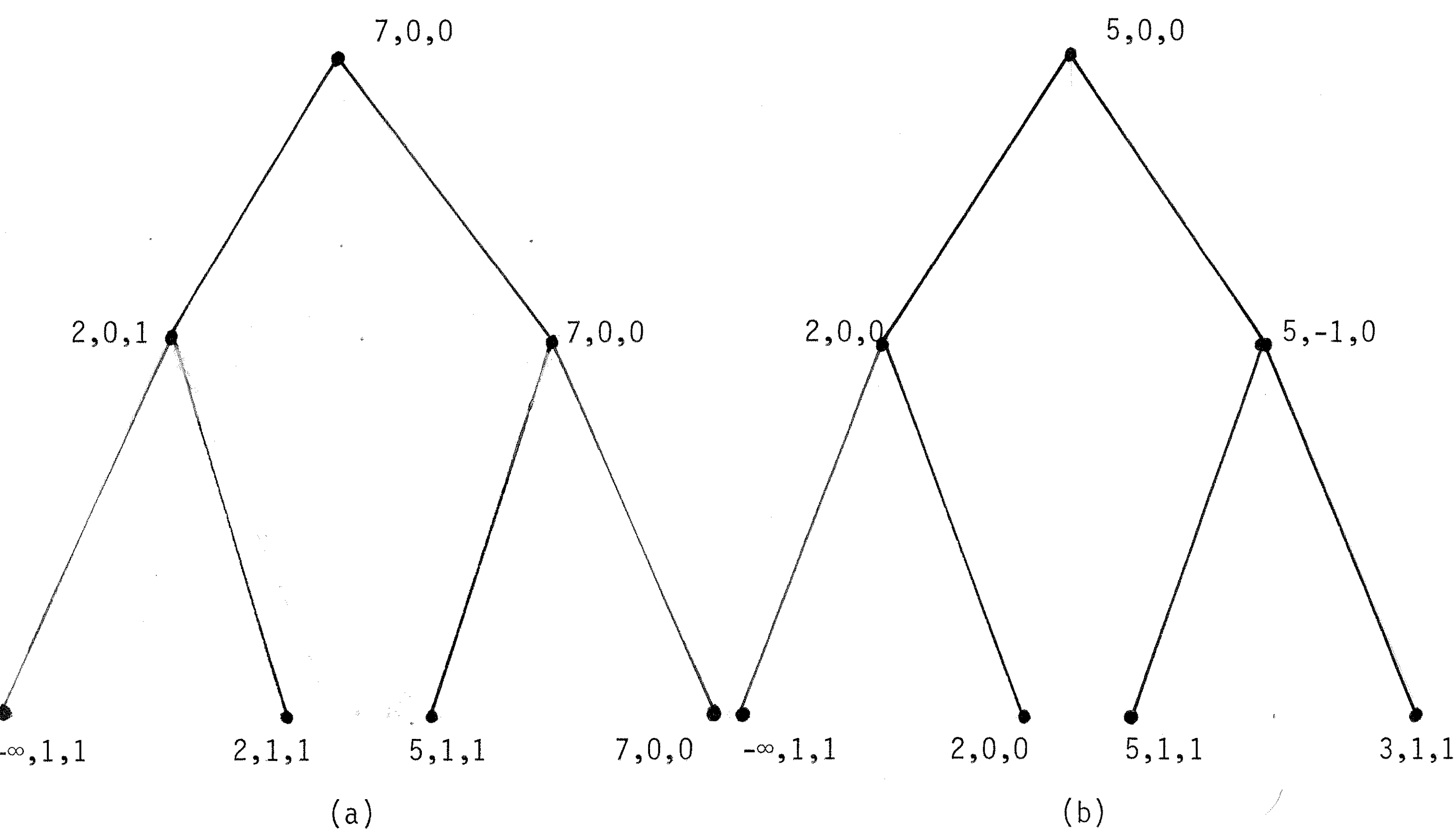


Figure 5.2.
Balanced tree for (a) B_0 (b) $B_0 - 7 + 1$.

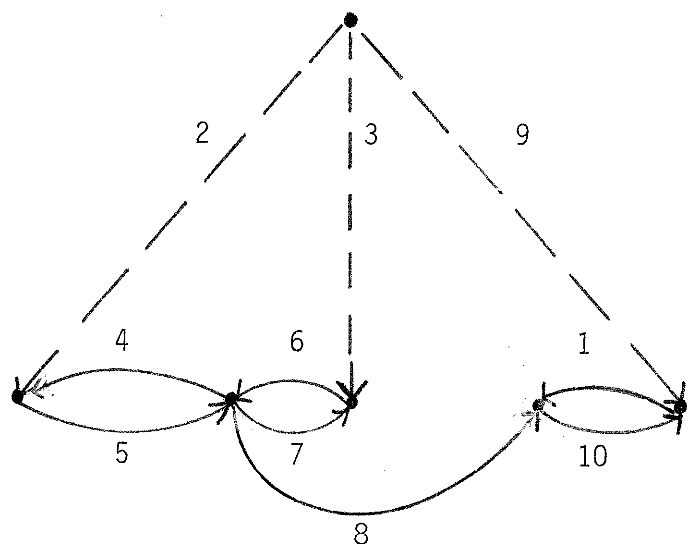


Figure 7.1
Example directed graph.

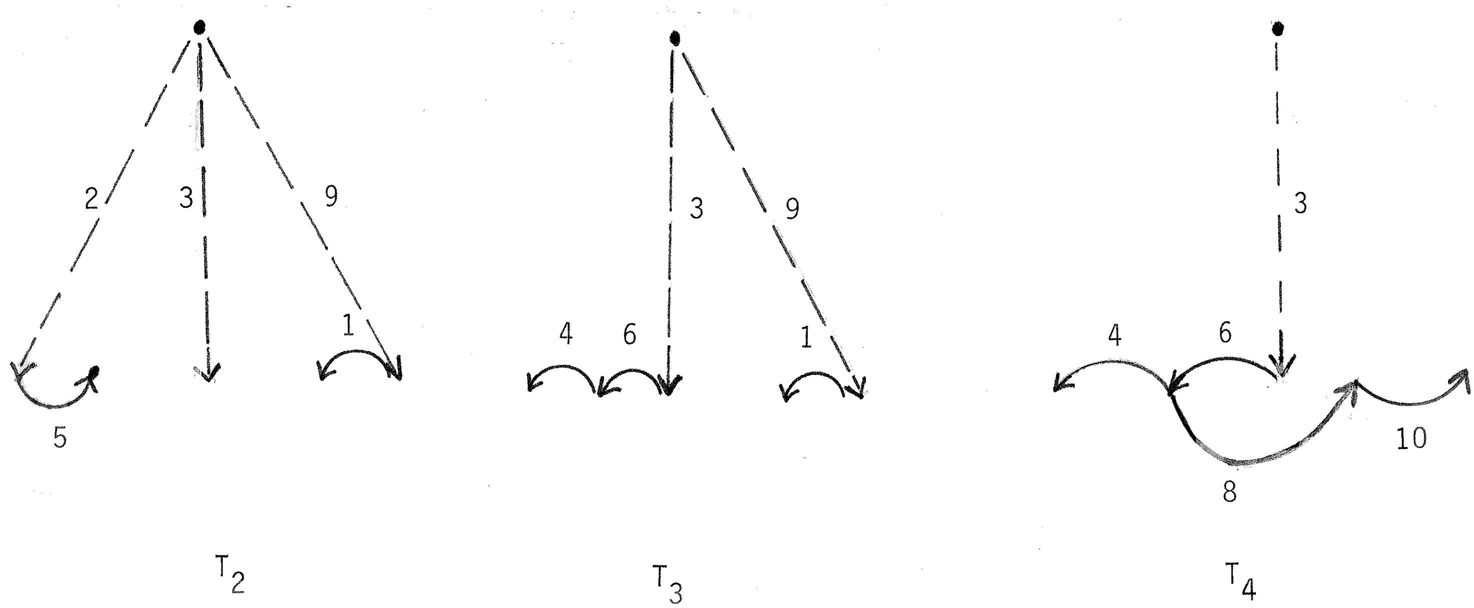
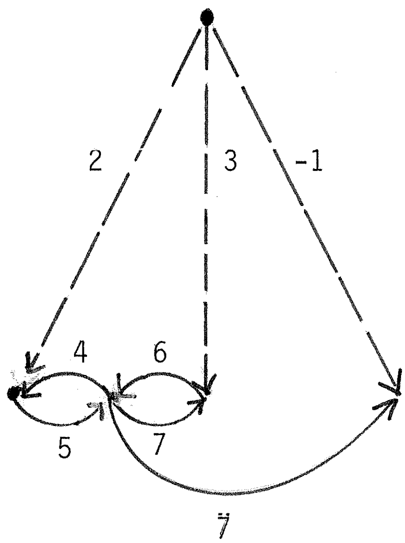
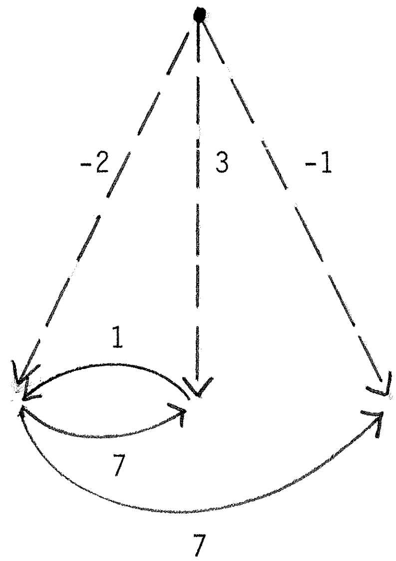


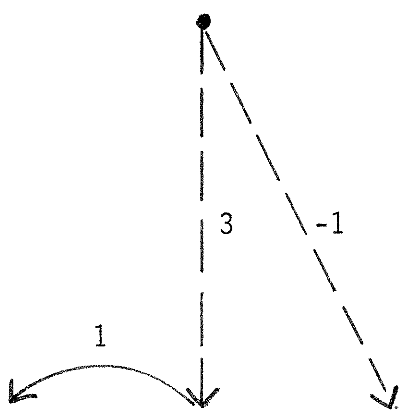
Figure 7.2
Optimal trees for Figure 7.1.



(a)
(9,10) - incomplete

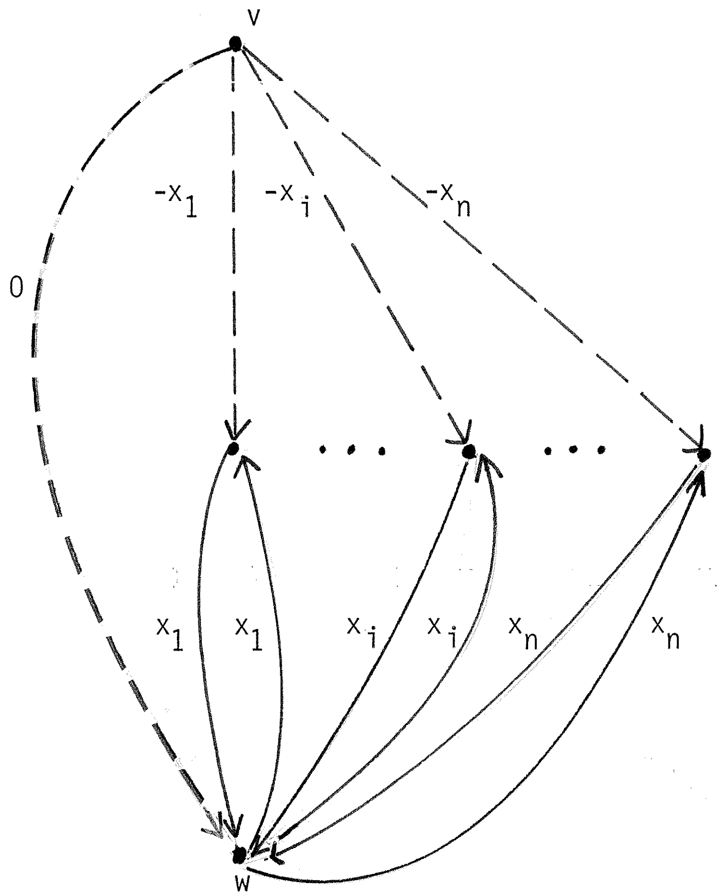


(b)
(2,4) - incomplete

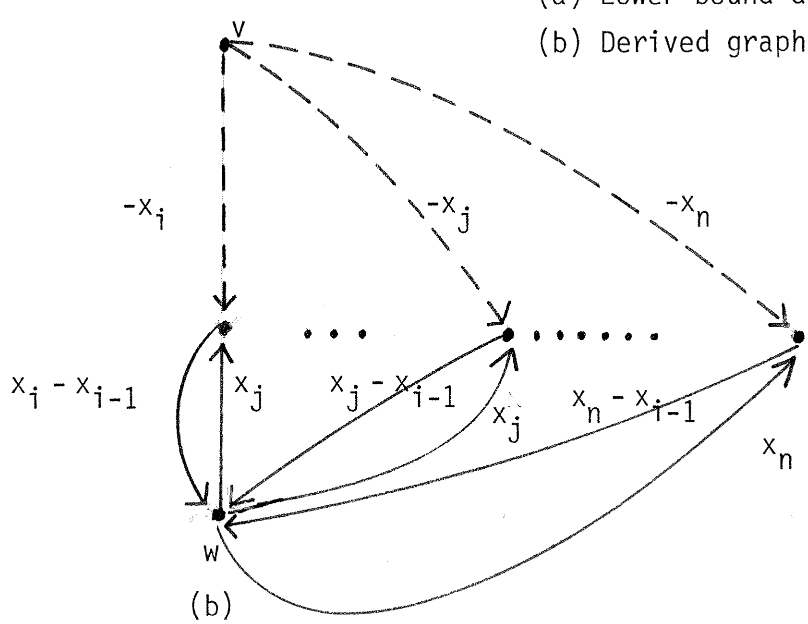


(c)
(-2,1) - complete

Figure 7.3.
Swaps done by procedure G.



(a)



(b)

Figure 7.4.
(a) Lower bound directed graph.
(b) Derived graph.