CUCS-48-83

Efficient Algorithms for Finding Maximal Matching in Graphs

Zvi Galil
Columbia University and
Tel-Aviv University

February 1983

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

Key words: Shmathematics, algorithmic tools, data structures monsters, matching, polygamy, the asexual case, the assignment problem, moonlighting, augmenting path, ET, blossoms, shrink, The Main Theorem of Botany, The ACM Longest Paper Award, generalized priority queue, d-heap, warm-up, duality, primal-dual, sexual discrimination, affirmative action, joint income tax return.

1. 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 the design.

1. Shmathematics.

Initially we use some mathematical arguments to characterize the solution. The Mathematics used is usually quite simple ('sh' for shallow). This leads to a simple algorithm that is usually not very efficient.

2. 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 [AHU]. Alternatively, we may try to find a way to reduce the number of steps in the original algorithm possibly by finding a better way to organize the information.

3. Data structures and monsters.

Sometimes we can speed up our algorithm by using an efficient data structure that supports the primitive operations that the algorithm uses. We may even resort to the introduction of monsters. These are very complicated data structures that bring about some asymptotic speed up, that usually becomes meaningful only for very large problem size. (For a real-life monster see [Ga].)

In these three stages we sometimes use a known technique: a certain result in Mathematics, say, or a known algorithmic tool or date structure. In the more interesting problems we need to invent new techniques, or to refine existing ones for our purposes. We may need to prove new Shmathematics; to find an appropriate way to organize information, or even how to use a known algorithmic tool; or to invent a new data structure or make some observations concerning known data structures that make them useful for our purposes.

A word of caution about Shmathematics. Indeed, 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.

In most cases we use all three stages above in this order, 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 maximal cardinality or weighted matching in general or bipartite graphs. Mathematically, these are all special cases of the case of weighted matching in general graphs. We, however, will consider them in increasing order of difficulty because the easier the problem, the faster or the simpler its solution.

2. 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; i.e., 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 maximal 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 boys and girls, but each edge (i,j) has a nonnegative weight w_{ij} associated with it. Our goal is to find a matching with the maximal 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 them, or can be solved by using, in turn, the solutions to these problems as subroutines.

3. An augmenting path

An important notion for all four problems is that of an augmenting path. We will solve each one of them in stages, and in each stage we will 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 (w.r.t. M) is a simple path, such that every other edge on it is matched. An augmenting path (w.r.t. M) is an alternating path between two singles. It must be of odd length, and in the bipartite case its two endpoints must be of different sex. The following Theorem is due to Berge (see [L]).

Theorem 1: The matching M has maximal cardinality iff there is no augmenting path w.r.t. 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 conversely) we increase the size of M by 1. We call this operation augmenting the matching M. The other part is not trivial, but is quite easy. While it is immediately clear why the notion of augmenting path is important for cardinality matching, it is surprising that it is also important when we maximize weight (problems 3 and 4).

4. Problem 1.

Theorem 1 gives an immediate algorithm. It consists of O(n) stages. In each stage a search for an augmenting path is conducted. If there exists an augmenting path, 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. After cleaning all labels from previous stages all single boys are labeled with S. We then apply two labeling rules.

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

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

The label contains also the vertex from which the label has arrived. (In the case of an S label this information is redundant.) The search continues until either the search succeeds or it fails. The search succeeds if a free 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 the converse is also true.

By Lemma 1, if the search fails, then 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 best algorithm for Problem 1 is by Hopcroft and Karp [HK]. They found a way how 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 is 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 shortest augmenting paths. This subgraph is layered. In layer number 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 cases. In case 1, a single girl is reached and we complete the last layer and delete the nonsingle girls from it. In case 2, we cannot continue. In this case the algorithm (not only the phase) terminates. This is justified by Lemma 1.

In $\tilde{G}$ we find a maximal set of disjoint augmenting paths using depth-first-search. Each time we reach a single girl we find an augmenting path, erase its edges from $\tilde{G}$ and start from another single boy. Each 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 [HK].

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 [FF] network flow algorithm for these special networks. Similarly, Hopcroft and Karp's algorithm is actually Dinic's algorithm [Di] applied to these special networds. This was first observed in [ET].

5. Problem 2.

As for Problem 1, Theorem 1 suggests a possible algorithm of O(n) stages. In each stage we look for an augmenting path. 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 with S the spouse of j. We call this rule (R12).

The search is conducted by scanning in turn the S-vertices. 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.

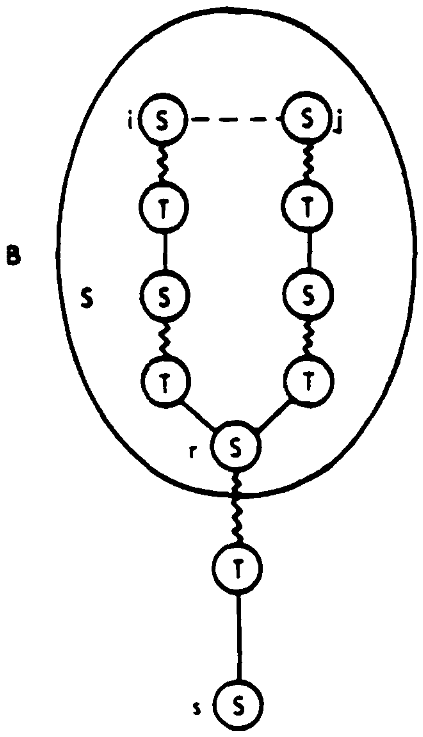
In case (C1) we apply (R12). In case (C2) we do the following: we backtrack from i and j, using the labels, to the single persons s_i and s_j from which i and j got their S labels. If $s_i \neq s_j$ we found an augmenting path from s_i to s_j and augment the matching. The trouble begins (or, life starts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

We next describe Edmonds' remarkable work in Botany, where 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 parts 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 <u>blossom</u> and r its <u>base</u>. (See Fig. 1.)

Edmonds' idea was to <u>shrink</u> B: replace it by a single supervertex B and replace the set A of edges incident with vertices of B by the set $A^1 = \{(B,j) | j \notin B, \exists (i,j) \in A\}$. At most one member of A^1 (incident with r) is matched. (There are none iff r = s.) If $\hat{G}$ is the graph obtained from G after such a shrinking, then the shrinking is justified by The Main Theorem of Botany:

<u>Theorem 2</u>. There is an augmenting path in G iff there is an augmenting path in $\hat{G}$.



<u>Figure 1</u>. A blossom.

We do not know of any easy proof for Theorem 2. (See [L].) One 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: we replace the matched edge, say (B,k), with (r,k); we replace the unmatched edge, say (j,B), with the edge (j,i) which it originated from 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 around the blossom. Storing B as a doubly linked list with a marked base makes this very easy.

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. So a blossom is a recursive structure because we may shrink many times. It will be 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, at any time each vertex in the _original graph_ G belongs to one blossom in the _current graph_. For each blossom B, the collection of the doubly linked lists form a tree which we call the _structure tree_ of B. Its leaves are the vertices that belong to B.

If the search succeeds (in (C2)) we find the augmenting path in the current graph. Then we use the easy part of Theorem 2 sketched above 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. All this takes O(m) 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 maximal and we are done.

A naive implementation [E1] takes $O(n^4)$ ($O(n^3)$ per stage). A more careful implementation takes $O(n^3)$ [G1]: 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 name. In order to find out quickly which blossom a given vertex belongs to, 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 have already been inserted into 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 $O(n)$. Using the 'set union' algorithm to manipulate the sets of vertices in the blossoms for the membership tests takes $O(n\alpha(m,n))$ per stage for a total of $O(mn\alpha(m,n))$, where α is the inverse of Ackermann's function [T1].

The obvious question that comes to mind is whether the idea 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.

In [EK] the authors 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 [K] is a strong contender for the ACM Longest Paper Award. (It will probably lose only to Slisenko's real-time palindrome recognizer [S].)

A simpler approach was formed more recently [MV]. As in the bipartite case, a phase consists of two parts: (1) identifying the subgraph $\tilde{G}$ of G that contains all shortest augmenting paths; and (2) finding in $\tilde{G}$ a maximal set of disjoint autmenting paths of shortest length. Both parts are more complicated than in the bipartite case because of the existence of blossoms. We do not give the details here. The immediate implementation of the algorithm described in [MV] 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, and as a result their algorithm runs in time $O(m\sqrt{n})$. Quite recently, a linear-time algorithm for some special cases of the disjoint set union was found [GT]. One of these special cases is the one needed in Problem 2.

6. Some Observations on Data Structures.

The most efficient algorithms for Problem 3 and Problem 4 use some observations on data structure 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; 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 implementation of p.q.'s are known; e.g., 2-3 trees ([AHU],[Kn]).

In p.q.'s elements have fixed priorities. We consider 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 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 in [T2].

To define p.q.$_2$ we first need some assumptions. We assume that the elements are partitioned into groups. Every group can be either <u>active</u> or <u>nonactive</u>. An element is active if its group is active. Assume that the elements are totally ordered. 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 to one of the groups;

(2)' delete an element;

(3)' find an <u>active</u> element with the minimal priority;

(4)' decrease the priorities of all the <u>active</u> elements by some real number δ;

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

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

(7)' split a group according to an element in it.

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 <u>one</u> operation [GMG]:

<u>Theorem 3</u>. p.q.$_1$ and p.q.$_2$ can be implemented in time O(log n) per operation.

We will also make use of Johnson's <u>d-heap</u> [J]. The d refers to 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 and increasing the priority of an element. Type 1 involves 'sifting up' the heap for a cost of $\log_d n$ while type 2 involves sifting down the heap for a cost of $d \log_d n$. Consequently, the following theorem holds.

Theorem 4. Let $\ell = \lceil m/n \rceil + 1$. An ℓ-heap supports m operations of type 1 and n operations of type 2 in time $O(m \log_\ell n)$.

7. Problem 3 or A Warm-up for Problem 4.

We use duality theory of linear programming. We define the problem as a linear program. We then 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 iff 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 $0 \leq x_{ij}$. 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): $\pi_{ij} = u_i + u_j - w_{ij}$. $\pi_{ij} \geq 0$ are the constraints of the dual problem. (Whenever we mention π_{ij} below we always assume that $(i,j) \in E$.) By duality (see [L] for details), M has a maximal weight if (3.0)-(3.2) hold. (This fact can be proven directly by a one line proof.)

(3.0) For every i,j, $u_i, u_j, \pi_{ij} \geq 0$.

(3.1) (i,j) is matched $\Rightarrow \pi_{ij} = 0$.

(3.2) Boy i is single $\Rightarrow u_i = 0$.

So, we only have to find a matching M and an assignment of the dual variables that satisfy (3.0)-(3.2). We use the primal-dual method. The 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 $M = \emptyset$, $u_i = \max_{k,\ell} w_{k,\ell}$ for boys and $u_j = 0$ for girls (a typical case of sexual discrimination). The initial solution satisfies (3.0), (3.1) but violates (3.2) (single boys have positive dual variables). The algorithm makes changes that preserve (3.0), (3.1) and reduce the number of violations of (3.2).

The algorithm consists of $O(n)$ <u>stages</u>. 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 (and can file a joint income tax return).

If the search fails we change the dual variables as follows. Let $\delta = \min(\delta_1, \delta_2)$, $\delta_1 = \min_{i:\text{S-boy}} u_i$, $\delta_2 = \min_{\substack{i:\ \text{S-boy} \\ j:\ \text{free girl}}} \pi_{ij}$.

For an S-boy i we set $u_i \leftarrow u_i - \delta$, and for a T-girl j we set $u_j \leftarrow u_j + \delta$ (affirmative action). It is easy to see that $\delta > 0$ and the change preserves (3.0), (3.1). Also $\delta_1 = u_{i_0}$ for any single boy i_0. If $\delta = \delta_1$, then after the change (3.2) holds and we are done. Otherwise, for each edge (i,j) 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 $\pi_j = \min_{i:\ \text{S-boy}} \pi_{ij}$ and let

$E_j = \{(i,j) | i$ is 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 the E_j's do not change. Also, if $\delta = \delta_2 = \pi_{j_0}$, then the slacks of the edges in E_{j_0} become 0 and they all 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 $C = \{(i,j) | \pi_j > 0$, i an S-boy, j a free girl$\}$ as a p.q.$_1$, since all these π_{ij}'s are reduced by δ. Whenever we scan an S-vertex i we consider <u>all</u> edges (i,j) with j 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 C as a p.q.$_2$. (We do not need here the split operation and nonactive groups never become active.) For every girl j we have the group $C_j = \{(i,j) | \pi_{ij} > 0$, i an S-boy$\}$. The group is active if j is free. One can see that the p.q.'s used here satisfy the conditions of Theorem 4, and consequently we get the best time bound for Problem 3: $O(mn \log_{\lceil m/n+1 \rceil} n)$.

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 [D]. The source is a new vertex which is connected to all single boys with new edges of length zero. The lengths of the other edges are the slacks at the beginning of the stage. The reduction of a stage to a shortest path problem is well known [Gl]. The various implementations of Dijkstra's algorithm are (1) the naive implementation $O(n^2)$, (2) using p.q.'s $O(m \log n)$, and (3) using Theorem 4

$O(m \log_{\lceil m/n+1 \rceil} n)$. Hence, the corresponding time bounds for n stages immediately follow. The main purpose of this section was to serve as a warm-up for the next section.

8. Problem 4.

If we try to solve Problem 4 exactly as we solved Problem 3, we immediately face difficulty. The linear program obtained by dropping the integrality constraints from the integer program may have no integer optimal solution. Edmonds [E2] 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 form

$$\sum_{\substack{(i,j)\in E \\ i,j\in B}} x_{ij} \le \lfloor |B|/2 \rfloor \text{ for every odd subset of the vertices } B.$$

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 proven directly.

We now proceed as before. We will 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 (4.0)-(4.3) hold:

(4.0) For every i,j and k, $u_i, \pi_{ij}, z_k \ge 0$,

(4.1) (i,j) is matched $\Rightarrow \pi_{ij} = 0$,

(4.2) i is single $\Rightarrow u_i = 0$,

(4.3) $z_k > 0 \Rightarrow B_k$ is full $\left(\left|\{(i,j)\in M \mid i,j\in B_k\}\right| = \lfloor |B_k|/2 \rfloor\right)$.

In fact, as in Problem 3, we need duality for motivation only. The sufficiency of (4.0)-(4.3) for optimality can be proven directly by a one line proof [GMG].

We can use (4.0)-(4.3) 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). Since we will 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 $M = \emptyset$, $u_i = (\max_{k,\ell} w_{k,\ell})/2$ and $z_k = 0$ (no blossoms). The initial solution violates only (4.2). The algorithm makes changes that preserve (4.0), (4.1), (4.3) and reduce the number of violations of (4.2).

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 use only edges with zero slack. If the search is successful we augment the matching.

To preserve (4.3) 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 the 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 to the queue Q. 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 $\delta_1 = \min_{i:\ \text{S-vertex}} u_j$, $\delta_2 = \min_{\substack{i:\ \text{S-vertex} \\ j:\ \text{free vertex}}} \pi_{ij}$,

$\delta_3 = \min_{\substack{i,j:\text{ S-vertices}\\ \text{not in the same blossom}}}(\pi_{ij}/2)$, $\delta_4 = \min_{B_k \text{ a T-blossom}}(z_k/2)$. 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; and

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

Such a choice of δ preserves (4.0), (4.1) and (4.3).

If $\delta = \delta_1$, then after the change (4.2) 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.

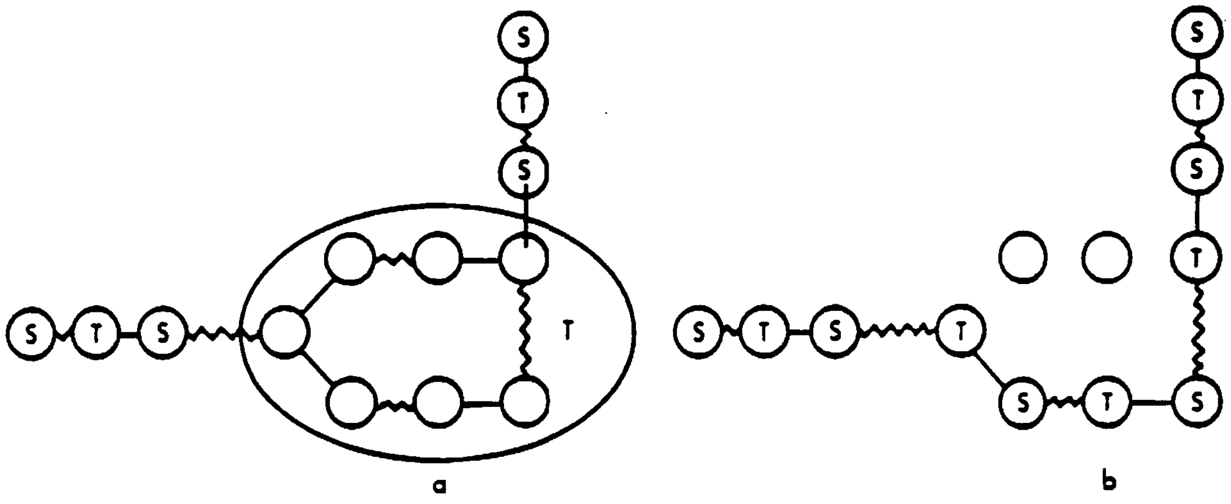


Figure 2. Expanding a T blossom: a) before and b) after the expansion.

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 an edge π_{ij} becomes 0 and we can use it for the search.

At the end of each stage we also expand all S-blossoms B_k with $z_k = 0$.

Let us call a substage each change in the dual variables. 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. Consequently, $\delta = \delta_3$ ($\delta = \delta_4$) at most O(n) per stage. $\delta = \delta_2$ at most O(n) times per stage, since when $\delta = \delta_2$ a blossom becomes a T-blossom. Finally, $\delta = \delta_1$ at most once. Consequently, there are O(n) substages per stage.

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

The only parts which 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 $\varphi_{k,\ell} = \min_{\substack{i \in B_k \\ j \in B_\ell}} (\pi_{ij}/2)$. We record the edge $e_{k,\ell}$ on which the minimum is attained and maintain $\varphi_k = \min_\ell \varphi_{k,\ell}$. We do not maintain $\varphi_{k,\ell}$, but any time we need it we compute it by using $e_{k,\ell}$. 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 $\{e_{k,\ell}\}$ 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 first "make" each T-blossom B_m an S-blossom by considering all its edges and computing for it $\{\varphi_{m,\ell}\}$ and $\{e_{m,\ell}\}$. Then we use the

$\varphi_{m,\ell}$'s of $B_{i_1},\ldots,B_{i_r}$ to compute φ_k, $\{e_{k,\ell}\}$ 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 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 $\varphi_{k,\ell}$'s considered after making the T-blossoms S-blossoms. $T(n) = O(n^2)$, and the total cost of computing δ_3 is $O(n^3)$. The discussion above results in an $O(n^3)$ algorithm [G1], [L].

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 also have one p.q.$_1$ to maintain u_i for T-vertices, and another p.q.$_1$ for z_k for S-blossoms B_k.

If we try to maintain δ_2 by a p.q.$_1$, we have 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 should not be considered for finding the value of 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).

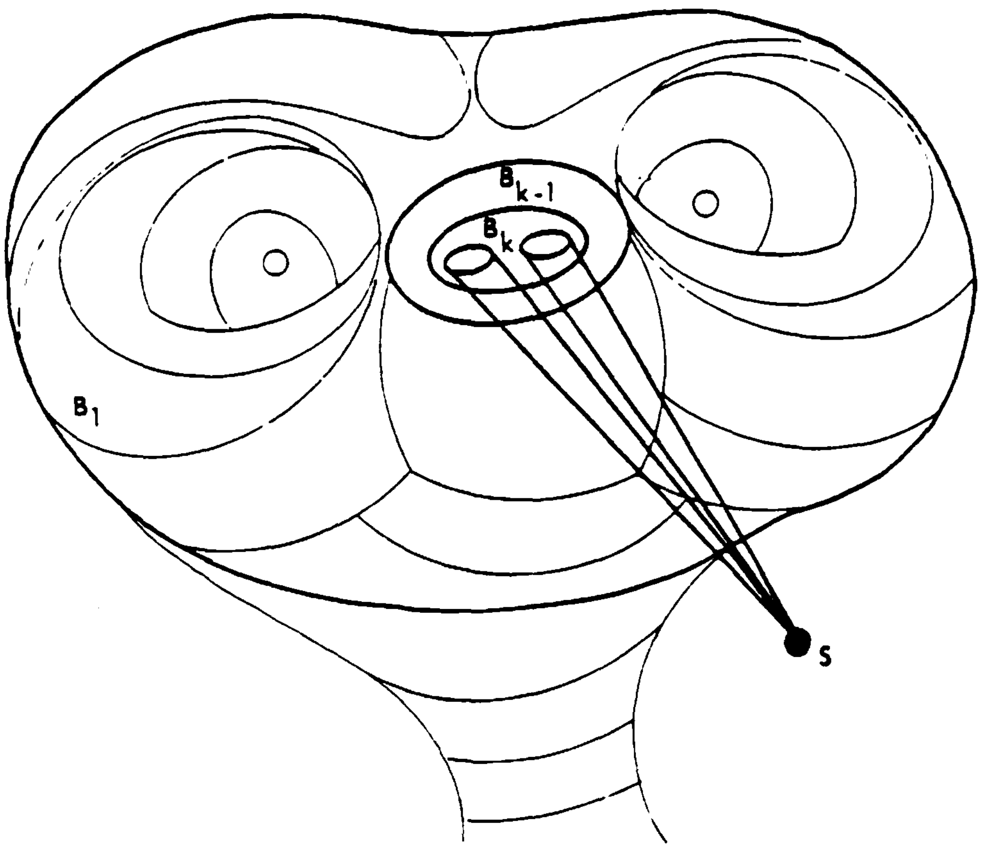


Figure 3. Edges from a single vertex to the innermost blossom that we may have to scan again and again if the blossoms $B_1, \ldots, B_k$ are eventually expanded.

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 when we make a change in the dual variables. It is now easy to verify that the seven 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 $\{(i,j) | i \text{ an S-vertex}, j \in B\}$. 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 corresponding to B to 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 j is in blossom B and $\pi_{ij} > 0$.

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

We insert (i,j) with priority π_{ij} to the active (nonactive) group corresponding to B.

(C5) B is an S-blossom.

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 of 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 them 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 $\pi_{ij} > 0$. Similarly, all z_k's in the p.q.$_1$ that computes δ_4 are positive. (Since an element is deleted as soon as its priority becomes 0.) Consequently, $\delta > 0$.

To derive an O(mn log n) 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 [AHU]. Note that the number of finds, concatenates and splits is O(n) per stage.

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

The 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.)

9. Conclusion.

We have considered four versions of the max 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.

I. Applications of Matching.

We do not list here the many applications of solutions to problems 1-4. For some applications see [L].

II. Generalization of Matching.

There are various ways that problems 1-4 can be generalized. For example Gabow [G3] has recently considered similar problems where some kinds of polygamy are allowed. He found efficient reductions to the corresponding matching problem.

III. Special cases of Matching.

Many applications solve one of the problems 1-4 but with only special graphs. Possibly, the extra information may lead to better algorithms. For example, Problem 1 is used to find routing in superconcentrators [GG]. The graphs that arise in this application have constant degree, and hence the solution given here takes time $O(n^{1.5})$. Perhaps this can be improved.

IV. Probabilistic 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 ([AV],[Ka]). An interesting problem is to find improved probabilistic algorithms which use random choices (rather than random inputs).

V. Approximation algorithms.

As for all optimization problems, we may settle for approximate solutions. For cardinality matching, the solution with the phases yields a good approximation by executing only a constant number of phases. For simple, fast and very good approximation algorithms for special graphs see [IMM], [KS].

We next discuss possible improvement 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 stages. Similarly, except for a logarithmic factor, improving the bound for weighted matching requires the use of an approach that does not make $O(n)$ augmentations. Perhaps the introduction of phases may lead to improved algorithms for problems 3, 4. Note that the solution to Problem 3 is slightly better than the solution to Problem 4 due to the use of Theorem 4. 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 solve efficiently any of the problems without augmenting paths?

Are blossoms necessary?

Can we solve Problem 4 without duality?

Assume 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 of 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.

Finally, we briefly consider parallel algorithms:

Can we solve any one of the four problems in time $O(\log^k n)$ with polynomial number of processors?

Is Problem 4 log-space complete for P?

A positive answer to the latter implies that a positive answer to the former is unlikely. Recently, the problem of Network Flow has been shown to be log-space complete for P [GSS]. As was observed in [BGH] there is a nonuniform algorithm that computes the size of the maximal 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 maximal matching.

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

References

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

[AV] D. Angluin and L.G. Valiant, Fast probabilistic algorithims for Hamiltonian paths and matchings, JCSS 18 (1979), 144-193.

[BGH] A. Borodin, J. von zur Gathen and J.E. Hopcroft, Fast parallel and gcd computations, Proc. 23rd IEEE Symp. on FOCS (1982), 64-71.

[D] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959), 263-271.

[Di] E.A. Dinic, Algorithm for solution of a problem of maximal flow in a network with power estimation, Soviet Math. Dokl. 11 (1970), 1277-1280.

[E1] J. Edmonds, Path, trees and flowers, Canad. J. Math. 17 (1965), 449-467.

[E2] J. Edmonds, Maximum matching and a polyhedron with 0,1 vertices, J. Res. NBS, 69B (April-June 1965), 125-130.

[EK] S. Even and O. Kariv, An $O(n^{2.5})$ algorithm for maximum matching in graphs, Proc. 16th IEEE Symp. on FOCS (1975), 100-112.

[ET] S. Even and R.E. Tarjan, Network flow and testing graph connectivity, SIAM J. on Comput. 4 (1975), 507-518.

[FF] L.R. Ford, and D.R. Fulkerson, Maximal flow through a network, Canadian J. Math. 8, 3 (1956), 399-404.

[G1] H.N. Gabow, Implementation of algorithms for maximum matching on nonbipartite graphs, Ph.D. Thesis, Department of Computer Science, Stanford University, 1974.

[G2] H.N. Gabow, An efficient implementation of Edmonds' algorithm for maximum matching on graphs, J. ACM 23 (1976), 221-234.

[G3] H.N. Gabow, Personal communication.

[Ga] Z. Galil, An $O(E^{2/3}V^{5/3})$ algorithm for the maximal flow problem, Acta Information 14 (1980), 221-242.

[GG] O. Gabber and Z. Galil, Explicit construction of linear-sized super concentrators, JCSS 22, 3 (1981), 407-420.

[GMG] Z. Galil, S. Micali and H.N. Gabow, Priority queues with variable priority and an O(EV log V) algorithm for finding a maximal weighted matching in general graphs, Proc. 23rd IEEE Symp. on FOCS (1982), 255-261.

[GT] H.N. Gabow and R.E. Tarjan, A linear time algorithm for a special case of disjoint set union, manuscript, July 1982 (to appear in Proc. 14th ACM STOC).

[GSS] L. Goldschlager, R. Shaw, and J. Staples, the maximum flow problem is log space complete for P, TCS 21 (1982), 105-111.

[IMM] M. Iri, K. Murota and S. Matsui, Linear time approximation algorithms for finding the minimum weight perfect matching on a plane, Info. Proc. Letters 12 (1981), 206-209.

[J] D. Johnson, Efficient algorithms for shortest paths in sparse graphs, J. ACM 24 (1977), 1-13.

[K] O. Kariv, An $O(n^{2.5})$ algorithm for maximal matching in general graphs, Ph.D. Thesis, Department of Applied Mathematics, The Weizman Inst., Rehovot, Israel, 1976.

[Ka] R.M. Karp, An algorithm to solve the assignment problem in expected time O(mn log n), Network 10, 2 (1980), 143-152.

[Kn] D.E. Knuth, The Art of Computer Programming, Vol 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[KM] T. Kameda and I. Munro, A $O(|V|\cdot|E|)$ algorithm for maximum matching of graphs, Computing 12 (1974), 91-98.

[KS] R.M. Karp and M. Sipser, Maximal matchings in sparse graphs, Proc. 22nd IEEE Symp. on FOCS (1981), 364-375.

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

[MV] S. Micali and V.V. Vazirani, An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs, Proc. 21st IEEE Symp. on FOCS (1980), 17-27.

[S] A.O. Slisenko, Recognition of palindromes by multihead Turing machines, in Problems in the Constructive Trend in Mathematics, VI (Proc. of the Steklov Institute of Mathematics 129), V.P. Orevkov and N.A. Sanin (eds.), Academy of Sciences of the USSR (1973), 30-202; English translation by R.H. Silverman, American Math. Society, Providence, Rhode Island (1976), 25-208.

[T1] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22, (1975), 215-225.

[T2] R.E. Tarjan, Finding optimum branchings, Network 7 (1977), 25-35.