SIAM J. COMPUT.
Vol. 15, No. 1, February 1986

009

AN $O(EV \log V)$ ALGORITHM FOR FINDING A MAXIMAL WEIGHTED MATCHING IN GENERAL GRAPHS*

ZVI GALIL†, SILVIO MICALI‡ AND HAROLD GABOW§

Abstract. We define two generalized types of a priority queue by allowing some forms of changing the priorities of the elements in the queue. We show that they can be implemented efficiently. Consequently, each operation takes $O(\log n)$ time. We use these generalized priority queues to construct an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs.

Key words. matching, augmenting path, blossoms, generalized priority queues, primal dual algorithm, time complexity

Introduction. We are given a graph $G=(V, E)$ with vertex set V and edge set E. Each edge $(i, j) \in E$ has a weight w_{ij} associated with it. A *matching* is a subset of the edges, no two of which have a common vertex. We want to find a matching with the maximal total weight.

In this paper we deal with the general problem. There are three restricted versions of the problem: we can restrict attention to bipartite graphs, or to maximizing cardinality (unit weights) or both. For a survey on the status of the four versions of the problem see [5]. In the time bounds mentioned below we use V and E for the size of the corresponding sets. No confusion will arise.

Edmonds [3] gave the first polynomial time algorithm to the problem, whose time bound is $O(V^4)$. Lawler [8] and independently Gabow [4] improved Edmonds' algorithm by finding a way to implement it in $O(V^3)$..

We develop an $O(EV \log V)$ algorithm, which is much better for sparse graphs. We note that for the problem of finding a maximal flow in networks, a number of efficient algorithms for sparse graphs have been developed in recent years ([6], [9]), while an $O(V^3)$ algorithm has been known for some time [7]. Our algorithm is also an implementation of Edmonds' algorithm.

Our improvement is derived from some simple observations on data structures. We design two generalized types of a priority queue by allowing some forms of changing the priorities of the elements in the queue. We show that each operation on these priority queues can still be implemented in time $O(\log n)$, where n is the total number of elements.

In § 1 we define the two types of priority queues. In § 2 we show how to implement each operation on these priority queues in time $O(\log n)$. In § 3 we review the notions of augmenting paths and blossoms. In § 4 we describe our version of Edmonds' algorithm. We leave out some details of the implementation. In § 5 we show how a straightforward implementation yields an $O(EV^2)$ algorithm. (Edmonds' bound was

* Received by the editors June 16, 1983, and in revised form May 15, 1984.

† Computer Science Department Tel-Aviv University, Ramat Aviv, Tel-Aviv, Israel, and Computer Science Department, Columbia University, New York, New York 10027. The research of this author was supported in part by the National Science Foundation under grant MCS78-25301 at the University of California at Berkeley, by the Israel Commission of Basic Research, and by the National Science Foundation under grant MCS-8303139 at Columbia University.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02138. The research of this author was supported in part by DARPA under grant N00039-82-C-0235 and by the National Science Foundation under grant MCS82-04506 at the University of California at Berkeley.

§ Computer Science Department, University of Colorado, Boulder, Colorado 80309. The research of this author was supported by the National Science Foundation under grant MCS-8302648.

$O(V^4)$.) Then we show the changes needed to obtain an $O(V^3)$ bound, yielding (a more complete version of) the algorithm by Lawler [8]. In § 6 we show how to use the generalized priority queues to obtain the $O(EV \log V)$ algorithm.

1. Generalized priority queues. A *priority queue* [1] or a p.q. in short is an abstract data structure consisting of a collection of *elements*, each with an associated real valued *priority*. Three operations are possible on a p.q.:

(1) insert an element i with priority p_i;

(2) delete an element; and

(3) find an element with the minimal priority.

An implementation of a p.q. is said to be *efficient* if each operation takes $O(\log n)$ time where n is the number of elements. Many efficient implementations of p.q.'s are known, e.g., 2-3 trees [1].

In p.q.'s elements have *fixed* priorities. We consider here the following question. What happens if we allow the priority of the elements to change? Obviously, an additional operation which changes the priority of one element can be easily implemented in time $O(\log n)$. On the other hand, it is not natural to allow arbitrary changes in an arbitrary subset of the elements in one operation simply because one has to specify all these changes.

We introduce two generalized types of p.q.'s which we denote by p.q.$_1$ and p.q.$_2$. The first simply allows a uniform change in the priorities of *all* the elements currently in it. The second allows a uniform change in the priorities of an easily specified subset of the elements.

More precisely, p.q.$_1$ enables the following additional operation:

(4) subtract from the priorities of all the *current* elements some real number δ.

This type of p.q. is not new. A version of p.q.$_1$ was used by Tarjan [10]. Note that in (4) we can add instead of subtract. In our case we will mostly subtract $\delta > 0$.

To define p.q.$_2$ we first need some assumptions. We assume that the elements are partitioned into groups. Every group is either *active* or *nonactive*. An element is active (or not) if its group is. We assume that the elements are totally ordered. By splitting a group according to an element i we mean to create two groups from all the elements in the group greater (not greater) than i. Note that unlike the usual split operation we split a group according to an element and not according to its priority.

The operations possible for p.q.$_2$ are:

(1)′ insert an element i with priority p_i to one of the groups;

(2)′ delete an element;

(3)′ find an *active* element with the minimal priority;

(4)′ decrease the priorities of all the *active* elements by some real number δ;

(5)′ generate a new empty group (active or not);

(6)′ delete a group (active or not);

(7)′ change the status of a group from active to nonactive or vice versa; and

(8)′ split a group according to an element in it.

In § 6 we use p.q.$_1$ and p.q.$_2$ to obtain an improved algorithm for finding a maximal weighted matching in general graphs.

2. An efficient implementation for p.q.$_1$ and p.q.$_2$. It may look at first that one may need up to n steps to update all the priorities as a result of one change. However, it is possible to implement efficiently p.q.$_1$ and p.q.$_2$. In particular, the change of priorities will be achieved implicitly by *one* operation.

p.q.$_1$ can be easily simulated by a conventional p.q. We maintain $\Delta = \sum \delta$, where the sum is over all changes δ so far. In the p.q. we use *modified priorities* which are

computed when elements are inserted into the p.q. The modified priority of i is $p_i+\Delta$. So when an element is inserted we add Δ to its priority. The nice property of the modified priority is that, unlike the original priority, it *does not* change. Tarjan's implementation [10] is more complicated because he also allows merging of p.q.'s. Instead of storing priorities he maintains differences of priorities.

The efficient implementation of p.q.$_2$ is less straightforward. Each group g has a p.q. A_g corresponding to it, and each element has its modified priority. However, the modification is not the same for all the elements. If i is inserted into group g, then its modified priority is set to $p_i+\Delta_g$, where $\Delta_g=\sum\delta$, and the sum is over the changes made when g was a part of an active group (possibly g itself). As for p.q.$_1$, these modified priorities do not change. To update Δ_g we maintain Δ_g^{last}, which is the value of Δ when g was last considered (in operations (1)′, (2)′, (7)′, or (8)′). Whenever we consider an active group g, before resetting its Δ_g^{last} we update Δ_g as follows: $\Delta_g \leftarrow \Delta_g+\Delta-\Delta_g^{\text{last}}$. When we split a group g to groups g_1 and g_2 we set $\Delta_{g_i}, \Delta_{g_i}^{\text{last}} \leftarrow \Delta_g$ for $i=1,2$. We also maintain a p.q.$_1$ B which contains one element with the minimal priority from every active group.

Implementing the first seven operations is quite easy. Note that an insert to or a delete from A_g may require an insert to or a delete from B (or both). Note also that if i is in group g and its modified priority which is stored in A_g is q_i $(=p_i+\Delta_g)$, then if it is inserted to B, the modified priority in the p.q. that implements B should be $q_i-\Delta_g+\Delta$. To efficiently implement a split one needs to make a key observation on 2-3 trees. We need the observation because, unlike conventional p.q.'s, we split according to an element and not its priority.

In [1] two kinds of priority queues are described. In the first kind the elements are stored in the leaves and each internal node contains the smallest element of the two (or three) subtrees rooted at his sons. In the second kind the elements are stored in the leaves, and in addition the order is preserved; i.e. the smallest element is stored in the leftmost leaf, etc. This kind supports the operations of concatenate and split. Such priorities queues are called *concatenable queues.*

In our case we have two order relations: the priorities and the order of the elements. Fortunatey, the same 2-3 tree can support both. It contains the information of the first kind for handling the priorities, and of the second kind to handle the order of the elements. The ability to handle both simultaneously is the result of the following observation: assume we treat our 2-3 trees as being of the second type; we split them or concatenate them. If we visit and possibly make changes in a node, we also visit all its ancestors in the tree up to the root. These are exactly all the nodes that may be affected and have to be updated if the tree is of the first kind. For more details on the various operations see [1].

3. Blossoms and their representation. We assume that we are given a graph referred to as the *original graph,* and a matching M. The algorithm discovers certain sets of vertices (of odd size) called *blossoms* and shrinks them. It is convenient to consider also the vertices of the graph as (trivial) blossoms of size one. Consequently, at any moment the blossoms constitute the vertices of the *current graph.*

An *alternating path* from a vertex u_0 to a vertex u_r in the original graph is a sequence of edges $\{e_i=(u_{i-1},u_i)\}_{i=1}^r$ such that $u_1,\cdots,u_r$ are distinct and for $i=1,\cdots,r-1$, $e_i\in M$ iff $e_{i+1}\notin M$. An *alternating path* from a blossom B_0 to a blossom B_r (possibly $B_0=B_r$) is a sequence of edges $\{e_i=(u_{i-1},v_i)\}_{i=1}^r$ such that for $i=0,1,\cdots,r$ u_i, $v_i\in B_i$ where $B_1,\cdots,B_r$ are distinct blossoms and for $i=1,\cdots,r-1$, $e_i\in M$ iff $e_{i+1}\notin M$. When the algorithm discovers an alternating path of odd length

$\{e_i=(u_{i-1}, v_i)\}_{i=1}^{r}$ (r odd) from a blossom B_0 to itself ($B_0=B_r$; $e_1, e_r \notin M$), a new blossom B is formed. The blossoms $B_1, \cdots, B_r$ stop being blossoms and are referred to as the subblossoms of B. Consequently, at any time each vertex is in a unique blossom.

Each blossom has a *base* vertex. The base of a trivial blossom is the unique vertex in it. The base of the blossom B defined above is the base of B_r. Note that if b is the base of B and c is a vertex in B then $(b, c) \notin M$. Also if u is in B and is not the base of B, then there is a v in B such that $(u, v) \in M$ and for every w not in B $(u, w) \notin M$.

A nontrivial blossom is represented by the doubly linked list $\{(B_i, e_i)\}_{i=1}^{r}$ and by its base. Note that

Fact 1. For every $1 \leqq i \leqq r-1$, $(e_1, e_2, \cdots, e_i)$ and $(e_r, e_{r-1}, \cdots, e_{i+1})$ are alternating paths from B_0 to B_i. One is of odd length and one of even length. The one of even length is the one whose last edge is in M.

An easy induction on the structure of the blossom implies

Fact 2. In the original graph, there is an even length alternating path from the base of the blossom to any vertex in it.

A vertex i is *matched* if there is an edge (i, j) in M, and is *exposed* otherwise. A blossom is matched (exposed) if its base is. Edges in M are said to be matched. An *augmenting path* is an alternating path between two exposed vertices (blossoms). By Fact 2, any augmenting path between two exposed blossoms can be expanded to an augmenting path in the original graph between the two (exposed) bases of these blossoms.

One can define a tree that represents the structure of a blossom. In this tree $B_1, \cdots, B_r$ are the sons of B, and the leaves are the vertices of the blossom. We call it the *structure tree*. This tree is implicitly represented by the lists $\{(B_i, e_i)\}_{i=1}^{r}$. The tree implies a total order on the vertices of the blossom: $u < v$ if u is to the left of v in the tree. Note that the base of a blossom is its largest vertex.

Although we conceptually consider the blossoms shrunk, we do not actually shrink them. Edges (u, v) retain their identity. So u and v may belong to blossoms but the edge remains (u, v). If we use such an edge and reach a vertex v we will need to find the blossom of v. So in addition we represent blossoms as ordered sets of vertices. The operations that we need are find, concatenate and split [1].

4. The algorithm.

4.1. A sketch of the algorithm. The algorithm applies the primal-dual method [8]. At any moment we have a matching M and an assignment of values to the dual variables: u_i for every vertex i, and z_k for every odd subset B_k of vertices, $|B_k| = 2r_k + 1$, $r_k > 0$. As will be explained below, it is not important to know what is the meaning of the dual variables.

For every edge (i, j) we define

$$\pi_{ij} = u_i + u_j - w_{ij} + \sum_{k:\, i,\, j \in B_k} z_k.$$

By duality theory (see [8]), the matching has maximum weight if (0)-(3) hold for every vertex i, edge (i, j), and odd subset B_k:

(0) $u_i, \pi_{ij}, z_k \geqq 0$;

(1) $(i, j) \in M \Rightarrow \pi_{ij} = 0$;

(2) i exposed $\Rightarrow u_i = 0$; and

(3) $z_k > 0 \Rightarrow B_k$ is full ($|\{(i, j) | i, j \in B_k, (i, j) \in M\}| = r_k$).

In fact, we need duality theory for motivation only. The following short proof implies that if (0)-(3) hold, then the matching M has maximal weight: let u_i, z_k and

π_{ij} be the values associated with M, and let N be any other matching. Then

$$\sum_{(i,j)\in N} w_{ij} \leqq \sum_{(i,j)\in N} (u_i + u_j) + \sum_{(i,j)\in N} \sum_{k:i,j\in B_k} z_k \leqq \sum_i u_i + \sum_k r_k z_k = \sum_{(i,j)\in M} w_{ij}.$$

The first inequality follows from $\pi_{ij} \geqq 0$; the second from u_i, $z_k \geqq 0$ and the fact that N is a matching; and the equality follows from (2), (3) and the fact that M is a matching.

The algorithm will have $z_k > 0$ only for blossoms B_k. Consequently the number of positive z_k's will be small ($O(V)$). Moreover, (3) will hold automatically.

We start with $M = \Phi$ and $u_i = (\max_{k,l} w_{k,l})/2$ for all i and no blossoms (and no z_k's). So except for (2) all other conditions for optimality hold. The algorithm makes changes that preserve (0), (1), (3) and eventually reduce the number of violations of (2) to zero. The resulting matching therefore has maximal weight.

4.2. The search. The main part of the algorithm consists of a search for an augmenting path between two exposed blossoms. The search uses only edges (i, j) with $\pi_{ij} = 0$. During the search, blossoms are labeled by S and T, where an S (T) label denotes an even (odd) length alternating path from an exposed blossom. (Other papers use outer and inner for S and T.) A blossom labeled by S (T) is referred to as an *S-blossom* (a *T-blossom*). A vertex in an S-blossom (a T-blossom) is an *S-vertex* (a *T-vertex*). We also have free blossoms—those without a label, and free vertices—those in free blossoms. During the search new (S) blossoms can be generated. The search may lead to the discovery of an augmenting path. In this case the matching is augmented and we have two less exposed vertices and consequently two less violations of (2). After an augmentation all the labels are erased. So, all blossoms become free. Each augmentation terminates a *stage.*

Initially all exposed blossoms are labeled S. Then the search uses *useful* edges to label more blossoms. A useful edge is an unmatched edge (i, j) with $\pi_{ij} = 0$, i an S-vertex and j is either a free vertex (Case 1) or an S-vertex in a blossom different from the blossom of i (Case 2).

Case 1. j is in a free blossom B with base b. In this case B is labeled with $[T, (i, j)]$. There must be an edge in M of the form (b, c) (otherwise B would be labeled by S). Assume c is in a blossom C. C must be free because we always use immediately the edge in the matching. (It cannot be labeled S because an S label arrives always through a matched edge, so it could arrive only through (b, c). It cannot be labeled by T because if C were labeled by T, we would have immediately labeled B by S.) We label C by $[S, (b, c)]$. The second part of the label records the edge through which it has arrived. In the case of an S label, this part is redundant because c is the base of C and (b, c) is the unique edge in M that is incident with c.

Case 2. j is in an S-blossom B, i is in an S-blossom $C \neq B$.

Using the second part of the labels, we backtrack along the two paths from exposed blossoms to B and to C. If the exposed blossoms are different, an augmenting path has been found. If they are the same, a new blossom is discovered.

If we discover an augmenting path between two exposed blossoms, we first change the status of the edges on the path (from matched to unmatched and vice versa). Consider a blossom B on this path and the two edges $e \in M$ and $e' \notin M$ incident with it. The first enters b, the base of B, and the second leaves through some vertex c that is in some subblossom B_i of B. (See Fig. 1.) We recursively find the even length alternating path in B from b to c (guaranteed by Fact 2) and change the status of its edges: Using the list of subblossoms of B and Fact 1, we find the alternating path through the subblossoms of B ($e_1, \cdots, e_i$ or $e_{i+1}, \cdots, e_k$) of even length. We change the status of the edges on this even length path. We also change the base of B to c

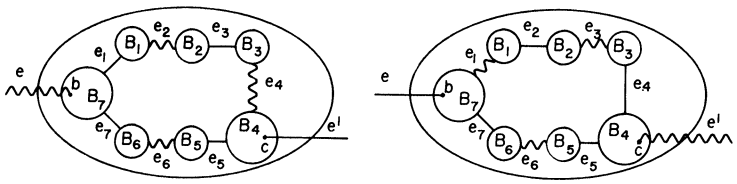

FIG. 1. *Recursively finding the augmenting path. Matched edges are drawn wiggly.* (a) *Before the augmentation in a blossom B. The base is b and the list of subblossoms* $\{(B_1, e_1), \cdots, (B_7, e_7)\}$. (b) *After the augmentation in B. The base is c and the subblossom list is* $\{(B_3, e_4), (B_2, e_3), (B_1, e_2)(B_7, e_1), (B_6, e_7), (B_5, e_6), (B_4, e_5)\}$.

and cyclically permute the list of subblossoms of B (so B_i is now last). We continue recursively with the subblossoms along this even length path. The parts of the alternating paths inside the two exposed blossoms are found similarly.

In case the backtracking leads to the same exposed blossom, we find the first common blossom D on the two paths. We use the parts of the paths from D to B and to C to generate the list $\{(B_i, e_i)\}_{i=1}^{r}$ for the new blossom. $B_r = D$ and e_i are taken from the two paths. We initialize the dual variable associated with the new blossom to 0, and label the new blossom by S.

During the search we choose any useful edge and act according to the case we are in. As a result, some useful edges may stop being useful and some edges may become useful. The search may succeed (if we find an alternating path) or fail (if there are no more useful edges).

4.3. A change in the dual variables. If the search fails, we make the following changes in the dual variables. We choose $\delta > 0$ and execute:

(a) $u_i \leftarrow u_i - \delta$ for every S-vertex i;

(b) $u_i \leftarrow u_i + \delta$ for every T-vertex i;

(c) $z_k \leftarrow z_k + 2\delta$ for every S-blossom B_k; and

(d) $z_k \leftarrow z_k - 2\delta$ for every T-blossom B_k.

Such a choice of δ preserves (1) and (3). To preserve (0) we choose $\delta = \min(\delta_1, \delta_2, \delta_3, \delta_4)$, where

$$\delta_1 = \min_{i:\ S\text{-vertex}} u_i$$

$$\delta_2 = \min_{\substack{(i,j)\in E \\ i:\ S\text{-vertex} \\ j:\ \text{free vertex}}} \pi_{ij}$$

$$\delta_3 = \min_{\substack{(i,j)\in E \\ i,j:\ S\text{-vertices not in the same blossom}}} (\pi_{ij}/2)$$

$$\delta_4 = \min_{B_k\ a\ T\text{-blossom}} (z_k/2)$$

Note that $\delta_1 = u_{i_0} = (\max_{k,l} w_{k,l})/2 - \Delta$, where i_0 is any exposed vertex and Δ is the sum of the changes δ so far. This is because initially $u_i = (\max_{k,l} w_{k,l})/2$ for every S-vertex i, and the fact that the exposed vertices were always S-vertices and their u_i's were always decreased by δ. Consequently, if $\delta = \delta_1$, then after the change (2) is satisfied and we have a matching with maximal weight.

If $\delta = \delta_4$, we expand all T-blossoms B_k on which the minimum was attained. (Their z_k becomes 0.) Expanding a blossom B is described in Fig. 2. B stops being a blossom and its subblossoms become blossoms. The label of B is $[T, (p, q)]$ where (p, q) is the edge through which B received its T label. Assume $q \in B_i$, where $B_1, \cdots, B_r$ are the subblossoms of B. The subblossoms on the odd length path from $B_0 = B_r$ to B_i (see Fact 1) except for B_0 and B_i become free. The ones on the even length path get alternating labels starting and ending with T. It is here where we need the split operation. For $i = 1, \cdots, r-1$, we split each B_i from B according to its base which is its largest element. As a result of expanding a T-blossom some edges may become useful. If that is the case we resume the search. Otherwise we make another change of the dual variables.

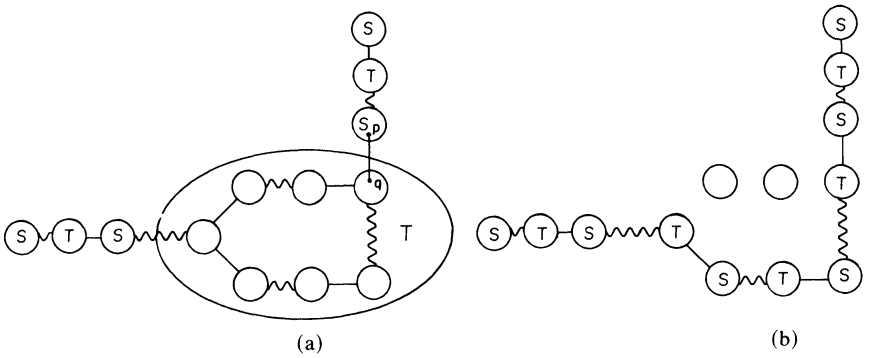

FIG. 2. *Expanding a T-blossom*: (a) *before, and* (b) *after the expansion.*

If $\delta = \delta_2$ ($\delta = \delta_3$), all edges (i, j) with i an S-vertex and j a free vertex (an S-vertex not in the same blossom) on which the minimum was attained become useful (their π_{ij} becomes 0) and we resume the search. The two cases correspond to the two cases in § 4.2.

At the end of each stage we also expand all S-blossoms B_k with $z_k = 0$. Note that finding the alternating path within a blossom can be deferred to the time we expand it. This way we save the repeated changes within the same blossom.

Keeping the blossoms with positive dual variables to the next stage is important. This makes sure that (3) always holds. This explains why T-blossoms can be generated. The latter are expanded whenever their dual variables become 0.

5. The known algorithms. Let us call a *substage* each change in the dual variables. Obviously, there are $O(V)$ stages. There are $O(V)$ different blossoms per stage: each S-blossom corresponds to a unique node in one of the structure trees at the end of a stage. Each T-blossom (free blossom) corresponds to a unique node in one of the structure trees at the beginning of the stage. But, whenever $\delta = \delta_2$ ($\delta = \delta_3$) a new T-blossom (S-blossom) is generated, and whenever $\delta = \delta_4$ a T-blossom is expanded. Hence, $\delta = \delta_i$, $i = 2, 3, 4$, at most $O(V)$ times per stage. Finally, $\delta = \delta_1$ at most once. Consequently, there are $O(V)$ substages per stage.

The most costly part in a substage is finding useful edges and computing δ. The obvious way to do it takes $O(E)$ steps (in each substage we consider all the edges) and yields an $O(EV^2)$ algorithm. To maintain the sets one uses ordered lists for concatenate and split and an array for the find. The naive implementation costs $O(V^3)$. (There are $O(V)$ concatenates and splits per stage, each costs $O(V)$.) The cost of maintaining the dual variables is also $O(V^3)$ ($O(V)$ per substage). The resulting

algorithm is essentially Edmonds' algorithm. The time bound that was given for it was $O(V^4)$ because E was bounded above by V^2.

The only parts which require more than $O(V^3)$ are maintaining δ_2 and δ_3 and finding useful edges. The latter is handled automatically because $\delta_2 = 0$ ($\delta_3 = 0$) iff there are useful edges of Case 1 (Case 2) and these are the edges on which the minimum (0) is attained. We show first how to take care of δ_2. For every free vertex (T-vertex) j let

$$\pi_j = \min_{\substack{(i,j)\in E \\ i:\ S\text{-vertex}}} \pi_{ij}.$$

Then

$$\delta_2 = \min_{j:\ \text{free vertex}} \pi_j.$$

Together with π_j we record an edge (i,j), i an S-vertex, such that $\pi_j = \pi_{ij}$. For each change of δ, we only change π_j for free vertices j. Consequently, the changes of $\{\pi_j\}$ and computing δ_2 cost $O(V^3)$. Recall that free vertices may become T-vertices (when a blossom is labeled by T) and T-vertices may become free (when we expand a T-blossom). That is why we need π_j's for T-vertices as well.

To take care of δ_3, we define for every pair of S-blossoms B_k, B_l,

$$\varphi_{k,l} = \min_{\substack{(i,j)\in E \\ i\in B_k,\, j\in B_l}} (\pi_{ij}/2).$$

We record the edge $e_{k,l}$ on which the minimum is attained and maintain $\varphi_k = \min_l \varphi_{k,l}$. We do not maintain $\varphi_{k,l}$, but any time we need it we compute it by using $e_{k,l}$. Obviously $\delta_3 = \min_k \varphi_k$. The changes in the dual variables and computing δ_3 cost $O(V^3)$ as for δ_2. We have to update $\{\varphi_k\}$ and $\{e_{k,l}\}$ any time an S-blossom B_k is constructed from $B_{i_1}, \cdots, B_{i_r}$. Recall that $(r+1)/2$ of them are S-blossoms and $(r-1)/2$ of them are T-blossoms. We first "make" each T-blossom B_m an S-blossom by scanning all its edges and computing for it $\{\varphi_{m,l}\}$ and $\{e_{m,l}\}$. Then we use the $\varphi_{m,l}$'s of $B_{i_1}, \cdots, B_{i_r}$ to compute φ_k, $\{e_{k,l}\}$ for the new blossom B_k, and to update $\{\varphi_j\}$ for $j \neq k$.

The total cost (per stage) to make T-blossoms S-blossoms is $O(E)$. We now compute $T(n)$, the rest of the cost of maintaining δ_3, where n is the number of S-blossoms plus the number of non S-vertices in the graph. As above, assume that a new S-blossom is constructed from r subblossoms. It follows that $T(n) \leqq crn + T(n-r+1)$ because rn is a bound on the number of $\varphi_{k,l}$'s considered after making the T-blossoms S-blossoms. $T(n) = O(n^2)$ (by induction on n), and the cost of computing δ_3 is $O(V^3)$. The resulting $O(V^3)$ algorithm is essentially a (more complete version of) Lawler's algorithm [8].

6. The $O(EV \log V)$ algorithm. The most costly part of Edmonds' algorithm is the frequent updates of the dual variables, which cause changes in $\{\pi_{i,j}\}$. Note that all the elements that determine each δ_i are decreased by δ for each change in the dual variables.

The new algorithm is also an implementation of Edmonds' algorithm. The high level description of § 4 (including the search, augmenting the matching, the change of dual variables and the resulting changes in the blossoms) is identical. The main difference is in maintaining the δ_i's by generalized priority queues that we describe next.

We maintain δ_1 by a p.q.$_1$. In this p.q. the elements are the S-vertices i and their priorities u_i. We do not need this p.q.$_1$ for computing δ_1, since $\delta_1 = u_{i_0} = (\max_{k,l} w_{k,l})/2 - \Delta$ where i_0 is any exposed vertex and Δ is the sum of the δ's so far. We use a p.q.$_1$ because we need to maintain the u_i's for computing π_{ij} when the edge

(i, j) is considered. For the same reason we maintain another p.q.$_1$ for the u_i's of the T-vertices.

We maintain δ_3 by a p.q.$_1$. The p.q. contains all *good edges* (i, j) with i and j in different S-blossoms as well as some *superfluous edges* (i, j) with i and j in the same S-blossom. The reason for having superfluous edges is that we do not have time to locate them and delete them any time a new S-blossom is constructed. The priority of a good edge (i, j) is $\pi_{ij}/2$.

We maintain δ_4 by a p.q.$_1$. The elements in the p.q. are the T-blossoms B_k and their priority $z_k/2$. We have a similar p.q.$_1$ for the S-blossoms, because we need to maintain their z_k. (At the end of a stage they become free and in the next stage they may become T-blossoms.)

If we try to maintain δ_2 by a p.q.$_1$, we have a difficulty. Consider Fig. 3. Initially there may be a large free blossom B_1. At that time all edges in Fig. 3 should be considered for finding the value of δ_2. B_1 may become a T-blossom. Then these edges are not among those edges that determine δ_2. Later on B_1 may be expanded and one of its subblossoms, B_2, may become free. The latter may later become a T-blossom and so on. A simple implementation requires the consideration of each such edge an unbounded number of times (up to k in Fig. 3).

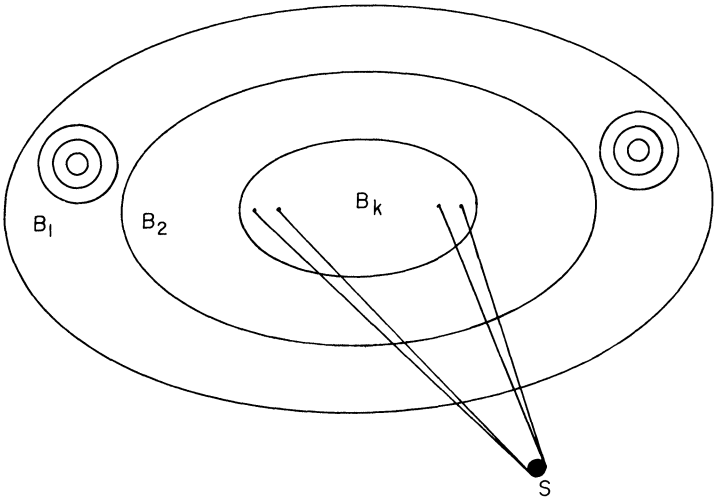

FIG. 3. *Edges from an exposed vertex to the innermost blossom that we may have to consider again and again if the blossoms* $B_1, \cdots, B_k$ *are eventually expanded.*

To maintain δ_2 we have a p.q.$_2$. For every free blossom (T-blossom) B_k we have an active (a nonactive) group of all the edges from S-vertices to vertices in B_k. The priority of an edge (i, j) is π_{ij}. Note that if (i, j) is in a nonactive group (i is an S-vertex and j is a T-vertex), then π_{ij} does not change when we make a change in the dual variables. It is now easy to verify that the eight operations of p.q.$_2$ suffice for our purposes.

Consider a group g which corresponds to a blossom B. The elements of the group are the edges $\{(i, j) | i \text{ an } S\text{-vertex}, j \in B\}$. The order on the elements is derived from the order on the vertices of B. The order between two edges (i_1, j) and (i_2, j) is arbitrary. The order enables us to split the group corresponding to B to the groups corresponding to $B_1, \cdots, B_r$ when we expand B to its subblossoms.

The search is similar to the one described in § 4.2. The labeling process is identical. During the search, whenever we have a new S-vertex i we consider in turn *all* the

edges (i, j). This requires a queue Q for new S-vertices, since we sometimes have many new S-vertices at once. When considering an edge (i, j) we distinguish between 3 cases depending on the type of B the blossom of j.

Case I (II). B is a free blossom (T-blossom). We insert (i, j) with priority π_{ij} to the active (nonactive) group corresponding to B.

Case III. B is an S-blossom. If the blossom of i is not B we insert (i, j) with priority $\pi_{ij}/2$ to the p.q.$_1$ that maintains δ_3.

During the search we compute $\delta = \min(\delta_1, \delta_2, \delta_3, \delta_4)$. If $\delta > 0$, we make a change of δ in the dual variables. This is accomplished by increasing Δ by δ, and results in a new value of $\delta = 0$.

If $\delta = 0$, we consider all $\delta_l = 0$. If $\delta_1 = 0$, then we are done. If $\delta_2 = 0$, this means that the minimum (0) is achieved on an edge (i, j) j in a free blossom B; i.e. (i, j) is useful. We delete (i, j) from the corresponding p.q. and label as in Case 1 of § 4.2. In addition the group corresponding to B becomes nonactive (B is labeled by T) and the group corresponding to C is deleted and the vertices in C (that become S-vertices) are inserted into Q. We repeat the above as long as $\delta_2 = 0$.

If $\delta_3 = 0$ we delete one by one the elements (i, j) in this p.q. with priority $\pi_{ij} = 0$. If i and j are in the same blossom we do not do anything. Otherwise ((i, j) is useful) we act as in Case 2 of § 4.2. If a new S-blossom is generated, then for all the subblossoms B_i that were T-blossoms up till now we delete the group corresponding to B_i (from the p.q.$_2$ of δ_2) and insert all the vertices of B_i to Q.

If $\delta_4 = 0$, we delete one by one the elements B_k in this p.q. with priority $z_k = 0$. For each such B_k, we expand it and label the new blossoms (the previous subblossoms of B_k) as in § 4.3 and Fig. 2. We split the corresponding group in the p.q.$_2$ of δ_2. The groups corresponding to the new free blossoms (T-blossoms) are inserted as active (nonactive) groups to the p.q.$_2$. The vertices of the new S-blossoms are inserted to Q.

To derive an $O(EV \log V)$ time bound we need to implement carefully two parts of the algorithm:

1. We maintain the sets of vertices in each blossom (for finding the blossom of a given vertex) by concatenable queues [1]. Note that the number of finds, concatenates and splits is $O(E)$ per stage.

2. Assume we consider an edge (i, j) where both i and j are S-vertices not in the same blossom. If we execute the backtracking as described above, we may need up to $O(V^3)$ time. Instead, we make a careful backtrack by backtracking one blossom on both paths each time, marking the blossoms on the way. If there are r subblossoms in the new blossom, then we will visit at most $2r$ blossoms before discovering the first common blossom on both paths (D). So the total number of blossoms that we traverse in one stage is $O(V)$. (Charge 2 each one of the corresponding nodes in the corresponding structure tree.)

The time bound is easily derived as follows. There are at most V augmentations. Between two augmentations we consider each edge at most twice and have $O(E)$ operations on (generalized) p.q.'s. (This includes 1 and 2 above.)

Note added in proof. The $Q(EV \log V)$ algorithm for finding weighted matching in general graphs has been recently improved (slightly) to $O(EV \log \log \log_d V + V^2 \log V)$, where $d = \max(E/V, 2)$ [11]. This time bound equals $O(EV)$ if $E = \Omega(V^{1+a})$, for any $a > 0$ and consequently is $o(EV \log V)$ unless $E = \Omega(V^2)$ and is $o(V^2)$ unless $E = O(V)$. The new algorithm is similar to the $O(EV \log V)$ algorithm. The main difference is the use of new data structures instead of regular p.q.'s in the p.q.$_1$'s and p.q.$_2$.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. EDMONDS, *Path, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449-467.

[3] ———, *Maximum matching and a polyhedron with* 0, 1 *vertices*, J. Res. NBS 69B (April-June 1965), pp. 125-130.

[4] H. N. GABOW, *Implementation of algorithms for maximum matching on nonbipartite graphs*, Ph.D. thesis, Stanford Univ., Stanford, CA, 1974.

[5] Z. GALIL, *Efficient algorithms for finding maximal matching in graphs*, Tech. Rep., Dept. Computer Science, Columbia Univ., New York, 1983.

[6] Z. GALIL AND A. NAAMAD, *An* $O(EV \log^2 V)$ *algorithm for the maximal flow problem*, J. Comput. System Sci., 21 (1980), pp. 203-217.

[7] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434-437.

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

[9] D. D. SLEATOR, *An* O $(mn \log m)$ *algorithm for maximum network flow*, Ph.D. thesis, Stanford Univ., Stanford, CA, December 1980.

[10] R. E. TARJAN, *Finding optimum branchings*, Networks, 7 (1977), pp. 25-35.

[11] H. N. GABOW, Z. GALIL AND T. H. SPENCER, *Efficient implementation of graph algorithms using contractions*, Proc. 25th IEEE Symposium on Foundations of Computer Science, 1984, pp. 347-357.