Efficient Algorithms for Finding Maximum Matching in Graphs

ZVI GALIL

Department of Computer Science, Columbia University, New York, N.Y., 10027 and Tel-Aviv University, Tel-Aviv, Israel

This paper surveys the techniques used for designing the most efficient algorithms for finding a maximum cardinality or weighted matching in (general or bipartite) graphs. It also lists some open problems concerning possible improvements in existing algorithms and the existence of fast parallel algorithms for these problems.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Algorithmic tools, the asexual case, assignment problem, augmenting path, blossoms, data structures, d-heap, duality, ET, generalized priority queue, Main Theorem of Botany, matching, monsters, moonlighting, polygamy, primal–dual method, shmathematics, shrink, warm-up

INTRODUCTION

There are no recipes for designing efficient algorithms. This is somewhat unfortunate from the point of view of applications: Any time we have to design an algorithm, we may have to start (almost) from scratch. However, it is fortunate from the point of view of researchers: It is unlikely that we are going to run out of problems or challenges.

Given a problem, we want to find an algorithm that solves it efficiently. There are three stages in designing such algorithms:

(a) *Shmathematics.* Initially, we use some simple mathematical arguments to characterize the solution. This leads to a simple algorithm that is usually not very efficient.

(b) *Algorithmic Tools.* Next, we try to apply a number of algorithmic tools to speed up the algorithm. Examples of such tools are "divide and conquer" and dynamic programming [Aho et al. 1974]. Alternatively, we may try to find a way to reduce the number of steps in the original algorithm by finding a better way to organize the information.

(c) *Data Structures.* Sometimes we can speed up an algorithm by using an efficient data structure that supports the primitive operations used by the algorithm. We may even resort to the introduction of monsters: very complicated data structures that bring about some asymptotic speedup that is usually meaningful only for very large problem size. (For a real-life monster see Galil [1980].)

In these three stages we sometimes use a known technique: a certain result in mathematics, say, or a known algorithmic tool or data structure. In the more interesting problems we need to invent new techniques

CONTENTS

or refine existing ones for our purposes. We may need to prove new shmathematics, find an appropriate way or invent new ones, or use algorithmic tools in a new way.

A word of caution about shmathematics. In many cases it is not deep; however, that does not mean that its results are trivial. What counts in our case is not how deep or elegant a theorem is, but whether it is useful for improving our algorithm.

Usually, we use all three stages of the design process in the order shown above, but this is not always the case. We do not always use all three. Once we have a quite efficient algorithm, we may reuse any of the three and not necessarily in this order. In particular, we may use shmathematics again and again: first to characterize the solution, and then to analyze the running time by justifying an algorithmic tool or by proving the properties of certain data structures.

In this paper we exemplify the design of efficient algorithms by surveying algorithms for the four closely related problems of finding a maximum cardinality or weighted matching in general or bipartite graphs. Mathematically, these are all special cases of the problem of weighted matching in general graphs. However, we consider them in increasing order of difficulty because the easier the problem, the faster or simpler its solution.

1. THE FOUR PROBLEMS

The input consists of an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. The vertices represent persons, and each edge represents the possibility that its endpoints marry. A *matching* M is a subset of the edges such that no two edges in M share a vertex; that is, we do not allow polygamy.

Problem 1: Max Cardinality Matching in Bipartite Graphs. The vertices are partitioned into boys and girls, and an edge can only join a boy and a girl. We look for a matching with the maximum cardinality. We can make Problem 1 harder in two different ways, resulting in Problems 2 and 3.

Problem 2: Max Cardinality Matching in General Graphs. This is the asexual case, where an edge joins two persons.

Problem 3: Max Weighted Matching in Bipartite Graphs. Here we still have vertices representing boys and girls, but each edge (i, j) has a weight w_{ij} associated with it. Our goal is to find a matching with the maximum total weight. This is the well-known assignment problem of assigning people to jobs (disallowing moonlighting) and maximizing the profit.

Problem 4: Max Weighted Matching in General Graphs. This problem is obtained from Problem 1 by making it harder in both ways.

The four combinatorial problems are important and interesting in themselves. Moreover, many combinatorial problems can be reduced to one of the above-described four or can be solved by using, in turn, the solutions to them as subroutines.

It was not clear initially how to solve Problems 2–4 in polynomial time. The first polynomial-time algorithm for Problem 3 is due to Kuhn [1955]. The first polynomial-time algorithms for Problems 2 and 4 are due to Edmonds. The later improved algorithms are based on Edmond's monumental work (Edmonds 1965a, 1965b).

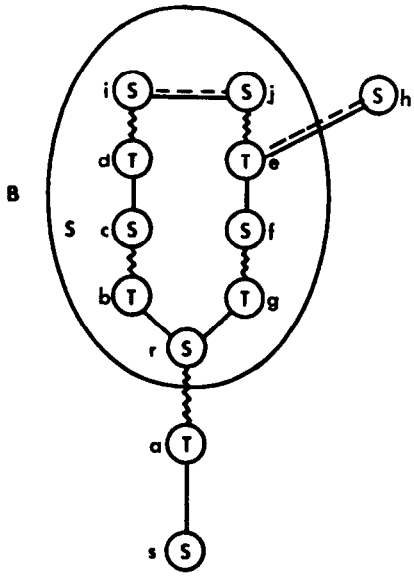

Figure 1. An edge (i, j) is considered and a blossom B is generated. Edge (e, h) has not been considered. Wiggly edges are matched.

2. AN AUGMENTING PATH

An important notion for all four problems is that of an *augmenting path*. We solve each one of them in stages, and in each stage we have a matching M. Initially, M is empty. A vertex i is *matched* if there is an edge (i, j) in M and *single* otherwise. An edge is *matched* if it is in M and *unmatched* otherwise. An *alternating path* (with respect to M) is a simple path, such that every other edge on it is matched. An *augmenting path* (with respect to M) is an alternating path between two single vertices. It must be of odd length, and in the bipartite case its two endpoints must be of different sex.

Consider Figure 1. The wiggly edges are the matched edges. The path $s, a, r, b, c, d, i, j, e, h$ is an augmenting path. Any contiguous part of this path (e.g., b, c, d, i) is an alternating path.

The following theorem is due to Berge [1957] and Norman and Rabin [1959].

Theorem 1

The matching M has maximum cardinality if and only if there is no augmenting path with respect to M.

One part of the theorem is trivial. If there is an augmenting path, then by changing the status of the edges on the path (matched edges become unmatched, and vice versa) we increase the size of M by 1. We call this operation *augmenting the matching M*. The other part of the theorem is not trivial but is quite easy: We assume that M is not a maximum matching and show the existence of an augmenting path. Let M' be a matching of cardinality larger than M. Consider $M \oplus M'$, the set of edges in M or M' but not in both. Hence $M \oplus M'$ consists of alternating paths and cycles (with respect to both M and M'). At least one of them must be an augmenting path with respect to M. (In all other types of alternating paths or cycles the number of edges from M is at least as large as the number of edges from M'.)

3. PROBLEM 1: MAX CARDINALITY MATCHING IN BIPARTITE GRAPHS

Theorem 1 gives an immediate algorithm. It consists of $O(n)$ (at most $n/2$) *stages*. In each stage a search for an augmenting path is conducted. If any augmenting paths exist, the search finds one and the matching is augmented. Since the search takes $O(m)$ time, the algorithm runs in $O(mn)$ time.

The search is conducted as follows. Vertices are labeled successively, boys with an S label and girls with a T label. A labeled boy (girl) is referred to as an S-boy (a T-girl). After all labels from previous stages are cleaned, all single boys are labeled with S. We then apply two labeling rules iteratively:

(R1) If (i, j) is not matched and i is an S-boy and j a *free* (unlabeled) girl, then label j by T; and

(R2) If (i, j) is matched and j is a T-girl and i a free boy, then label i by S.

The label also contains the vertex from which the label has arrived. (In the case of an S label this information is redundant.) The search continues until the search either succeeds or fails. The search succeeds if a single girl is labeled by T. The search fails if we cannot continue anymore. The following lemma can be proved by induction.

Lemma 1

(a) *If a boy i (a girl j) is labeled by S (T), then there is an even- (odd-) length alternating path from a single boy to i (j), and*
(b) *if the search fails and there is an even- (odd-) length alternating path from a single boy to boy i (girl j), then i (j) is labeled by S (T).*

By Lemma 1, if the search fails, there is no augmenting path and the algorithm (not only the stage) terminates. If a single girl j is labeled by T, we have actually found an augmenting path to j. The path can be reconstructed by using the labels. The search can be easily implemented in time $O(m)$ using any traversal of the graph that starts with the single boys. The labels S and T are also called even and odd or outer and inner in the literature.

The best algorithm for Problem 1 is by Hopcroft and Karp [1973]. They discovered a way to find many augmenting paths in one traversal of the graph. Their algorithm is divided into *phases*. In each phase, a maximal set of vertex disjoint augmenting paths of shortest length is found and used to augment the matching. So, a phase may achieve the possible effect of many stages.

We now describe one phase. We use rules R1 and R2 as before. Using breadth-first search, starting from the single boys, we identify the subgraph $\tilde{G}$ of G consisting of all the vertices and edges that are on some shortest augmenting path. This subgraph is *layered*. In layer $2m$ ($2m + 1$) appear all boys i (girls j) such that the *shortest* alternating path from a single boy to i (j) is of length $2m$ ($2m + 1$). We finish the construction of $\tilde{G}$ in one of two ways. Either a single girl is reached and we complete the last layer and delete nonsingle girls from it, or we cannot continue. In the latter case the algorithm (not only the phase) terminates, which is justified by Lemma 1.

In $\tilde{G}$ we find a maximal set of disjoint augmenting paths using depth-first search. Every time we reach a single girl, we find an augmenting path, erase its edges from $\tilde{G}$, and start from another single boy. Every time we backtrack along an edge, we delete it from $\tilde{G}$. It is quite easy to see that a phase takes $O(m)$ time. The importance of the notion of a phase is explained by the following lemma [Hopcroft and Karp 1973].

Lemma 2

The number of phases is at most $O(\sqrt{n})$.

Consequently, Hopcroft and Karp's algorithm runs in time $O(m\sqrt{n})$.

It is interesting to note that the algorithm (not the time analysis, i.e., not Lemma 2) was actually known before. Problem 1 is a special case of the max flow problem for special networks. (Add a source and a sink, connect the source to all the boys and the sink to all the girls, and take all capacities to be one.) Augmenting paths correspond to the flow augmenting paths in network flow, and the $O(mn)$ algorithm is just the Ford and Fulkerson network flow algorithm [Ford and Fulkerson 1956] for these special networks. Similarly, Hopcroft and Karp's algorithm is actually Dinic's algorithm [Dinic 1970] applied to these special networks. This was first observed by Even and Tarjan [1975] (ET).

4. PROBLEM 2: MAX CARDINALITY MATCHING IN GENERAL GRAPHS

As for Problem 1, Theorem 1 suggests a possible algorithm of $O(n)$ stages. In each stage we look for an augmenting path. This search is complicated by the existence of odd-length cycles, which cannot exist in bipartite graphs. We start by labeling all single persons S and apply rules R1 and R2 with the following two changes. First, we replace "boys" or "girls" by "persons." Second, any time R1 is used and j is labeled by T, R2 is immediately used to label the spouse of j with S. (Since j was not labeled before, it must be married and its spouse must be unlabeled.) We call this rule R12.

The search is conducted by scanning the S-vertices in turn. Scanning a vertex means considering in turn all its edges except the matched edge. (There will be at most one.) If we scan the S-vertex i and consider the edge (i, j), there are two cases:

(C1) j is free; or
(C2) j is an S-vertex.

C2 cannot occur in the bipartite case. The case in which j is a T-vertex is discarded.

In case C1 we apply R12. In case C2 we do the following: Backtrack from i and j, using the labels, to the single persons s_i and s_j from which i and j got their S labels. If $s_i \neq s_j$, we find an augmenting path from s_i to s_j and augment the matching. The trouble begins (or life starts to be interesting) when $s_i = s_j$.

We next describe Edmonds' remarkable work, in which the concept of *blossoms* is introduced. Blossoms play a crucial role in all algorithms for the nonbipartite case (Problems 2 and 4).

If $s_i = s_j = s$, let r be the first common vertex on the paths from i and j to s. It is easy to see that r is an S-vertex, that the part of the two paths from i and j to r are disjoint, and that the parts from r to s are identical. We have found an odd-length alternating path from r to itself through (i, j). We call this cycle B a *blossom* and r its *base* (see Figure 1).

In this case the method of labeling introduced so far might overlook an augmenting path. For example, Figure 1 shows a possible labeling. The augmenting path $s, a, r, b, c, d, i, j, e, h$, which "goes around the blossom B," will not be discovered.

Edmonds' idea (Edmonds 1965a) was to *shrink* B: Replace it by a single supervertex B and replace the set A of edges incident with vertices in B by the set $A' = \{(B, j) \mid j \notin B, \exists\ (i, j) \in A\}$. Note that at most one member of A' (incident with r) is matched; none are matched if $r = s$. If $\hat{G}$ is the graph obtained from G after such shrinking, then the shrinking is justified by the following theorem due to Edmonds (which may be called the Blossoms Theorem or the Main Theorem of Botany):

Theorem 2

There is an augmenting path in G if and only if there is an augmenting path in $\hat{G}$.

We do not know of any easy proof for Theorem 2. We do not discuss the "only if" part here (see Lawler [1976] or the second edition of Tarjan [1983]). The "if part" is obvious. Given an augmenting path in $\hat{G}$, it immediately yields an augmenting path in G. If the path goes through B, then we do the following: Replace the matched edge, say (B, k), with (r, k); replace the unmatched edge, say (B, j), with the edge (i, j), from which (B, j) originated (recall the set A' above), followed by the even alternating path in B from i to r. Such a path always exists. If i was an S-vertex when B was formed, we use the labels and backtrack from i to r. Otherwise, we use the labels in reverse order around the blossom. Storing B as a doubly linked list with a marked base makes this very easy. Note that, in Figure 1, as soon as the blossom B is shrunk, the augmenting path can be found. Note also that not every alternating path from a vertex to itself is a blossom, only those discovered by the algorithm (in case C2).

The search for an augmenting path uses a queue Q, where new S-vertices are inserted. During the search, vertices from Q are scanned and new blossoms are occasionally generated. A blossom is a recursive structure because it can be nested. It is convenient to refer to vertices that do not belong to any blossom as (degenerate) blossoms of size 1. When a new blossom B is generated from blossoms $B_1, \ldots, B_k$, we call the latter the *subblossoms* of B. We do not refer to them as blossoms anymore. As a result, each vertex in the *original graph* G always belongs to one blossom in the *current graph*. The structure of each blossom B is described by a tree, called the *structure tree of* B. In this tree, the root is labeled with B, and the sons of a node labeled C are labeled with $C_1, \ldots, C_k$ if at some time the blossom C (which is now part of B) was generated from subblossoms $C_1, \ldots, C_k$. Thus, the leaves of the structure tree of B are the vertices that belong to B. The structure tree is represented by the collection of the doubly linked lists of the various subblossoms of B.

When a new blossom B is generated, it is labeled by S and inserted into Q. Its subblossoms that are still in Q are deleted from Q. The search continues (in $\hat{G}$).

If the search succeeds (in C2), we find the augmenting path in the current graph. Then we use the easy part of Theorem 2 and the structure trees to recursively unwind the augmenting path in the original

graph. We next augment the matching, erase all labels and blossoms, and start a new stage. Constructing the augmenting path, augmenting the matching, and erasing the labels take $O(n)$ time. If the search fails (Q becomes empty), a repeated application of Theorem 2 (each time a blossom is shrunk) and an application of a modified version of Lemma 1 (in which "boys" and "girls" are replaced by "persons") imply that the current matching is maximum, and we are done.

A naive implementation [Edmonds 1965a] takes $O(n^4)$ ($O(n^3)$ per stage). A more careful implementation takes $O(n^3)$ [Gabow 1976a]: Since the blossoms are disjoint, the total size of all structure trees at any moment is $O(n)$. When we generate a new blossom, we do not rename the edges; edges retain their original names. In order to find out quickly to which blossom a given vertex belongs, we maintain a membership array. When B becomes a blossom, we put the T-vertices into the queue Q; so we later scan them instead of scanning the new vertex B. The other vertices of B (the S-vertices) have already been inserted into Q, and we do not delete them from Q. When we consider an edge in C2, we ignore it if both endpoints are in the same blossom. In this implementation a stage takes $O(n^2)$ time.

A slightly better bound can be obtained as follows. If we find the base r of a new blossom B more carefully, by backtracking one vertex at a time, once from i and once from j, marking vertices on the way, we find the base and construct the blossom in time $O(k)$, where k is the number of subblossoms of B. Hence, the total time per stage devoted to finding bases and constructing blossoms is $O(n)$. (Each node in the structure tree is charged $O(1)$.) Using the "set union" algorithm to maintain the sets of vertices in the blossoms for the membership tests takes $O(m\alpha(m, n))$ per stage for a total of $O(mn\alpha(m, n))$, where α is the inverse of Ackermann's function [Tarjan 1975].

The obvious question that comes to mind is whether the ideas of the phases can be realized in the nonbipartite case. Recall that in one phase we discovered a maximal set of vertex disjoint augmenting paths of shortest length. This is important because Lemma 2 holds for general (not necessarily bipartite) graphs. Executing a phase in general graphs is more complicated than in the bipartite case because of the existence of blossoms.

Even and Kariv [1975] showed how to execute a phase in time $\min(n^2, m \log n)$. This resulted in an $O(\min(n^{2.5}, m\sqrt{n} \log n))$ algorithm. A more detailed version [Kariv 1976] is a strong contender for the ACM Longest Paper Award. (It will probably lose only to Slisenko's real-time palindrome recognizer [Slisenko 1973].)

More recently, a simpler approach was found by Micali and Vazirani [1980]. In phase i, $i = 0, 1, \ldots$ of the algorithm, a maximal set of (shortest) augmenting paths of length $L = 2i + 1$ is sought as follows: An *odd* (*even*) *level* of a vertex v is defined as the length of the shortest odd- (even-) length alternating path from a single vertex to v. Both types of level numbers are computed in linear time. Let the *sum* of a free (matched) edge be the sum of the even (odd) levels of its endpoints plus one. A *bridge* is an edge of sum L that belongs to an augmenting path of length L. Every augmenting path of length L is shown to contain a bridge. Bridges are identified and new search techniques applied to find disjoint augmenting paths of length L. We do not give the details here.

The immediate implementation of Micali and Vazirani's algorithm takes $O(m\sqrt{n}\alpha(m, n))$ time. The authors claimed that the particular case of the disjoint set union used by their algorithm can be shown to require only linear time; as a result their algorithm runs in time $O(m\sqrt{n})$. Quite recently, Gabow and Tarjan [1983] found a linear-time algorithm for some special cases of the disjoint set union. One of these special cases is the one needed in Problem 2. (As a result, also the $O(mn\alpha(m, n))$ algorithm mentioned above can be implemented in time $O(nm)$.)

5. SOME OBSERVATIONS ON DATA STRUCTURES

The most efficient algorithms for Problems 3 and 4 use some observations on data structures that we now review. A *priority*

queue (p.q.) 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;
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 in the p.q. Many efficient implementations of p.q.s are known, for example, 2–3 trees [Aho et al. 1974; Knuth 1973].

In p.q.s elements have *fixed* priorities. What happens if we allow the priority of the elements to change? Obviously, an additional operation that 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 we have to specify all these changes.

We consider 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. decrease the priorities of all the *current* elements by some real number δ.

A version of p.q.$_1$ was used by Tarjan [1977].

To define p.q.$_2$, we first need some assumptions. We assume that the elements belong to a totally ordered set and that they are partitioned into groups. Every group can be either *active* or *nonactive*. An element is active if its group is active. By splitting a group according to an element e, we mean creating two groups from all the elements in the group greater (not greater) than e. 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 into 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;
8′. split a group according to an element in it.

It may seem 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 p.q.$_1$ and p.q.$_2$ efficiently. In particular, the change of priorities can be achieved implicitly in *one* step (see Galil et al. [1986]):

Theorem 3

p.q.$_1$ and p.q.$_2$ can be implemented in time $O(\log n)$ per operation.

The implementation of the generalized p.q.s uses regular p.q.s and a collection of offsets to handle the change in priorities.

We also make use of Johnson's *d-heap* [Johnson 1977], where d is the number of sons of internal nodes. (The usual heap is a 2-heap.) We partition the primitive operations into two types. Type 1 includes inserting an element and decreasing the priority of an element, and type 2 includes deleting an element. Type 1 involves "sifting up" the heap for a cost of $O(\log_d n)$, whereas type 2 involves "sifting down" the heap for a cost of $O(d \log_d n)$. Consequently, the following theorem holds.

Theorem 4

Let $d = \lceil m/n + 1 \rceil$. A d-heap supports m operations of type 1 and n operations of type 2 in time $O(m \log_d n)$.

6. PROBLEM 3: MAX WEIGHTED MATCHING IN BIPARTITE GRAPHS, OR A WARM-UP FOR PROBLEM 4

Introducing weights and maximizing the weight make the problem much harder. One can show that the following approach solves Problems 3 and 4. Start with the empty matching, and in each stage find an augmenting path with the maximal increase of the weight. One then can show that after k stages, we have a matching of

maximum weight among matchings of size k (e.g., see Tarjan [1983]). In the case of Problem 3, finding such an augmenting path is relatively simple, because it can be easily reduced to solving a single-source shortest path problem for graphs with nonnegative edge lengths. But finding such an augmenting path for general graphs is much harder, so we choose a completely different approach.

We use duality theory of linear progamming (specifically the primal–dual method) to derive the algorithm. We need linear programming for motivation only. Once we obtain the algorithm (for Problem 3 or 4), we prove its correctness by a one-line proof. Therefore, the description below is somewhat sketchy. (See Lawler [1976] or Papadimitriou and Steiglitz [1982] for more details.)

After defining the problem as a linear program, we consider the dual problem and then use complementary slackness to transform our optimization problem into a problem of solving a set of inequalities (constraints). A pair of feasible solutions for the primal and the dual problems are both optimal if, for every positive variable in the one, the corresponding inequality in the other is satisfied as equality.

In the case of Problem 3, defining the problem as a linear program is immediate. We describe it as an integer program and replace the integrality constraints $x_{ij} \in \{0, 1\}$ by $0 \le x_{ij}$. Since the matrix of constraints is unimodular, we must have an optimal integral solution.

We will have a primal solution, a matching M, and a dual solution, an assignment of dual variables u_i, u_j (corresponding to boys i and girls j). For convenience we define slacks π_{ij} for every edge (i, j): $\pi_{ij} = u_i + u_j - w_{ij}$. The inequalities $\pi_{ij} \ge 0$ are the constraints of the dual problem. (Whenever we mention π_{ij} below we always assume that $(i, j) \in E$.) By duality, M has a maximal weight if 6.0–6.2 hold:

6.0 For every i and j; u_i, u_j, $\pi_{ij} \ge 0$.
6.1 (i, j) is matched $\Rightarrow \pi_{ij} = 0$.
6.2 Boy i (or girl j) is single $\Rightarrow u_i = 0$ ($u_j = 0$).

The sufficiency of 6.0–6.2 for optimality can be proved directly as follows: Assume that M, u_i, u_j, π_{ij} satisfy 6.0–6.2 and let N be any matching. Then $\sum_{(i,j)\in N} w_{ij} \le \sum_{(i,j)\in N} (u_i + u_j - \pi_{ij}) \le \sum_i u_i + \sum_j u_j$ (by 6.0 and the fact that N is a matching), while $\sum_{(i,j)\in M} w_{ij} = \sum_i u_i + \sum_j u_j$ (by 6.1 and 6.2 and the fact that M is a matching). Consequently, M is a maximum weight matching.

So, we only have to find a matching M and an assignment of the dual variables that satisfy 6.0–6.2. We use the *primal–dual* method. This method starts with a simple solution, which violates some of the constraints. The solution is then modified in a way that guarantees that the number of violations is reduced. In our case we start with $M = \emptyset$, $u_i = \max_{k,l} w_{k,l}$ for boys and $u_j = 0$ for girls. The initial solution satisfies 6.0 and 6.1, and violates only 6.2. The algorithm makes changes that preserve 6.0 and 6.2 and reduce the number of violations of 6.2.

The algorithm consists of $O(n)$ *stages*. In each stage we look for an augmenting path, as in the simple algorithm for Problem 1, except that we use only edges with zero slack ($\pi_{ij} = 0$). If the search is successful, we augment the matching (i.e., change the primal solution) and start a new stage. This is progress because one single boy gets married.

If the search fails, we change the dual variables as follows. Let $\delta = \min(\delta_1, \delta_2)$, $\delta_1 = \min_{i:S\text{-boy}} u_i$, $\delta_2 = \min_{i:S\text{-boy},j:\text{free girl}} \pi_{ij}$. For an S-boy i we set $u_i \leftarrow u_i - \delta$, and for a T-girl j we set $u_j \leftarrow u_j + \delta$. It is easy to see that $\delta > 0$ and the change preserves 6.0 and 6.1. Also $\delta_1 = u_{i_0}$ for any single boy i_0 (u_i's of all boys had the same initial value, and in each change of the dual variables, all single boys had S label and their dual variables were decreased by the same δ). If $\delta = \delta_1$, then after the change 6.2 holds, and we are done. Otherwise, for each edge (i, j), i an S-boy and j a free girl, with $\pi_{ij} = \delta_2$ (there exists at least one), π_{ij} becomes zero, and we can continue the search. Since at least one girl gets a T label as a result, $\delta = \delta_2$ at most $O(n)$ times per stage.

The naive implementation of the algorithm above takes $O(mn^2)$ time. The most costly part is maintaining δ_2. For every free girl j, let $\pi_j = \min_{i:S\text{-boy}} \pi_{ij}$ and let $e_j = (i, j)$ be an edge with i an S-vertex and

$\pi_{ij} = \pi_j$. Then $\delta_2 = \min_{j:\text{free girl}} \pi_j$. Note that when we make a change of δ in the dual variables, π_j is reduced by δ and e_j does not change. Also, if $\delta = \delta_2 = \pi_{j_0}$, then the slack of e_{j_0} becomes 0 and it can be used for continuing the search. By maintaining π_j and e_j for all free girls j, an $O(n^3)$ implementation of the algorithm follows.

In a different implementation, we maintain the collection $C = \{(i, j) \mid \pi_j > 0, i \text{ an } S\text{-boy}, j \text{ a free girl}\}$ as p.q.$_1$, since all these π_{ij}'s are reduced by δ in a change in the dual variables. Whenever we scan an S-vertex i, we consider all edges (i, j), where j is a free vertex. Those edges with $\pi_{ij} > 0$ are inserted into the p.q.$_1$. Consequently, this implementation takes $O(mn \log n)$ time.

A small improvement is achieved if we maintain π_j and e_j (as above) for free girls j in a p.q.$_1$. Then δ_2 is the minimum of this p.q. One can see that the p.q. used here satisfies the conditions of Theorem 4, and, consequently, we get an $O(mn \log_{\lceil m/n+1 \rceil} n)$ time bound, which dominates the two bounds of $O(n^3)$ and $O(mn \log n)$.

A closer look at a stage reveals that an augmenting path is found using Dijkstra's algorithm for all shortest paths from a single source [Dijkstra 1959]. The source is a new vertex, which is connected to all single boys with new edges of length zero. The lengths of other edges are the π_{ij}'s at the beginning of the stage. The reduction of a stage to a shortest-path problem is well known [Gabow 1974]. In fact, each stage discovers the augmenting path that causes the largest increase in the weight of the matching. The various implementations of Dijkstra's algorithm are (1) the naive implementation: $O(n^2)$, (2) an implementation using p.q.s: $O(m \log n)$, and (3) an implementation using Theorem 4: $O(m \log_{\lceil m/n+1 \rceil} n)$. Hence, the corresponding time bounds for n stages follow immediately.

The main purpose of this section is to serve as a warm-up for the next section.

7. PROBLEM 4: MAX WEIGHTED MATCHING IN GENERAL GRAPHS

If we try to solve Problem 4 exactly as we solved Problem 3, we immediately run into problems. The linear program obtained by dropping the integrality constraints from the integer program for Problem 3 may have no integer optimal solution. Edmonds [1965b] found an ingenious way to remove this difficulty, which led to a polynomial-time algorithm for Problem 4. He added an exponential number of constraints of the following form. For every odd subset of vertices B,

$$\sum_{\substack{(i,j)\in E \\ i,j\in B}} x_{ij} \le \left\lfloor \frac{|B|}{2} \right\rfloor.$$

These new constraints must be satisfied by any matching, and, surprisingly, their addition guarantees an integer optimal solution. This fact follows from the correctness of the algorithm, which can be proved directly.

We now proceed as before. We have a primal solution, a matching M, and a dual solution, an assignment of dual variables u_i for every vertex i and z_k for every odd subset of vertices B_k. We now define slacks π_{ij} slightly differently: $\pi_{ij} = u_i + u_j - w_{ij} + \sum_{i,j\in B_k} z_k$. (Again $\pi_{ij} \ge 0$ are the constraints of the dual problem.) By duality, M has maximal weight if 7.0–7.3 hold:

7.0 For every i, j, and k, u_i, π_{ij}, $z_k \ge 0$.
7.1 (i, j) is matched $\Rightarrow \pi_{ij} = 0$.
7.2 i is single $\Rightarrow u_i = 0$.
7.3 $z_k > 0 \Rightarrow B_k$ is full ($r_k \equiv \lfloor |B_k|/2 \rfloor = |\{(i, j) \in M \mid i, j \in B_k\}|$).

In fact, as in Problem 3, we only need duality theory for motivation. The sufficiency of 7.0–7.3 for optimality can be proved directly: Assume that M, u_i, π_{ij}, z_k satisfy 7.0–7.3, and let N be any matching. Then $\sum_{(i,j)\in N} w_{ij} \le \sum_{(i,j)\in N} (u_i + u_j - \pi_{ij} + \sum_{(i,j)\in B_k} z_k) \le \sum_i u_i + \sum_k r_k z_k$ (by 7.0 and the fact that N is a matching), while $\sum_{(i,j)\in M} w_{ij} = \sum_i u_i + \sum_k r_k z_k$ (by 7.1–7.3 and the fact that M is a matching).

We can use 7.0–7.3 to derive a polynomial algorithm because we will have $z_k > 0$ only for blossoms or subblossoms, and their total number at any moment is $O(n)$. We maintain only z_k's that correspond to blossoms. Since we consider only π_{ij} for i, j not in the same blossom, $\pi_{ij} = u_i + u_j - w_{ij}$, as in Problem 3.

We again use the primal–dual method. We start with $M = \emptyset$, $u_i = (\max_{k,l} w_{k,l})/2$ and no z_k's (no blossoms). The initial solution violates only 7.2. The algorithm makes changes that preserve 7.0, 7.1, 7.3 and reduce the number of violations of 7.2.

As in Problem 3, the algorithm consists of $O(n)$ stages. In each stage we look for an augmenting path using the labeling R12 and the two cases C1, C2 as in the simple algorithm for Problem 2, except that we only use edges with $\pi_{ij} = 0$. If the search is successful, we augment the matching.

To preserve 7.3 we keep blossoms with $z_k > 0$ shrunk at the end of the stage. As a result, we have two new kinds of blossoms in addition to the S-blossoms we had in Problem 2. (Recall that a newly generated blossom is labeled by S.) Since the labels are erased at the end of a stage, we may have free blossoms at the beginning of a stage. During the search a free blossom can become a T-blossom. (Recall that a blossom is just a vertex in the current graph.) We call the vertices of an S-blossom (a T-blossom or a free blossom) S-vertices (T-vertices or free vertices). When, during the search, a new S-blossom B_k is formed, the vertices in its T-blossoms (which now become subblossoms) become S-vertices and are inserted in the queue Q (of S-vertices). We also initialize a new z_k to zero.

If the search is not successful, we make the following changes in the dual variables. We choose $\delta = \min(\delta_1, \delta_2, \delta_3, \delta_4)$, where

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

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

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

$$\delta_4 = \min_{B_k:T\text{-blossom}} z_k.$$

We then set

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

Such a choice of δ preserves 7.0, 7.1, and 7.3

If $\delta = \delta_4$, we expand all T-blossoms B_k on which the minimum was attained. (Each corresponding z_k becomes 0.) The expansion of blossom B is shown in Figure 2. B stops being a blossom and its subblossoms become blossoms. All vertices of the new S-blossoms are inserted into Q.

If $\delta = \delta_2$ ($\delta = \delta_3$), we consider 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. For each such edge π_{ij} becomes 0 and we can use it for continuing the search. At the end of each stage we also expand all S-blossoms B_k with $z_k = 0$.

Let us call each change in the dual variables a *substage*. Each S-blossom corresponds to a unique node in one of the structure trees at the end of the stage. Each T-blossom corresponds to a unique node in one of the structure trees at the beginning of the stage. Consequently, for $i = 2, 3, 4$, $\delta = \delta_i$ at most $O(n)$ times per stage: when $\delta = \delta_2$, a blossom becomes a T-blossom; when $\delta = \delta_3$, either the stage ends or a new S-blossom is generated, and when $\delta = \delta_4$, a T-blossom is expanded. Finally, $\delta = \delta_1$ at most once. Consequently, there are $O(n)$ substages per stage.

The most costly part of a substage is computing δ. The obvious way to compute it takes $O(m)$ steps and yields an $O(mn^2)$ algorithm. Edmonds' time bound was $O(n^4)$.

The only parts that require more than $O(n^3)$ are maintaining δ_2 and δ_3. δ_2 is handled as in the $O(n^3)$ algorithm for Problem 3. To take care of δ_3, we define for every pair of S-blossoms B_k, B_l:

$$\varphi_{k,l} = \min_{i \in B_k, j \in B_l} \left(\frac{\pi_{ij}}{2}\right).$$

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 $\delta_3 = \min_k \varphi_k$. A change in the dual variables and computing δ_3 costs $O(n^3)$ as for δ_2. We have to update $\{\varphi_k\}$ and $e_{k,l}$ any time an S-blossom B_k is constructed from $B_{i_1}, \ldots, B_{i_r}$. Recall that $(r + 1)/2$ of the subblossoms are S-blossoms and $(r - 1)/2$ of them are T-blossoms. We

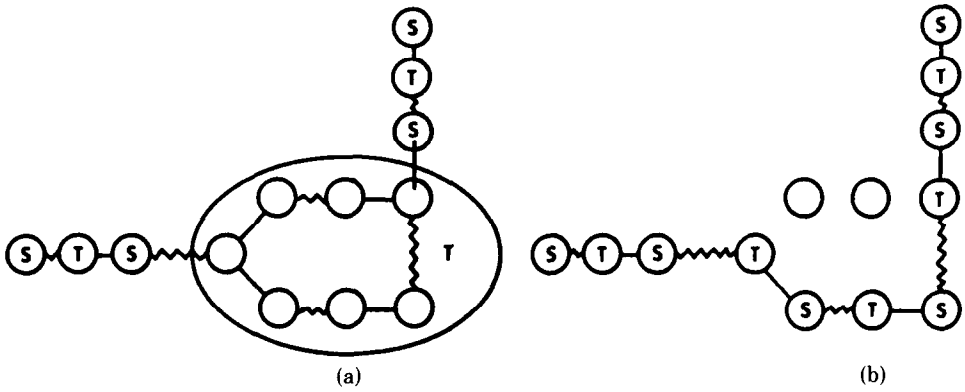

Figure 2. Expansion of a T-blossom (a) before and (b) after expansion.

first "make" each T-blossom B_p into an S-blossom by considering all its edges and computing for it $\{\varphi_{p,l}\}$ and $\{e_{p,l}\}$. Then we use the $\varphi_{p,l}$'s of $B_{i_1}, \ldots, B_{i_r}$ to compute φ_k and $\{e_{k,l}\}$ for the new blossom B_k and to update $\{\varphi_j\}$ for $j \neq k$. The total cost (per stage) to make T-blossoms into S-blossoms is $O(m)$. We now compute the rest of the cost $T(N)$, where N is the number of S-blossoms plus the number of non-S-vertices in the graph. $T(N) \leq crN + T(N - r + 1)$ because rN is a bound on the number of $\varphi_{k,l}$'s considered after making the T-blossoms into S-blossoms. $T(N) = O(N^2)$ (by induction on N), and the total cost of computing δ_3 is $O(n^3)$. The discussion above results in an $O(n^3)$ algorithm [Gabow 1974; Lawler 1976].

The most costly part of the algorithm is the frequent updates of the dual variables, which cause changes in $\{\pi_{ij}\}$. Note that all the elements that determine each δ_i are decreased by δ each change in the dual variables. We maintain δ_1, δ_3, δ_4 by a p.q.$_1$. We use a p.q.$_1$ to maintain u_i for T-vertices, and another p.q.$_1$ for z_k's of S-blossoms B_k.

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

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. 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 as a result of 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 $\{(i, j) \mid i \text{ an } S\text{-vertex}, j \in B\}$. The order on the elements is derived from the order on the vertices of B. The latter is taken to be the left-to-right order of the leaves of the structure tree. The order between two edges (i_1, j) and (i_2, j) is arbitrary. The order enables us to split the group g into the groups corresponding to $B_1, \ldots, B_r$ when we expand B to its subblossoms.

To maintain the generalized priority queues, we make a change in the scanning of a new S-vertex i. We also take into account edges (i, j) with $\pi_{ij} > 0$ and have three more cases in addition to C1 and C2 for edges (i, j) with $\pi_{ij} = 0$. Assume that j is in a blossom B with $\pi_{ij} > 0$.

(C3) B is a free blossom.
(C4) B is a T-blossom.

We insert (i, j) with priority π_{ij} into the

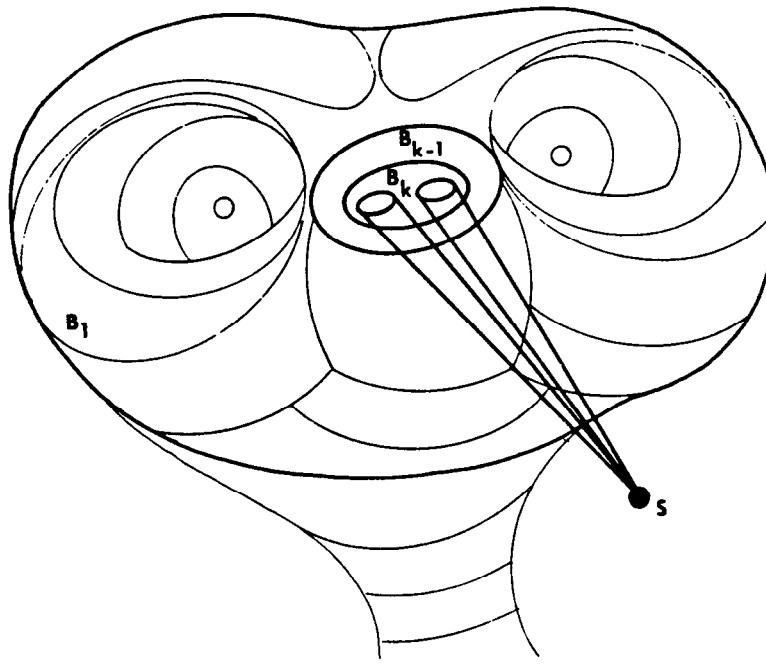

Figure 3. Edges from a single vertex to the innermost blossom that we may have to scan again and again if the blossoms $B_1, \ldots, B_k$ are eventually expanded.

active (nonactive) group corresponding to B.

(C5) B is an S-blossom and $i \notin B$.

We insert (i, j) with priority $\pi_{ij}/2$ to the p.q.$_1$ that computes δ_3.

Remark 1

Since $\delta_1 = u_{i_0}$ for any single vertex i_0, we do not need a generalized p.q. to compute δ_1. Nevertheless, we have a p.q.$_1$ for the u_i's of the S-vertices and also a p.q.$_1$ for the u_i's of the T-vertices for computing π_{ij} when the edge (i, j) is considered.

Remark 2

We have a p.q.$_1$ for the z_k's for S-blossoms, because at the end of a stage they all become free, and in the next stage they may become T-blossoms.

Remark 3

The p.q.$_1$ for computing δ_3 contains also edges (i, j) with i and j in the same blossom. We do not have time to locate and delete such edges each time a new blossom is constructed. Consequently, if $\delta = \delta_3$ and $\delta_3 = \pi_{ij}$, we first check whether i and j are in the same blossom. If they are, we delete the edge and possibly compute a new (larger) δ.

Remark 4

All edges (i, j) in the generalized p.q.'s that compute δ_2 or δ_3 have $\pi_{ij} > 0$, since an element is deleted as soon as its priority becomes 0. Similarly, all z_k's in the p.q.$_1$ that computes δ_4 are positive. Consequently, $\delta > 0$.

To derive an $O(mn \log n)$ time bound, we need to implement two parts of an algorithm carefully:

1. We maintain the sets of vertices in each blossom (for finding the blossom that contains a given vertex) by concatenable queues [Aho et al. 1974]. Note that the number of finds, concatenates and splits is $O(m)$ per stage, and each takes $O(\log n)$ time.

2. In C2 we use the careful backtracking described for Problem 2.

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

8. CONCLUSION

We have considered four versions of the maximum matching problem and discussed the development of the most efficient algorithms for solving them. By "most efficient algorithms" we mean those that have the smallest asymptotic running times. We now mention briefly a number of closely related additional topics and give some references. These are intended to serve as examples and certainly do not form an exhaustive list.

(*a*) *Applications of Matching.* We do not list here the many applications of solutions to Problems 1–4. For some applications see Lawler [1976].

(*b*) *Generalization of Matching.* Problems 1–4 can be generalized in a number of ways. For example, Gabow [1983a] has recently considered similar problems where some kinds of polygamy are allowed. He found efficient reductions to the corresponding matching problem. Stockmeyer and Vazirani [1982] showed that several natural generalizations of matching are NP-complete.

(*c*) *Special Cases of Matching.* Many applications solve one of the Problems 1–4, but only for special graphs. For example, Problem 1 is used to find routing in superconcentrators [Gabber and Galil 1981]. The graphs that arise in this application have vertices with bounded degree, and hence the solution given here takes time $O(n^{1.5})$. Perhaps this can be improved. For better algorithms for some special cases of Problem 1, see Cole and Hopcroft [1982] and Gabow [1976b].

(*d*) *Randomizing Algorithms.* Several algorithms that work very well for random graphs or for most graphs have been developed. They are usually faster and simpler than the algorithms discussed here [Angluin and Valiant 1979; Karp 1980]. An interesting problem is to find improved randomizing algorithms that use random choices (rather than random inputs).

(*e*) *Approximation Algorithms.* As for all optimization problems, we may settle for approximate solutions. For cardinality matching, the solution that uses phases yields a good approximation by executing only a constant number of phases. For simple, fast, and good approximation algorithms for special graphs see Iri et al. [1981], Karp and Sipser [1981], and Plaisted [1984].

(*f*) *Improvements.* We next discuss possible improvements of the algorithms considered in this paper. All the time bounds discussed in this paper can be shown to be tight. One can construct families of inputs for which the algorithms require the number of steps that is specified by the stated upper bounds. There are no known lower bounds for any of the four problems. Improving the $O(m\sqrt{n})$ bound for cardinality matching must involve the discovery of a new approach that does not use phases. Similarly, except for a logarithmic factor, improving the bound for weighted matching requires the use of an approach that does not make $\Theta(n)$ augmentations. Perhaps the introduction of phases may lead to improved algorithms for Problems 3 and 4. Note that the solution to Problem 3 is slightly better than the solution to Problem 4, due to the use of d-heaps. It may still be possible to find a similar improved solution for Problem 4.

There are several theoretical questions concerning Problems 1–4. Their solution may lead to simpler or faster algorithms:

- Can we find efficient solutions to any of the problems without augmenting paths?
- Are blossoms necessary?
- Can we solve Problem 4 without duality?

Assume that we have solved an instance of a weighted matching problem and then make a small change such as adding or deleting some edges or changing the weight of a few edges. It is not clear how to make use of the solution to the original problem.

It seems that, using the algorithms described here, we may have to spend $O(mn \log n)$ time to find the new solution. For some partial solution to these types of questions, see Derigs [1981, 1982] and Weber [1981].

Finally, we briefly consider parallel algorithms:

- Can we solve any one of the four problems in time $O(\log^k n)$ with a polynomial number of processors?
- Is Problem 4 log-space complete for P (the class of problems solvable in polynomial time)?

A positive answer to the latter implies that a positive answer to the former (regarding Problem 4) is unlikely. Recently, the problem of network flow has been shown to be log-space complete for P by Goldschlager et al. [1982]. As was observed by Borodin et al. [1982], there is a nonuniform algorithm that computes the size of the maximum matching in time $O(\log^2 n)$ with a polynomial number of processors. It is not clear how to use it in order to find a similar algorithm that finds a maximum matching.

9. VERY RECENT PROGRESS

Since this paper was first written, there have been a number of results related to the topics raised in the last section.

Ball and Derigs [1983] consider an alternative approach to Problem 4, which also uses duality. It was implemented in time $O(n^3)$ and $O(mn \log n)$ using the ideas of Galil et al. [1986]. Stages are interpreted as searches for shortest paths. In one of the variants we successively add a new vertex with its edges in each stage.

The solutions for Problems 3 and 4 were slightly improved. A new data structure, called Fibonacci heap (or F-heap), was introduced by Fredman and Tarjan [1984]. It supports most operations including inserting, merging, and decreasing the key of an element in $O(1)$ amortized time. (Operations are associated with amortized time so that the total time is bounded above by the total amortized time.) Deletion is the only expensive operation, costing $O(\log n)$ amortized time, where n is the total number of elements. Using F-heaps immediately improves Dijkstra's algorithm, as well as one stage of Problem 3, to $O(m + n \log n)$. Consequently, the best time bound for Problem 3 is currently $O(n(m + n \log n))$. For a different algorithm that uses F-heaps and yields the same time bound, and for references to other algorithms for Problem 3, see Goldfarb [1985].

Even with F-heaps, it was not clear how to improve the best algorithms for Problem 4. There were two difficulties. There was a problem with splits, since F-heaps do not support splits. There was also a problem with the edges (i, j), where i, j are S-vertices in the same blossom, because such edges have to be deleted and deletion is relatively expensive. New developments in data structures were used to overcome these difficulties [Gabow et al. 1984]. Consequently, the current best algorithm for Problem 4 takes time

$$O(n(m \log \log \log_{\lceil m/n+1 \rceil} n + n \log n)).$$

This bound is better than

$$O(mn \log_{\lceil m/n+1 \rceil} n),$$

but worse than the best bound currently known for Problem 3. The algorithm still uses the observations of Section 6.

The question of the possibility of using phases for Problems 3 and 4 was partially answered by Gabow [1983b, 1985]. He considered the case in which weights are integers bounded above by N. By using scaling techniques, he was able to use algorithms for Problems 1 and 2 for solving Problems 3 and 4. As a result, he obtained an $O(n^{3/4} m \log N)$ algorithm for Problem 3 and a similar time bound for Problem 4.

A new simple algorithm has recently been designed for Problem 2 [Rabin and Vazirani 1984]. This algorithm is related to three questions raised in Section 9. It is a randomizing algorithm that uses neither blossoms nor augmenting paths. The algorithm consists of up to n stages in which an $n \times n$ matrix is inverted. Using the asymptotically best algorithm for inverting matrices of Coppersmith and Winograd [1982], an $O(n^{3.5})$ expected time bound follows. (A more realistic bound is $O(n^4)$.) It is still a challenge to find simple algorithms that may use randomization but will also improve the time bound for Problem 2.

Note that no simple approach is yet known for Problem 4.

As for parallel algorithms, a new randomizing algorithm solves Problem 2 (and Problem 1) in expected time $O(\log^3 n)$ with $O(n^{7.5})$ processors [Karp et al. 1985]. So it is now possible to efficiently find a maximum matching, rather than just its size. More recently, this algorithm was improved [Galil and Pan 1985]. The new algorithm has the same time complexity, but requires only $O(n^{3.5})$ processors. Even more recently, a simple randomized algorithm for Problem 2 was discovered, which improved the expected running time to $O(\log^2 n)$ and requires $O(mn^{3.5})$ processors [Mulmuley et al. 1985]. All the parallel algorithms use the algebraic approach of Rabin and Vazirani [1984], which uses no augmenting paths. The remaining challenges are to improve the time and processor bounds (or to achieve the best time and processor bounds simultaneously), and to find a deterministic algorithm with similar time and processor complexities. The new parallel algorithms also yield good solutions for the special cases of Problems 3 and 4, in which the weights are given in unary. However, the status of (the general versions of) Problems 3 and 4 is still open.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation Grants MCS-830-3139 and DCR-85-11713.

I would like to thank Dannie Durand, Hal Gabow, and Stuart Haber for their helpful comments, Kerny McLaughlin for her help with the figures, and Bella Galil for preparing Figure 3.

REFERENCES

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

ANGLUIN, D., AND VALIANT, L. G. 1979. Fast probabilistic algorithms for Hamiltonian paths and matchings. *J. Comput. Syst. Sci. 18*, 144–193.

BALL, M. O., AND DERIGS, U. 1983. An analysis of alternative strategies for implementing matching algorithms. *Network 13*, 517–549.

BERGE, C. 1957. Two theorems in graph theory. *Proc. Nat. Acad. Sci. 43*, 842–844.

BORODIN, A., VON ZUR GATHEN, J., AND HOPCROFT, J. E. 1982. Fast parallel and gcd computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 64–71.

COLE, R., AND HOPCROFT, J. E. 1982. On edge coloring bipartite graphs. *SIAM J. Comput. 11*, 540–546.

COPPERSMITH, D., AND WINOGRAD, S. 1982. On the asymptotic complexity of matrix multiplication. *SIAM J. Comput. 11*, 472–492.

DERIGS, U. 1981. A shortest augmenting path method for solving minimal perfect matching problems. *Networks 11*, 379–390.

DERIGS, U. 1982. Shortest augmenting paths and sensitivity analysis for optimal matchings. Rep. 82222-OR, Institut für Okonometrie und Operations Research, Univ. Bonn, West Germany, April.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 263–271.

DINIC, E. A. 1970. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Sov. Math. Dokl. 11*, 1277–1280.

EDMONDS, J. 1965a. Path, trees and flowers. *Can. J. Math. 17*, 449–467.

EDMONDS, J. 1965b. Matching and a polyhedron with 0,1 vertices. *J. Res. N. B. S. B, 69* (April–June), 125–130.

EVEN, S., AND KARIV, O. 1975. An $O(n^{2.5})$ algorithm for maximum matching in graphs. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 100–112.

EVEN, S., AND TARJAN, R. E. 1975. Network flow and testing graph connectivity. *SIAM J. Comput 4*, 507–518.

FORD, L. R., AND FULKERSON, D. R. 1956. Maximal flow through a network. *Can. J. Math 8*, 399–404.

FREDMAN, M. L., AND TARJAN, R. E. 1984. Fibonacci heaps and their uses (in improved network optimization algorithms). In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 338–346.

GABBER, O., AND GALIL, Z. 1981. Explicit construction of linear-sized superconcentrators. *J. Comput. Syst. Sci. 22*, 407–420.

GABOW, H. N. 1974. Implementation of algorithms for maximum matching on nonbipartite graphs. Ph.D. dissertation, Dept. of Computer Science, Stanford Univ., Stanford, Calif.

GABOW, H. N. 1976a. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. ACM 23*, 2 (Apr.), 221–234.

GABOW, H. N. 1976b. Using Euler partitions to edge color bipartite multigraphs. *Int. J. Comput. Inf. Sci. 5*, 344–355.

GABOW, H. N. 1983a. An efficient reduction technique for degree-constrained subgraphs and bidirected network flow problems. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (Boston, Apr. 25–27). ACM, New York, pp. 448–456.

GABOW, H. N. 1983b. Scaling algorithms for network problems. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 248–257.

GABOW, H. N. 1985. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 90–100.

GABOW, H. N., AND TARJAN. R. E. 1983. A linear time algorithm for a special case of disjoint set union. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (Boston, Apr. 25–27). ACM, New York, pp. 246–251.

GABOW, H. N., GALIL, Z., AND SPENCER, T. H. 1984. Efficient implementation of graph algorithms using contraction. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 347–357.

GALIL, Z. 1980. An $O(E^{2/3}V^{5/3})$ algorithm for the maximal flow problem. *Acta Inf. 14*, 221–242.

GALIL, Z., AND PAN, V. 1985. Improved processor bounds for algebraic and combinatorial problems in RNC. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 490–495.

GALIL, Z., MICALI, S., AND GABOW, H. N. 1986. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput. 15*, 120–130.

GOLDFARB, D. 1985. Efficient dual simplex algorithms for the assignment problem. *Math. Program. 33*, 187–203.

GOLDSCHLAGER, L., SHAW, R., AND STAPLES, J. 1982. The maximum flow problem is log space complete for P. *Theor. Comput. Sci. 21*, 105–111.

HOPCROFT, J. E., AND KARP, R. M. 1973. $N^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput. 2*, 225–231.

IRI, M., MUROTA, K., AND MATSUI, S. 1981. Linear time approximation algorithms for finding the minimum weight perfect matching on a plan. *Inf. Process. Lett. 12*, 206–209.

JOHNSON, D. 1977. Efficient algorithms for shortest paths in sparse networks. *J. ACM 24*, 1 (Jan.), 1–13.

KARIV, O. 1976. An $O(n^{2.5})$ algorithm for maximal matching in general graphs. Ph.D. dissertation, Dept. of Applied Mathematics, The Weizmann Institute, Rehovot, Israel.

KARP, R. M. 1980. An algorithm to solve the assignment problem in expected time $O(mn \log n)$. *Network 10*, 143–152.

KARP, R. M., AND SIPSER, M. 1981. Maximal matchings in sparse graphs. In *Proceedings of the 22nd IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 364–375.

KARP, R. M., UPFAL, E., AND WIGDERSON, A. 1985. Constructing a perfect matching is in random NC. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* (Providence, R.I., May 6–8). ACM, New York, pp. 22–32.

KNUTH, D. E. 1973. *The Art of Computer Programming.* Vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass.

KUHN, H. W. 1955. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart 2*, 253–258.

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

MICALI, S., AND VAZIRANI, V. V. 1980. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximal matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 17–27.

MULMULEY, K., VAZIRANI, U. V., AND VAZIRANI, V. V. 1985. Matching is as easy as matrix inversion. Manuscript, MSRI Berkeley, Berkeley, Calif.

NORMAN, R. Z., AND RABIN, M. O. 1959. An algorithm for a minimum cover of a graph. *Proc. Am. Math. Soc. 10*, 315–319.

PAPADIMITRIOU, C. H., AND STEIGLITZ, K. 1982. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, N.J.

PLAISTED, D. A. 1984. Heuristic matching for graphs satisfying the triangle inequality. *J. Algorithms 5*, 163–179.

RABIN, M. O., AND VAZIRANI, V. V. 1984. Maximum matchings in general graphs through randomization. Rep. TR-15-84, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass., Oct.

SLISENKO, A. O. 1973. Recognition of palindromes by multihead Turing machines. In *Problems in the Constructive Trend in Mathematics. VI.* In *Proceedings of the Steklov Institute of Mathematics* , vol. 129, V. P. Orevkov and N. A. Sanin, Eds. Academy of Sciences of the U.S.S.R., pp. 30–202; R. H. Silverman, *Trans. Am. Math. Soc.* (1976), 25–208.

STOCKMEYER, L. J., AND VAZIRANI, V. V. 1982. NP-completeness of some generalizations of the maximum matching problem. *Inf. Process. Lett. 15*, 11–19.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (Apr.), 215–225.

TARJAN, R. E. 1977. Finding optimum branchings. *Network 7*, 25–35.

TARJAN, R. E. 1983. Data structures and network algorithms. *SIAM, publications.*

WEBER, G. 1981. Sensitivity analysis of optimal matchings. *Networks 11*, 41–56.

BIBLIOGRAPHY

KAMEDA, T., AND MUNRO, I. 1974. $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing 12*, 91–98.

Received October 1983; final revision accepted May 1986