MATHEMATICS OF OPERATIONS RESEARCH
Vol. 10, No. 4, November 1985
Printed in U.S.A.

A NOTE ON FINDING MINIMUM-COST EDGE-DISJOINT SPANNING TREES*†

JAMES ROSKIND‡ AND ROBERT E. TARJAN§

Let G be an undirected graph with n vertices and m edges, such that each edge has a real-valued cost. We consider the problem of finding a set of k edge-disjoint spanning trees in G of minimum total edge cost. This problem can be solved in polynomial time by the matroid greedy algorithm. We present an implementation of this algorithm that runs in $O(m \log m + k^2n^2)$ time. If all edge costs are the same, the algorithm runs in $O(k^2n^2)$ time. The algorithm can also be extended to find the largest k such that k edge-disjoint spanning trees exist in $O(m^2)$ time. We mention several applications of the algorithm.

1. Introduction. Let G be an undirected graph with n vertices and m edges, such that each edge has a real-valued cost. We shall assume that G has no multiple edges, although our results easily extend to allow multiple edges. Let k be a fixed positive integer. We consider the problem of finding a set of k edge-disjoint spanning trees in G of minimum total cost. We call this the *minimum spanning trees* problem. For $k = 1$, this is just the classical minimum spanning tree problem [3], [21]. The edges of G form a matroid if we define a set of edges F to be independent if and only if F can be partitioned into k forests [4], [5], [16]. Thus we can use the matroid greedy algorithm [11] to solve this problem. We initialize the independent set F to be empty and repeat the following step for each edge e in G, in nondecreasing order by cost:

Augmenting step. If $F \cup \{e\}$ is independent, replace F by $F \cup \{e\}$.

After all edges are processed, F is a basis (maximal independent set) of minimum cost. In particular, if G contains k edge-disjoint spanning trees, F is the edge set of the union of k edge-disjoint spanning trees of minimum total cost.

The hard part of this algorithm is testing the independence of $F \cup \{e\}$. To do this we maintain a partition of F into forests $F_1, F_2, \ldots, F_k$, which we update each time an edge is added to F. The updating algorithm uses augmenting sequences, much like the augmenting paths of network flow and matching theory. Indeed, an alternative approach to the problem is to formulate it as a polymatroidal network flow problem [8], [12]. (See §4.)

The idea behind the updating algorithm was discovered independently by Nash-Williams [14], [15] and Tutte [20] for edge-disjoint forests and extended to the matroid setting by Edmonds [5]. Clausen and Hansen [4] present two versions of the minimum spanning trees algorithm. They give some experimental results but do not analyze the theoretical running time of their methods.

In this paper we propose an efficient implementation of the mimimum spanning trees algorithm. The algorithm runs in $O(m \log m + k^2n^2)$ time, or in $O(k^2n^2)$ time if all the edge costs are the same. (For ease in stating time bounds we assume $m \geqslant 2$.) The novelties in our result are in the details of the data structures and the implementation.

*Received July 5, 1983; revised August 10, 1984.
AMS 1980 subject classification. Primary: 05C35. Secondary: 90C35.
IAOR 1973 subject classification. Main: Graphs. Cross references: Networks.
OR/MS Index 1978 subject classification. Primary: 491 Networks/graphs/tree algorithms.
Key words. Spanning trees, graphs, networks.
†Research partially supported by Hertz Fellowship.
‡Massachusetts Institute of Technology.
§AT&T Bell Laboratories.

0364-765X/85/1004/0701$01.25

In §2 we develop an algorithm for testing independence and maintaining a partition of F into edge-disjoint forests. This algorithm can equally well be presented in the setting of matroids, but for concreteness we state it specifically for the spanning trees problem. In §3 we give an efficient implementation of the algorithm. In §4 we discuss an extension, applications, and related work by others. The first author's Ph.D. thesis [18] contains additional results and discussion.

2. Updating edge-disjoint forests. We shall generally not distinguish between a forest and its edge set. The key idea in the updating algorithm is the idea of an *augmenting sequence*. The development below is not original but is based on the references cited in §§1 and 4. If F is a forest and $e = \{v, w\}$ is an edge such that v and w are in the same tree of F, we define $F(e)$ to be the unique path in F joining v and w. If i is any integer, we define $i+ = (i \bmod k) + 1$. For a given set of edge-disjoint forests $F_1, F_2, \ldots, F_k$ and edges e_0 and e_l, a *swap sequence from e_0 to e_l* is a sequence of edges $e_0, e_1, \ldots, e_l$ such that, for $j \in [0 \ldots l-1]$,[1] $e_{j+1} \in F_{j+}(e_j)$. The swap sequence is *augmenting* if e_0 is in none of the forests $F_1, F_2, \ldots, F_k$, the endpoints of e_l are in different trees of F_{l+}, and the swap sequence is minimal in the sense that there are no two edges e_j and $e_{j'}$ such that $j' > j+1$ and $e_{j'} \in F_{j+}(e_j)$. (See Figure 1.)

Given an augmenting sequence, we can increase the size of $\bigcup_{i=1}^{k} F_i$ as follows: for $j \in [0 \ldots l-1]$ we replace F_{j+} by $F_{j+} \cup \{e_j\} - \{e_{j+1}\}$ (a *swap*), and then we replace F_{l+} by $F_{l+} \cup \{e_l\}$. We call the entire process an *augmentation*. The following lemma shows that an augmentation produces a set of forests:

LEMMA 1. *After an augmentation each of $F_1, F_2, \ldots, F_k$ is a forest.*

PROOF. Let $e_0, e_1, \ldots, e_l$ by an augmenting sequence, let $i \in [1 \ldots k]$, and let F_i and F_i', respectively, be F_i before and after the augmentation. Consider the following algorithm, which manipulates a forest F, initially equal to F_i:

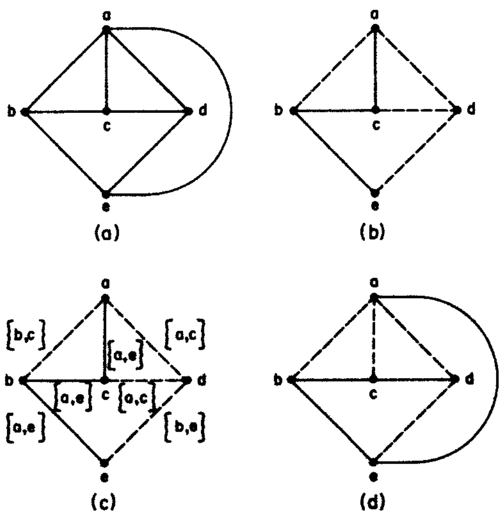

FIGURE 1. An augmenting swap sequence.
(a) Graph.
(b) Two edge-disjoint forests: the solid forest F_1 and the dashed forest F_2. Edge $\{a, e\}$ is in neither forest.
(c) Labels assigned by the labeling algorithm for finding an augmenting sequence, starting with $e_0 = \{a, e\}$. Labels denote edges with which the labeled edges can be swapped.
(d) After augmentation using swap sequence $\{a, e\}, \{a, c\}, \{c, d\}$.

[1] We denote by $[p \ldots q]$ the set of integers $p, p+1, \ldots, q$.

Step 1 (contraction). Repeat the following step for $j = i, i+k, i+2k, \ldots, i+\Delta k$, where Δ is maximum such that $i + \Delta k < l$:

Contraction step. Add e_{j-1} to F. Contract the resulting cycle to a single vertex.

Step 2 (expansion). Repeat the following step for $j = i+\Delta k, i+(\Delta-1)k, \ldots, i$:

Expansion step. Expand the cycle previously formed by adding e_{j-1} to F. Delete e_j from F.

Because an augmenting sequence is minimal, each contraction step combines two or more vertices of F. (That is, when e_{j-1} is added to F its endpoints have not yet been contracted together.) It follows that after each contraction or expansion step F is a forest whose trees span the same sets of vertices as the trees of F_i. The final value of F is F_i' if $i \neq l+$ or $F_i' - \{e_l\}$ if $i = l+$. In either case it follows that F_i' is a forest. ∎

REMARK 1. Lemma 1 is not entirely trivial. The definition of a swap sequence depends upon the initial forests, but these forests change as the swaps are carried out. Just because an edge e_{j+1} is on the cycle $F_{j+}(e_j)$, it does not immediately follow that e_{j+1} is on the cycle $F_{j+}'(e_j)$, where F_{j+}' is the result of applying previous swaps to F_{j+}. Indeed, if the swap sequence is not minimal, this need not be true. ∎

REMARK 2. The fact that we need only consider edge sequences such that each edge is in the next successive forest (i.e. $e_1 \in F_1$, $e_2 \in F_2$, etc.) is critical to the complexity of our algorithm. When searching for a way to extend a swap sequence to an augmenting sequence, we only have to consider the next forest modulo k. This saves a factor of k in the running time. (See [18] for further discussion.) ∎

The algorithm for testing independence is a labeling process that searches for an augmenting sequence. To test whether $F \cup \{e_0\}$ is independent, where $F = F_1 \cup F_2 \cup \cdots F_k$, we begin with every edge of F unlabeled, place e_0 on a queue, and repeat the following step (see Figure 1):

Labeling step. If the queue is empty, stop: $F \cup \{e_0\}$ is dependent. Otherwise, delete the first edge, say e, from the queue. Let i be the index such that $e \in F_i$; if there is no such index let $i = 0$ (the latter only occurs if $e = e_0$). Test whether the endpoints of e are in the same tree of F_{i+}. If not, stop: there is an augmenting sequence. If so, label by e every unlabeled edge on $F_{i+}(e)$ and add every such edge to the queue (in the order these edges occur on $F_{+i}(e)$).

This algorithm amounts to a breadth-first search of the edges reachable from e_0 by a swap sequence; for every labeled edge e, the sequence $e_0, e_1, \ldots, e_l$ such that $e_l = e$ and, for $j \in [1 \ldots l]$, e_{j-1} labels e_j, is a swap sequence (running sequentially through the forests) from e_0 to e with l minimum. If the algorithm stops because it has detected an augmenting sequence, we can obtain the augmenting sequence by tracing edge labels backwards from the last examined edge e and then augment F as described above. If the algorithm stops because the queue is empty, then $F \cup \{e_0\}$ is dependent, as the following lemma shows:

LEMMA 2. *Suppose the labeling algorithm stops because the queue is empty. Then* $F \cup \{e_0\}$ *is dependent.*

PROOF. Suppose the algorithm stops because the queue is empty. For $i \in [1 \ldots k]$, let S_i be the set of vertices incident to a labeled edge of F_i. We claim that $S_1 = S_2 = \cdots = S_k$, that S, the common value of $S_1, S_2, \ldots, S_k$, contains both endpoints of e_0, and that, for $i \in [1 \ldots k]$, the labeled edges of F_i form a subtree spanning S.

To verify the first two parts of the claim, let e_i be any labeled edge in F_i. Since the labeling step is applied to e_i without finding an augmenting sequence, both endpoints of e_i are in the same tree of F_{i+}, and every edge on the path $F_{i+}(e_i)$ is labeled. It follows that $S_i \subseteq S_{i+}$, and by induction that $S_1 = S_2 = \cdots = S_k$. A similar argument shows that both endpoints of e_0 are in $S_1 = S$.

To verify the third part of the claim, we shall prove that if x is an endpoint of e_0, y any vertex in S, and $i \in [1 \ldots k]$, there is a path of labeled edges in F_i connecting x and y. Suppose this is not true. Let $e \in F_{i+}$ be the earliest labeled edge such that, for some endpoints x of e_0 and y of e, there is no path of labeled edges in F_{i+} connecting x and y. Let e' be the edge labeling e, i.e. e is on $F_{i+}(e')$ and e is unlabeled when the labeling step is applied to e'. By assumption there is a path of labeled edges in F_i connecting x and an endpoint, say z, of e'. Each of the edges on this path defines a path of labeled edges in F_{i+}; thus x and z are connected by a path of labeled edges in F_{i+}. Since y and z are both on $F_{i+}(e')$, all of whose edges are labeled, x and y are connected by a path of labeled edges in F_{i+}. This contradiction implies the third part of the claim.

The claim implies that $F \cup \{e_0\}$ contains $k(|S|-1)+1$ edges with both endpoints in S. Thus $F \cup \{e_0\}$ is dependent, since any independent set must have at most $k(|S|-1)$ edges with both ends in S. ∎

By using the labeling algorithm for testing independence and using augmentation to update the partition of F into forests, we obtain an efficient version of the greedy algorithm for minimum spanning trees. We can make a small change that improves this method by reducing the time spent on fruitless searches for an augmenting sequence. Suppose the labeling step for some trial edge e_0 stops with an empty queue. Let S be the vertex set of the subtrees $T_1, T_2, \ldots, T_k$ of $F_1, F_2, \ldots, F_k$ defined by the labeled edges. (See the proof of Lemma 2.) No subsequent augmentation will affect any edge both of whose endpoints are in S.

Let us define a *clump* to be a set of vertices that are spanned in every forest F_i by a subtree of that forest. Any edge that has both of its endpoints in a clump cannot be independent of the current set F.

LEMMA 3. *If a set of vertices C is a clump for the current set F, and F is augmented using an augmentation sequence, then C is a clump in the augmented set F'.*

PROOF. None of the edges that interconnect any of the vertices of C in any forest F_i can be in the augmentation sequence. Since these edges remain intact in all forests after the augmentation, C remains a clump. ∎

Lemma 3 states roughly, "once a clump, always a clump." The next lemma shows that the maximal clumps are vertex-disjoint.

LEMMA 4. *If two sets of vertices A and B are clumps containing a common vertex x, then $A \cup B$ is a clump.*

PROOF. For any two vertices $y, z \in A \cup B$ and any $i \in [1 \ldots k]$, there is a path in F_i from y to x containing only vertices in A and another path from y to z containing only vertices in B, giving a path from y to z containing only vertices in $A \cup B$. It follows that $A \cup B$ is a clump. ∎

As we have said, any edge, both of whose endpoints are in a clump, cannot be in an augmenting sequence. We modify the greedy algorithm to discard any edge both of whose endpoints are known to be in a clump before applying the labeling algorithm to the edge. When we start the algorithm we partition the graph into n singleton clumps. Whenever we discover that two vertices x and y are in a common clump, we replace the known clumps currently containing x and y by their union. By using this strategy, every iteration of the labeling algorithm either finds an augmenting sequence, increasing the size of F, or causes the combining of at least two known clumps, reducing the number of known clumps by at least one. Thus the number of iterations of the labeling algorithm is at most $k(n-1)+n-1 = O(kn)$.

3. Implementation of the algorithm. To implement the minimum spanning trees algorithm, we need a way to represent each of the forests $F_1, F_2, \ldots, F_k$ and each of the known clumps. We can maintain an arbitrary partition of the vertices under the operation of union using the well-known fast disjoint set union algorithm [19]. This algorithm maintains a *canonical vertex* for each vertex set and allows us to perform two operations:

find(v): Return the canonical vertix of the set containing v.

unite(v, w): Form the union of the two sets containing v and w and choose a canonical vertex for the new set. This operation destroys the old sets containing v and w.

We use the set union algorithm to maintain a partition of the vertices into known clumps. We call this *partition zero*, and manipulate it by means of the operations $find_0$ and $unite_0$. Before we apply the labeling algorithm to an edge $\{x, y\}$, we test whether $find_0(x) = find_0(y)$. If so, we discard the edge; if not, we apply the labeling algorithm. If the labeling algorithm fails to find an augmenting sequence, we perform $unite_0(x, y)$. During the entire algorithm we perform a total of $2m$ $find_0$ operations and at most $n-1$ $unite_0$ operations. (After the $n-1st$ $unite_0$ operation we can terminate the algorithm, as we must by then have found k spanning trees.)

REMARK 3. In practice it may be more efficient to contract all the clumps into single vertices and work with the contracted graph, but this complicates the algorithm and does not improve its worst-case asymptotic complexity. ∎

For each $i \in [1 \ldots k]$ we maintain another vertex partition, called *partition* i, whose sets are the vertex sets of the trees of F_i. We manipulate partition i by means of $find_i$ and $unite_i$. Partition i allows us to test easily whether a given edge has both its ends in the same tree of F_i.

We also need to store two pieces of information for every edge e. One is an index specifying the forest containing e: if $e \in F_i$, *index*$(e) = i$; if e is not in any F_i, *index*$(e) = 0$. The other is the label of e assigned by the labeling algorithm; if e is not yet labeled, *label*(e) = **null**.

Overall initialization for the algorithm consists of putting each vertex into a singleton set in each of the $k+1$ partitions, initializing each of the edge sets $F_1, F_2, \ldots, F_k$ to be empty, and initializing *index*$(e) = 0$ for every edge e. We sort the edges in nondecreasing order by cost. Then we process the edges in order. To process an edge $e_0 = \{x, y\}$, we compute $c_1 = find_0(x)$ and $c_2 = find_0(y)$. If $c_1 = c_2$ we proceed with the next edge (x and y are in a common clump). If $c_1 \neq c_2$, we apply the labeling algorithm.

Initialization for the labeling algorithm consists of setting $label_i(v)$ = **null** for each vertex v and each index $i \in [1 \ldots k]$ and initializing the queue to contain the edge e_0. We also compute certain information about the structure of the forests F_i. For $i \in [1 \ldots k]$, we find the tree T_i in F_i containing vertex x, root T_i at x, and compute the parent $p_i(v)$ of every vertex v in T_i. We define $p_i(x) = x$ and $p_i(v)$ = **null** if v is not in T_i. The function p_i allows us to easily trace the path in T_i from any vertex v to x.

The time necessary for all initialization is $O(kn)$. Once the initialization is complete we repeat the labeling step until the queue is empty or an augmenting sequence is discovered.

To implement the labeling step efficiently, we use an important property of the set of labeled edges. For any index $i \in [1 \ldots k]$, the labeled edges in F_i define a subtree consisting of part or all of T_i including the root x. (We shall prove this below.) Suppose we delete an edge $e = \{v, w\}$ from the queue. We apply the labeling step to e by carrying out the following computation, where $i = (index(e) \bmod k) + 1$. If $find_i(v) \neq find_i(w)$, then v and w are in different trees of F_i; we stop, having detected an augmenting sequence. If $find_i(v) = find_i(w)$, one of the vertices v and w is in the subtree of labeled edges in F_i. If v is in the subtree we define $u = w$; otherwise we define $u = v$. (A vertex z is in the subtree if and only if $z = x$ or *label*$(\{z, p_i(z)\})$ $\neq$ **null**.) We find the unlabeled edges on $F_i(e)$ by ascending through the tree a vertex at a time from z toward x, until reaching either x or a previously labeled edge. The following steps give a more formal definition of this process.

Labeling step. If the queue is empty, stop: $F \cup \{e_0\}$ is dependent. Otherwise, delete the first edge, say $e = \{v, w\}$, from the queue. Let $i = (index(e) \bmod k) + 1$ and proceed as follows:

Initialization. If $find_i(v) \neq find_i(w)$, stop: there is an augmenting sequence. Otherwise, let u be the vertex among v and w such that $u \neq x$ and $label(\{u, p_i(u)\}) = \text{null}$. Initialize to empty a stack of edges to be labeled.

Find unlabeled edges on path. Repeat the following step until $u = x$ or $label(\{u, p_i(u)\}) = \text{null}$: push $\{u, p_i(u)\}$ onto the stack and replace u by $p_i(u)$.

Label edges. Repeat the following step until the stack is empty: pop the top edge e' off of the stack, define $label(e') = e$, and add e' to the queue.

LEMMA 5. *The labeled edges in F_i define a subtree consisting of part or all of T_i including the root x. When an edge $e = \{v, w\}$ is processed in the labeling step, either v or w is in T_i, where $i = (index(e) \bmod k) + 1$.*

PROOF. The first part of the lemma is immediate, since any edge in F_i that becomes labeled is either incident to x or shares an endpoint with a previously labeled edge in F_i. (The stack mechanism guarantees this.) We prove the second part of the lemma by induction on the number of processed edges. Let $e = \{v, w\}$ be a processed edge. Either $x \in \{v, w\}$ or there is a previously processed edge in F_i with a common endpoint, say v. Then v is in T_i by the induction hypothesis. ∎

Lemma 5 implies that the method above is a correct implementation of the labeling algorithm. When the algorithm stops, we must update the forests and the vertex partitions. If no augmenting sequence has been found, we perform $unite_0(x, y)$. (Recall that $e_0 = \{x, y\}$ is the edge to which the labeling algorithm was applied.) If an augmenting sequence has been found, we perform the corresponding augmentation. Suppose the labeling algorithm stops, having removed from the queue an edge $e = \{v, w\}$ such that $find_i(v) \neq find_i(w)$ for some $i \in [1, \ldots, k]$. We carry out the corresponding augmentation by performing $unite_i(v, w)$, repeating the following step unitl $label(e) = \textbf{null}$, and replacing $index(e)$ by i (this adds e to F_i).

Swap step. Simultaneously replace e, $index(e)$, and i by $label(e)$, i, and $index(e)$, respectively. (This removes e from $F_{index(e)}$ and adds it to F_i.)

REMARK 4. This implementation of the labeling algorithm is critical to obtaining the desired complexity bound. If we were to examine every edge of $F_i(e)$ for each edge e removed from the queue, the running time would increase by a factor of n. Our implementation guarantees that labeled edges are not needlessly re-examined, giving a running time of $O(kn)$ for the labeling algorithm. ∎

A straightforward analysis of the running time of the minimum spanning trees algorithm shows that the time is $O(m \log m + kn)$ for overall initialization and processing of edges $e_0 = \{x, y\}$ such that $find_0(x) = find_0(y)$, plus $O(kn)$ per execution of the labeling algorithm, including the time for updating after the algorithm halts. These bounds count each *find* as taking $O(1)$ time. Since there are $O(kn)$ applications of the labeling algorithm, the total time is $O(m \log m + k^2n^2)$, with each *find* counted as taking $O(1)$ time. The upper bound of Tarjan [17] for the disjoint set union algorithm implies that the amortized time per find is indeed $O(1)$, giving an overall time bound of $O(m \log m + k^2n^2)$. (The total number of finds is $O(m + k^2n^2) = O(k^2n^2)$ on sets containing a total of $O(kn)$ elements; the fact that the number of finds is quadratic in the number of elements implies an $O(k^2n^2)$ bound on the finds.) If all the edge weights are equal, the $O(m \log m)$ time needed to sort the edges is unnecessary, and the overall time bound is $O(k^2n^2)$. For any fixed value of k independent of m and n, the bound is $O(m \log m + n^2)$ for the weighted case, $O(n^2)$ for the unweighted case.

There are several potential improvements that can be made to our algorithm, but so far none of then has been shown to give an improvement in the asymptotic complexity. For example, it is possible to place $kn/2$ edges into k forests in $O(m + kn)$ time, compared with the $O(m + k^2n^2)$ time implied by our algorithm. See [18] for several such ideas.

4. Extension, applications, and related work. Our minimum spanning trees algorithm can be extended to solve the following problem in $O(m^2)$ time: find a maximum number of edge-disjoint spanning trees. We need to modify the way the labeling algorithm works. We maintain a partition of the edges so far processed into a collection of forests $F_1, F_2, \ldots, F_k$ with the property that for $i \in [1 \ldots k]$, $F_1 \cup F_2 \cup \cdots \cup F_i$ contains as many edges as possible. To apply the labeling algorithm to an edge e_0, we first try to find an augmenting sequence for e_0 with respect to F_1. If we find such a sequence, we perform the corresponding augmentation. If not, we put all labeled edges in F_1 back on the queue and continue the labeling algorithm, but now looking for an augmenting sequence with respect to F_1 and F_2. In general, if we fail to find an augmenting sequence with respect to $F_1, F_2, \ldots, F_i$, we put all labeled edges in (only) F_i back on the queue and continue the labeling algorithm, looking for an augmenting sequence with respect to $F_1, F_2, \ldots, F_{i+1}$. The labeled edges of $F_1, F_2, \ldots, F_i$ remain labeled; to continue the process we need only initialize the data structures for F_{i+1}. If the entire labeling algorithm fails to find an augmenting sequence even with respect to $F_1, F_2, \ldots, F_k$, edge e_0 becomes the first edge of a new forest F_{k+1}.

Once the algorithm is finished, we return the maximum i such that F_i is a tree; $F_1, F_2, \ldots, F_i$ constitute a maximum-cardinality set of edge-disjoint spanning trees. It is important to note that the augmenting sequences found by this algorithm do not in general obey the discipline that successive edges come from successive forests, but they still can be used to perform valid augmentations. (We leave the proof of this as an exercise.)

To analyze the complexity of this algorithm, we note that during the processing of an edge e_0, each labeled edge is put on the queue at most twice. This implies a time bound of $O(m)$ for each iteration of the labeling algorithm, and an $O(m^2)$ time bound overall.

At least three applications of the minimum spanning trees algorithm have appeared in the literature. Clausen and Hansen [4] discuss the possibility of extending the Held–Karp algorithm for the traveling salesman problem [7] to the case of k edge-disjoint tours by using an algorithm for finding k edge-disjoint spanning trees. There are, however, other obstacles to this approach that remain unresolved. The other applications are of the unweighted case for $k = 2$. An algorithm for finding two edge-disjoint spanning trees can be used to solve the Shannon switching game [1], [2], [6], [13], and to do so-called "mixed" analysis of electrical networks [10], [17]. Our algorithm runs in $O(n^2)$ time in this case. A previous algorithm by Kameda and Toida [9] has a running time of $O(mn\alpha(m,n))$, where $\alpha(m,n)$ is a functional inverse of Ackerman's function [19].

An alternative approach to the minimum spanning trees problem is to formulate it as a "polymatroidal" network flow problem. This has been done by Lawler and Martel [12] and by Imai [8]. Lawler and Martel give no computational bounds, but Imai derives a bound of $O(k^2n^2)$ for the unweighted k-trees problem and $O(m^2 \log n)$ for maximizing the number of disjoint trees. The former bound is the same as ours; the latter is worse by a factor of $\log n$. The relationship between this approach and ours remains to be elucidated.

References

[1] Bruno, J. and Weinberg, L. (1970). A Constructive Graph-Theoretic Solution of the Shannon Switching Game. *IEEE Trans. Circuit Theory* **CT-17** 74–81.

[2] Chase, S. M. (1972). An Implemented Graph Algorithm for Winning Shannon Switching Games. *Comm. ACM* **15** 253–256.

[3] Cheriton, D. and Tarjan, R. E. (1976). Finding Minimum Spanning Trees. *SIAM J. Comput.* **5** 724–742.

[4] Clausen, J. and Hansen, L. A. (1980). Finding k Edge-Disjoint Spanning Trees of Minimum Total Weight in a Network: An Application of Matroid Theory. *Math. Programming Study* **13** 88–101.

[5] Edmonds, J. (1965). Minimum Partition of a Matroid into Independent Subsets. *J. Res. Nat. Bur. Standards* **69B** 67–72.

[6] ———. (1965). Lehman's Switching Game and a Theorem of Tutte and Nash-Williams. *J. Res. Nat. Bur. Standards* **69B** 73–77.

[7] Held, M. and Karp, R. M. (1971). The Traveling Salesman Problem and Minimum Spanning Trees. Part II. *Math. Programming* **1** 6–25.

[8] Imai, H. (1983). Network Flow Algorithms for Lower-Truncated Transversal Polymatroids. *J. Oper. Res. Soc. Japan* **26** 186–211.

[9] Kameda, T. and Toida, S. (1973). Efficient Algorithms for Determining An External Tree of a Graph. *Proc. 14th IEEE Sympos. on Switching and Automata Theory* 12–15.

[10] Kishi, G. and Kajitani, Y. (1969). Maximally Distant Trees and Principal Partition of a Linear Graph. *IEEE Trans. Circuit Theory* **CT-16** 323–330.

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

[12] ——— and Martel, C. U. (1982). Flow Network Formulations of Polymatroid Optimization Problems. *Ann. Discrete Math.* **16** 189–200.

[13] Lehman, A. (1964). A Solution to the Shannon Switching Game. *SIAM J. Appl. Math.* **12** 687–725.

[14] Nash-Williams, C. St. J. A. (1961). Edge-Disjoint Spanning Trees of Finite Graphs. *J. London Math. Soc.* **36** 445–450.

[15] ———. (1964). Decomposition of Finite Graphs and Forests. *J. London Math. Soc.* **39** 12.

[16] ———. (1966). An Application of Matroids to Graph Theory. *Theory of Graphs: Internat. Sympos.*, Gordon and Breach, New York, 263–265.

[17] Ohtsuki, T., Ishizaki, Y. and Wantanabe, H. (1970). Topological Degrees of Freedom and Mixed Analysis of Electrical Networks. *IEEE Trans. Circuit Theory* **CT-17** 491–499.

[18] Roskind, J. (1983). Application of Edge Disjoint Trees to Failure Recovery in Data Communication Networks. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge; also Laboratory Report, Laboratory for Information and Decision Sciences, MIT.

[19] Tarjan, R. E. (1975). Efficiency of a Good but Not Linear Set Union Algorithm. *J. Assoc. Comput. Mach.* **22** 215–225.

[20] Tutte, W. T. (1961). On the Problem of Decomposing a Graph into n Connected Factors. *J. London Math. Soc.* **36** 221–230.

[21] Yao, A. C. (1975). An $O(|E|\log\log|V|)$ Algorithm for Finding Minimum Spanning Trees. *Inform. Process. Lett.* **4** 21–23.

ROSKIND: MASSACHUSETTS INSTITUTE OF TECHNOLOGY, CAMBRIDGE, MASSACHUSETTS 02139

Current Address: HARRIS COPORATION, GOVERNMENT INFORMATION SYSTEMS DIVISION, MELBOURNE, FLORIDA 32902

TARJAN: AT&T BELL LABORATORIES, 600 MOUNTAIN AVENUE, MURRAY HILL, NEW JERSEY 07974