An $O(\sqrt{|V|}\cdot|E|)$ Algorithm for Finding Maximum Matching in General Graphs

Silvio Micali * *and Vijay V. Vazirani* **
University of California - Berkeley

ABSTRACT

In this paper we present an $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding a maximum matching in general graphs. This algorithm works in 'phases'. In each phase a maximal set of disjoint minimum length augmenting paths is found, and the existing matching is increased along these paths.

Our contribution consists in devising a special way of handling blossoms, which enables an $O(|E|)$ implementation of a phase. In each phase, the algorithm grows Breadth First Search trees at all unmatched vertices. When it detects the presence of a blossom, it does not 'shrink' the blossom immediately. Instead, it delays the shrinking in such a way that the first augmenting path found is of minimum length. Furthermore, it achieves the effect of shrinking a blossom by a special labeling procedure which enables it to find an augmenting path through a blossom quickly.

PROBLEM STATEMENT AND PRELIMINARY DEFINITIONS

In this paper we present an efficient algorithm for finding a maximum matching in a general graph. The precise satement of the problem is as follows:

> Let G=(V,E) be a finite, undirected, connected graph (without loops or multiple edges) whose set of vertices is V and set of edges is E. A matching M is a subset of E such that no two edges of M are incident at a common vertex. A maximum matching is a matching whose cardinality is maximum.

We give the following basic definitions relative to a matching M:

If an edge is contained in M, then it is said to be 'matched', else it is said to be 'unmatched'. In this paper, matched edges will be drawn wiggly and unmatched edges will be drawn straight.

A vertex is 'free' if all edges incident at it are unmatched.

An 'alternating path' is a simple path whose edges are alternately in M and not in M.

An 'augmenting path' is an alternating path between two free vertices.

A HISTORICAL NOTE

The history of the maximum matching problem began in 1957 when Berge proved that a matching is maximum if and only if the graph has no augmenting paths. In 1965, Edmonds[1] used this result to give an $O(|V|^4)$ algorithm for this problem. Since then many combinatorists have solved this problem with better running time. Among them are Gabow[2], Kameda and Munro[3], and Lawler[4]. The best previous running times were due to Hopcroft and Karp[5] for bipartite graphs ($O(\sqrt{|V|}\cdot|E|)$), and to Even and Kariv[6] for general graphs ($O(|V|^{2.5})$). Our algorithm, close in spirit to that of Even and Kariv's, has a running time of $O(\sqrt{|V|}\cdot|E|)$.

SALIENT FEATURES OF THE ALGORITHM

The algorithm presented in this paper finds sets of augmenting paths in 'phases'. Given a matching M, a 'phase' may be defined as the process of finding a maximal set of disjoint minimum length augmenting paths (min aug paths) in the graph, and augmenting the matching along these paths. As shown by Hopcroft and Karp[5], only $O(\sqrt{|V|})$ such phases are needed for finding a maximum matching.

This research was supported by NSF Grant MCS-79-037667 and
* fellowship from Consiglio Nazionale della Ricerche - Italy, and
** Earle C. Anthony sholarship and Eugene C. Gee and Mona Fay Scholarship.

In order to describe the algorithm we first give the following definitions:

evenlevel: The evenlevel of a vertex v is the length of the minimum even length alternating path from v to a free vertex, if any, infinite otherwise.

oddlevel: The oddlevel of a vertex v is the length of the minimum odd length alternating path from v to a free vertex, if any, infinite otherwise.

level: The level of a vertex v is the minimum between evenlevel(v) and oddlevel(v), i.e. it is the length of the minimum alternating path from v to a free vertex.

outer: A vertex is outer iff level(v) is even.

inner: A vertex is inner iff level(v) is odd.

other level: If v is outer (inner) then its oddlevel (evenlevel) will be refered to as the other level of v.

bridge: An edge (u, v) is a bridge if either both evenlevel(u) and evenlevel(v) are finite, or both oddlevel(u) and oddlevel(v) are finite.

Note that since an augmenting path P has an odd length, every edge in P is a bridge. Note also that if there is a bridge (u, v), then some vertices (at least u and v) have both the evenlevel and the oddlevel finite.

We now explain the concept 'tenacity of a bridge':

tenacity: Given a bridge (u, v), tenacity ((u,v)) = min (evenlevel(u) + evenlevel(v), oddlevel(u) + oddlevel(v)) + 1.

So, the tenacity of a bridge represents the minimum length of a not necessarily simple alternating path from a free vertex to a free vertex containing the bridge. If such a path is simple, then it is an augmenting path. It can be proved that any min aug path P contains a bridge whose tenacity equals the length of P.

The algorithm consists of a main routine, SEARCH, and three subroutines: BLOSS-AUG (which is called with two vertices as parameters), FINDPATH and TOPOLOGICAL ERASE.

In each phase, SEARCH grows Breadth First Search (BFS) trees rooted at the free vertices of G in order to find the level of each vertex in G i.e. to find the evenlevel of outer vertices and the oddlevel of inner vertices. In order to do so SEARCH starts with the search level 0 and grows the BFS trees by incrementing the search level by one each time.

When SEARCH detects that a certain edge (u,v) is a bridge, it will call the subroutine BLOSS-AUG with the parameters u and v. If there is an augmenting path containing (u, v), its length is at least tenacity((u, v)). In fact, when BLOSS-AUG is called with parameters u and v, it looks for an augmenting path of exactly this length. So, if BLOSS-AUG is called at a lower search level for bridges having a lower tenacity, the first augmenting path found in a phase will have minimum length. Indeed, SEARCH calls BLOSS-AUG at search level i for bridges whose tenacity is 2i+1. This is accomplished by putting bridges whose tenacity is 2i+1 in the set bridges(i). Then, at the end of search level i, BLOSS-AUG is called for each edge in bridges(i).

In case there is no augmenting path of length tenacity((u,v)) containing the bridge (u, v), then BLOSS-AUG creates a new 'blossom' B (a set of vertices). Before this call, all vertices in B had exactly one level (even or odd) set to a finite value by SEARCH. During the present call, BLOSS-AUG will set to a finite value the other level of the vertices in B. In this process, some edges may be discovered to be bridges. The tenacity of these edges is computed, and they are inserted in the proper set of bridges.

When BLOSS-AUG detects the presence of an augmenting path containing (u,v), FINDPATH finds one such path, P. The present matching is increased along P; then TOPOLOGICAL ERASE removes the edges which, in the present phase, cannot be part of a min aug path disjoint from P. In a phase, if a min aug path is found at search level m, then a maximal set of disjoint 2m+1 long augmenting paths is found at the same search level and the phase ends. TOPOLOGICAL ERASE ensures that these paths are indeed disjoint. The fact that the phase ends when there are no more bridges having tenacity 2m+1 ensures that the set of min aug paths found is indeed maximal, since, as said, each min aug path P contains a bridge whose tenacity equals the length of P.

Since the algorithm executes a phase in $O(|E|)$ steps, it finds a maximum matching in $O(\sqrt{|V|}\cdot|E|)$ steps.

DESCRIPTION OF SEARCH

During the execution of a phase, SEARCH grows Breadth First Search trees rooted at the free vertices of G in order to find the level of each vertex.

SEARCH scans an edge at most once (in one of the two directions). A searched edge may be scanned in the opposite direction, by BLOSS-AUG. When this happens BLOSS-AUG marks the edge "used" to prohibit SEARCH from scanning it again.

At the start of a phase, the evenlevel and oddlevel of each vertex of G are set to infinity, to signify that no alternating path of any length has been found yet. Then, the evenlevel of each free vertex is reset to zero.

When the search level, i, is even, search is conducted from each vertex, v, with evenlevel(v)=i to find vertices u such that the edge (v, u) is "unused" and unmatched. If the oddlevel of u is infinity, then it is reset to i+1.

When i is odd, the search is conducted from each vertex, v, with oddlevel(v)=i, to find the unique matched neighbour, u, of v. Furthermore, the evenlevel of u is reset to i+1.

While growing the BFS trees, SEARCH constructs, for each searched vertex u, the set of its 'predecessors':

predecessors: Let u be a vertex of G which is not free. If u is inner and oddlevel(u)=2i+1 then v is a predecessor of u iff evenlevel(v)=2i and (u,v) is a member of E. If u is outer then v is a predecessor of u iff (u,v) is a matched edge.

The set of predecessors of each vertex u will be denoted by 'predecessors(u)'.

ancestor: Is the transitive (but non-reflexive) closure of the relation predecessor.

In addition, SEARCH constructs, for each inner vertex u, the set of its 'anomalies':

anomaly: Let u be an inner vertex and oddlevel(u) be 2i+1. Then v is an anomaly of u iff evenlevel(v)>2i+1 and (u, v) is a member of (E − M).

The set of anomalies of u will be denoted by 'anomalies(u)'.

EXAMPLE 1:

In figure 1, s and t are the predecessors of u, u is the predecessor of w, and v is an anomaly of u.

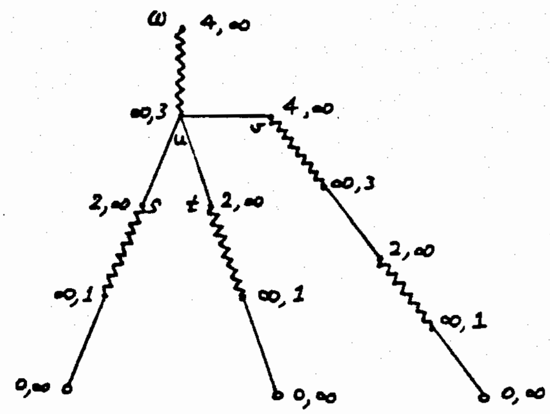

figure 1.

While scanning an edge, SEARCH checks to see if it is a bridge. When SEARCH discovers that an edge (u, v) is a bridge, it computes the tenacity of the edge, say 2i+1, and inserts (u, v) in bridges(i). At the end of search level i, SEARCH calls BLOSS-AUG, with parameters u and v, for each bridge in bridges(i). If during these calls, an augmenting path is found (more precisely, a maximal set of minimum length disjoint augmenting paths would be found), then the present matching will be increased and the phase will end. If instead, at the start of the present phase, the matching is already maximum, no augmenting paths can be found, but SEARCH will reach a search level i such that no vertices will have level i, and the algorithm will halt.

DESCRIPTION OF BLOSS-AUG

The subroutine BLOSS-AUG is called with vertices u and v such that the edge (u,v) is a bridge. This call will result either in the formation of a new blossom, or in the discovery of an augmenting path. A new blossom is formed if and only if the following condition holds:

BLOSSOMING CONDITION: there exist vertices, z, such that

1. z is an ancestor of both u and v.
2. u and v do not have any ancestors, other than z, whose level is equal to level(z).

If the blossoming condition does not hold, a min aug path is discovered.

<u>CONSTRUCTION OF A NEW BLOSSOM</u>. Assume that the blossoming condition holds for the bridge (u,v). Then BLOSS-AUG will construct a new blossom B. B will consist of all vertices w whose other level is still infinity, but can be set to a finite value due to the bridge (u,v), i.e. if w is inner (outer) there is a min even (odd) length alternating path, containing (u,v), from w to a free vertex. We give also an algorithm-oriented definition of B:

Among the z's of the blossoming condition which do not belong to any blossom, let b be the vertex whose level is maximum. Then the new blossom B is the set of vertices, w, such that

1. w does not belong to any other blossom when B is formed.
2. either w=u or w=v or w is an ancestor of u or w is an ancestor of v.
3. b is an ancestor of w.

Furthermore, b is designated to be the 'base' of B and u and v the 'peaks' of B.

<u>EXAMPLE 2</u>

Figure 2 shows the formation of a blossom. At search level 6, SEARCH detects the bridge (l, m), and calls BLOSS-AUG. During this call, blossom B is formed.

$$B = \{l,m,j,k,g,h,i,d,e,f\}.$$

The base of B is c and its peaks are l and m.

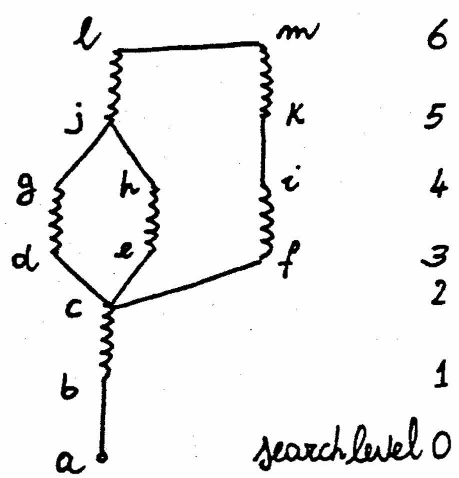

figure 2.

The following facts should be pointed out about blossoms:

1. At any stage in the algorithm, a vertex has both levels (even and odd) finite if and only if it belongs to a blossom at that stage.
2. A vertex can belong to at most one blossom.
3. The base, b, of a blossom B is always an outer vertex.
4. b does not belong to B because when B is being formed, there is no odd length alternating path from b to a free vertex.
5. As a consequence of fact 2, a peak of a blossom B does not necessarily belong to B.
6. Since at each search level i, SEARCH scans the edges in an arbitrary order, the set bridges(i) is formed in an arbitrary order. Consequently, our blossoms are not algorithm-independent structures. This point is illustrated in the next example.
7. If a vertex v belongs to a blossom B and it is contained in a min aug path P, then P also contains base(B).

<u>EXAMPLE 3</u>

At search level 4, if the bridge (i, j) is processed before (j, k), then the blossoms formed are:

$$B_1 = \{i,j,f,g\}$$

The base of B_1 is d and its peaks are i and j.

$$B_2 = \{k,h,d,e,b,c\}$$

The base of B_2 is a and its peaks are j and k.

However, if (j, k) is processed before (i, j) then the blossoms formed are:

$$B_1 = \{j,k,g,h,d,e,b,c\}$$

The base of B_1 is a and its peaks are j and k.

$$B_2 = \{i,f\}$$

The base of B_2 is a and its peaks are i and j.

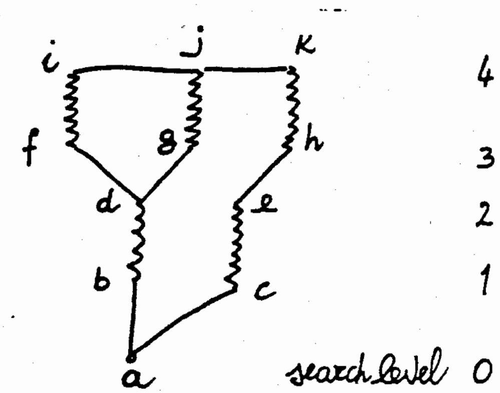

figure 3.

In order to accomplish the tasks of constructing blossoms and detecting the presence of augmenting paths within the running time of O(|E|) per phase, BLOSS-AUG performs a 'Double Depth First Search' (DDFS). The DDFS consists of growing two Depth First Search trees T_l and T_r contemporarily, i.e. if at a certain stage, the centers of activities of T_l and T_r are at v_l and v_r respectively, then the DDFS grows T_l if $\text{level}(v_l) \geq \text{level}(v_r)$, and it grows T_r otherwise. T_l and T_r are rooted at u and v respectively. This DDFS has the following special feature: when the search is conducted from a vertex w, which is the center of activity of one of the trees, say T_l, then the DDFS seeks only the vertices of predecessors(w) for growing T_l. While scanning an edge (w, p), where p is a member of predecessors(w), DDFS marks it "used" so that SEARCH may not scan (w, p) when it reaches w.

The vertices of T_l are marked "left" and those of T_r are marked "right" so that, in case an augmenting path contains these vertices, the function FINDPATH can find it.

During the DDFS, the two trees may find two different free vertices. In this case, an augmentation is possible. However, the search may not be so simple, for the two trees may meet at a vertex w. Then, clearly, only one of the trees can claim w and the free vertex reachable from it. So, first T_l is allowed to claim w (w is marked "left") Furthermore, T_r backs up and tries to find a vertex as deep as w, thus enabling the DDFS to proceed. However, if T_l fails, then T_r must claim w (the DDFS changes the mark on w to "right"). Now, T_l backs up and tries to find a vertex as deep as w. If T_l is also unsuccessful then an augmentation involving the edge (u, v) is not possible at this stage. This is so because there cannot be two disjoint alternating paths starting at u and v and reaching the same level as w. Now, a new blossom is created. The base of this blossom is w, its PeakL (left peak) is u, and its PeakR (right peak) is v. The blossom contains all of the vertices of T_l and T_r other than w, and the "right" mark on w is removed. At this point the other level of the vertices s in B is computed by the formula:

$$\text{tenacity}((u, v)) - \text{level}(s).$$

Once B is formed and the other level of its vertices is computed, some edges may be discovered to be bridges. Such newly discovered bridges are of two types: bridges having both endpoints in B, and bridges having only one endpoint in B.

For bridges (s, t) such that both s and t belong to B, the blossoming condition clearly holds. So, no augmenting path would be discovered if BLOSS-AUG is called with parameters s and t. Furthermore, the blossom B' that BLOSS-AUG would create will be empty because the other level of no new vertices can be set to a finite value due to (s, t). Therefore, such bridges are ignored.

For bridges (s, t) such that only one vertex, say s, belongs to B, it can be shown that s is an inner vertex and t is an anomaly of s. Conversely each anomaly of each inner vertex of B is a newly discovered bridge. So, BLOSS-AUG computes the tenacity, say 2j+1, of each such bridge and inserts it in bridges(j). Also, it marks the bridge"used". Note that if i is the present search level, then $j > i$.

Another special feature of the DDFS is that while the search is conducted from a vertex w to scan an edge (w, p), if p belongs to a blossom B_1 then it shifts the center of activity to $\text{base*}(B_1)$. In order to define the function base*(.), we introduce the partial order '$<$' on the bases of blossoms:

If B_1 and B_2 are blossoms, then,
$\text{base}(B_1) < \text{base}(B_2)$ iff
$\text{base}(B_1)$ belongs to B_2.

Furthermore the reflexive and transitive closure of will be denoted by '$\leq$'. Then,

$$\text{base*}(B_1) \overset{\text{def}}{=} \text{base}(B) \text{ iff } \text{base}(B_1) \leq \text{base}(B) \text{ and there is no } B' \text{ such that } \text{base}(B) < \text{base}(B').$$

This feature of the DDFS has the same effect as that of'shrinking' each blossom into a macronode located at its base*.

Clearly, the function base*(.) could be implemented by a Union Find. However, because of the special structure of blossoms, a path compression is sufficient to bound by O(|E|) the work done due to base* in a phase. Base* is implemented by a path compression as follows:

1. base*(B) = base(B) when B is formed, and
2. if just before a new computation of base*(B), $\text{base*}(B) = \text{base}(B_1)$, $\text{base*}(B_1) = \text{base}(B_2)$, ... $\text{base*}(B_k) = \text{base}(B')$, and $\text{base*}(B') = \text{base}(B')$, then, the new computation of base*(B) leaves upon termination $\text{base*}(B) = \text{base*}(B_1) = \ldots = \text{base*}(B_k) = \text{base}(B')$.

The subroutine uses two variables, DCV and barrier, whose function needs an explaination. At any stage, DCV (Deepest Common Vertex) points to the deepest vertex which has been discovered by both T_l and T_r. Before the first time that such a vertex is discovered, DCV is undefined. Barrier accomplishes the following task: suppose T_l and T_r meet at a vertex w. Furthermore, suppose that T_r backs up all the way and fails to find another vertex as deep as w; however, T_l is able to accomplish this task. Subsequently, T_l and T_r meet again. This time, T_r should not back up above w. This task of limiting T_r's backing up is accomplished by barrier. Barrier is initialized to v, and each time T_r fails during backtracking, barrier is shifted to the current DCV.

DESCRIPTION OF FINDPATH

When BLOSS-AUG detects the presence of a min aug path, it makes use of FINDPATH to find one such path, P.

FINDPATH is passed two vertices, "high" and "low" and a blossom B as parameters. High and low are such that level(high)$\geq$level(low) and they both belong to a common min aug path. FINDPATH returns the portion between high and low of one such path.

FINDPATH performs a Deapth First Search starting at high to find low. This Deapth First Search has some special features:

1. When the center of activity is at a vertex v belonging to B, the blossom passed as a parameter, only the predecessors of v are considered to continue the search. If the center of activity is transferred to one such predecessor, u, v is made the father of u.
2. It considers shrunk all blossoms other than B: assume that the center of activity is at a vertex v not belonging to B; it can be shown that v belongs to some other blossom B', then only base(B')=b is considered to continue the search. If the center of activity is transfered to base(B')=b then v is made the father of b.
3. The center of activity is never transferred to a vertex v$\notin$B such that its "left"/"right" mark is different from that of high, or to a vertex v whose level is less than that of low.

When the search succeeds in finding low (i.e. the center of activity is at low), FINDPATH constructs the 'generalized path' high=$x_1 \dots x_m$=low by reversing $x_m \dots x_1$: the father chain from low to high.

The path $x_1 \dots x_m$ is called a 'generalized path because it may not be a legal alternating path from high to low. This will be the case if x_j does not belong to B, for some j=1 ... m-1.

So for all such x_j, if any, OPEN is invoked. Its function is to open properly the blossom, say B', to which x_j belongs by finding an alternating path from x_j to base(B')=x_{j+1}.

If x_j is outer then OPEN calls FINDPATH with parameters x_j, x_{j+1}, B'.

If x_j is inner, then OPEN makes two calls to FINDPATH. Let us assume, w.l.o.g., that x_j is marked "left" (mark received at the time of the formation of B'). Then the first call finds a path, P_1, from PeakL(B') to x_j and the second a path, P_2, from PeakR(B') to base(B')=x_{j+1}. It should be noticed that P_1 and P_2 are disjoint. Let P^{-1} denote the reverse of P and "$\circ$" the concatenation operator. Then the alternating path from x_j to x_{j+1} is given by $P_1^{-1} \circ P_2$.

EXAMPLE 4

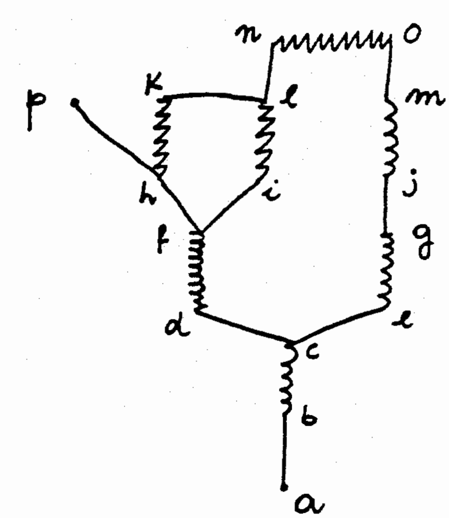

figure 4.

In this portion of graph there are two blossoms, B_1 and B_2. B_1={k,l,h,i} and base(B_1)=f; B_2={n,o,m,j,f,g,d,e} and base(B_2)=c.

FINDPATH is called with parameters high=p, low=a and B='undefined' (i.e. all blossom must be considered shrunk). The generalized path returned will be phfcba. Since $h \in B_1$ and $f \in B_2$, OPEN will be called twice. The first call will construct the path hklif (containing the bridge (k,l) since h is inner). The second call will construct the path fdc . The p-a path will then be phklifdcba.

DESCRIPTION OF TOPOLOGICAL ERASE

After FINDPATH has found a min aug path P and the matching has been increased along P, TOPOLOGICAL ERASE is called. This subroutine erases from the graph the path P and all those edges which cannot be part of a min aug path disjoint from P.

TOPOLOGICAL ERASE is very close in spirit to the well known topological sort. Each vertex has a counter which at any stage indicates the number of its unerased predecessor edges. A vertex is erased, along with all edges (predecessors or not) incident at it, either when its counter is decreased to zero or when it enters a min aug path detected by FINDPATH. Since the free vertices do not have any predecessor edges, their counter is set to one at the start of a phase, so it will remain one throughout the phase. It is not difficult to see that the total complexity of this routine is $O(|E|)$ per phase.

Note that if a blossom B is erased then all vertices in B are erased. Moreover, since FINDPATH puts in the augmenting path P the base of a blossom B whenever it puts in P a vertex belonging to B, we can also say that whenever a vertex of B is erased, all vertices in B are erased.

ACKNOWLEDGEMENTS

This work is affectionately dedicated to David Lichtenstein who gave us all the help a senior fellow student can give and much much more, and to Manuel Blum who supported our research in all possible ways and in some more ways possible only for him.

We are also very grateful to Giorgio Ausiello, Ravi Kannan, Richard Karp, Eugene Lawler and Robert Tarjan for their great patience in bearing with us through the first version of the algorithm and for their insightful criticism when we became clearer.

The avenue for approaching clarity was provided by Cheryl Khademan who gave us, as a present, a very pretty typed version of the routines.

In addition I, Silvio Micali, would like to express my deepest gratitude to Shimon Even for having introduced me to Graph Theory in the most stimulating way.

Routine SEARCH

(0) (initialization) For each vertex v, evenlevel(v):=infinite, oddlevel(v):=infinite, blossom(v):=undefined, predecessors(v):= ϕ, anomalies(v):= ϕ and v is marked "unvisited".
All edges are marked "unused" and "unvisited".
For i:=1 to |V| : bridges(i):= ϕ.
i:=-1.

(1) For each free vertex v, evenlevel(v):=0.

(2) i:=i+1.
If no more vertices have level i then HALT.

(3) If i is even then
for each v with evenlevel(v)=i find its unmatched, "unused" neighbors,
 for each such neighbor u:
  If evenlevel(u) is finite
   then temp:= (evenlevel(u) + evenlevel(v))/2,
    bridges(temp):=bridges(temp) $\cup$ {(u, v)}.
   else
    (a) (handle oddlevel) If oddlevel(u)=infinity then oddlevel(u):=i+1.
    (b) (handle predecessors) If oddlevel(u)=i+1 then predecessors(u):=predecessors(u) $\cup$ {v}.
    (c) (handle anomalies) If oddlevel(u) < i then anomalies(u):=anomalies(u) $\cup$ {v}.

(4) If i is odd then
for each v with (oddlevel(v)=i and v $\not\in$ B) take its matched neighbor u.
 (a) (handle bridges) If oddlevel(u)=i then
  temp:=(oddlevel(u) + oddlevel(v))/2,
  bridges((temp):=bridges(temp) U {(u, v)}

 (b) (handle predecessors) If oddlevel(u)=infinity then
  evenlevel(u):=i+1,
  predecessors(u):= {v}.

(5) For each edge (u, v) in bridges(i): call BLOSS-AUG(u, v).
If an augmentation occurred
 then go to step (0) (end of a phase)
 else go to step (2).

Note:

(1) "u $\not\in$ B" stands for "vertex u does not belong to any blossom," i.e., blossom(u) = undefined.
"u $\in$ B" stands for "vertex u belongs to a blossom. This blossom was named B", i.e., blossom(u) = B.

(2) The function base *($\cdot$) is defined in the description.

(3) The string operations "$^{-1}$" (inverse) and "$\circ$"(concatenation) are explained in the description.

Subroutine BLOSS-AUG (w_1, w_2 : vertices).

(0) (initialization) If w_1 and w_2 belong to the same blossom then go to step (5).
(neither is an augmentation possible, nor can a new blossom be created).
Otherwise, if $w_1 \in$ B then v_l :=base * (B)
else $v_l := w_1$.
If $w_2 \in$ B then v_r :=base * (B)
else $v_r := w_2$.
Mark v_l "left" and v_r "right".
f(v_l) is undefined, DCV is undefined, and barrier:= v_r.

(1.1) If v_l and v_r are free vertices then
P:=(FINDPATH(w_1, v_l , undefined) $)^{-1}\circ$ FINDPATH (w_2, v_r ,undefined).
Augment the matching along P, do a TOPOLOGICAL ERASE, and go to step (5).

(1.2) (v_l and v_r are not both free vertices)
If level(v_l) $\leq$ level(v_r)
then go to step (2.1)
else go to step (3.1).

(2.1) If v_l has no more "unused" ancestor edges then
if f(v_l) is undefined
then go to step (4) (create a new blossom)
else v_l :=f(v_l) and go to step (1.1).

(2.2) (v_l has "unused" ancestor edges). Choose an "unused" ancestor edge
$v_l \overset{e}{\text{—}}$ u. Mark e "used".
If u $\in$ B then u:=base * (B).

(a) If u is unmarked
then mark u "left", f(u):= v_l, v_l :=u , and
go to step (1.1).

(b) Otherwise (u is marked)
if u=barrier or u $\neq v_r$
then go to step (1.1).
else mark u "left", v_r :=f(v_r), v_l :=u,
DCV:=u, and go to step (1.1).

(3.1) If v_r has no more "unused" ancestor edges then
if v_r =barrier
then v_r :=DCV, barrier:=DCV, mark v_r "right",
v_l :=f(v_l), and go to step (1.1).
else v_r :=f(v_r) and go to step (1.1).

(3.2) (v_r has "unused" ancestor edges). Choose an "unused" ancestor edge
$v_r \overset{e}{\text{—}}$ u. Mark e used.
If u $\in$ B then u:=base * (B).

(a) If u is unmarked then mark it "right", f(u):= v_r, v_r :=u, and
go to step (1.1).

(b) Otherwise (u is marked)
if u= v_l then DCV:=u.
Go to step (1.1).

(4) (Creation of a new blossom)
Remove the "right" mark from DCV.
Create a new blossom (a set) B. Let B consist of all vertices that were marked "left" or "right" during the present call.
peakL(B):= w_1, peakR(B):= w_2, base (B):=DCV.

For each u in B :
blossom(u):=B.

(a) if u is outer then
oddlevel(u):= 2i + 1 - evenlevel(u)

(b) if u is inner then
evenlevel(u):= 2i + 1 - oddlevel(u),
for each v in anomalies(u) :
temp:= (evenlevel(u) + evenlevel(v))/2
bridges(temp):=bridges(temp) $\cup$ {(u, v)}.
Mark (u, v) "used".

(5) Return to SEARCH.

Function FINDPATH (high, low : vertices,
B : blossom)

0.0 (boundary condition) If high=low then Path:=high and go to step (8).

0.1 (initialization) v:=high.

1. If v has no more "unvisited" predecessor edges
then v:=f(v) and go to step (1).

2. If blossom(v) = B then choose an "unvisited" predecessor edge $v \overset{e}{—} u$. Mark e "visited".
else u:=base(blossom(v)).

3. If u=low then go to step (6) (the path has been found).

4. If (u is "visited") or (level(u) $\leq$ level(low)) or
(blossom(u)=B and u does not have the same "left"/"right" mark as high)
then go to step (1).

5. Mark u "visited".
f(u):=v, v:=u and go to step (1).

6. (u=low) Path:=low.
Until v=high do: Path:= v Path and v:=f(v).

7. (Path=$x_1 \cdots x_m$, where x_1 =high and x_m =low) For j=1 to $m-1$ do:
If blossom(x_j)$\neq$B then replace x_j and x_{j+1} with OPEN(x_j, x_{j+1}) in Path.

8. Return Path.

Function OPEN (entrance, base : vertices)

0. B:=blossom(entrance).

1. If entrance is outer
   then Path:=FINDPATH(entrance, base, B)
   and go to step (3).

2. (entrance is inner) Let PeakL and PeakR be the peak vertices of B.
   If entrance is marked "left"
   then Path := (FINDPATH(PeakL, entrance, B)) $^{-1}$ FINDPATH(PeakR, base, B)
   else Path := (FINDPATH(PeakR, entrance, B)) $^{-1}$ FINDPATH(PeakL, base, B)

3. Return Path.

References

[1] Edmonds, J., "Paths, Trees and Flowers"; Canadian J. 1965, Vol 17, pp. 449-467. "Maximum Matching and Polyhedron with 0,1 Vertices"; Journal of Research of the National Bureau of Standards, Jan.-June 1965, Vol. 69B, pp. 125-130.

[2] Gabow, H., "An Efficient Implementation of Edmonds' Maximum Matching Algorithm"; June 1972, Technical Report No. 31, Stan-CS, 72-328, to be published JACM. "Implementation of Algorithms for Maximum Matching on Non-Bipartite Graphs"; Ph.D. dissertation, Stanford University, 1973.

[3] Kameda, T. and Munro, I., "A $O(|V|\cdot|E|)$ Algorithm for Maximum Matching of Graphs"; Computing 1974, Vol. 12, pp. 91-98.

[4] Lawler, E.G., "Combinatorial Optimization Theory"; Holt, Rinehart and Winston, 1976, Chapter 6, pp. 217-239.

[5] Hopcroft, J.E. and Karp, R.M., "An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs"; SIAM J. on Comp. 2, December 1973, pp. 225-231.

[6] Even, S. and Kariv, O., "An $O(n^{2.5})$ Algorithm for Maximum Matching in General Graphs", Proceedings of the 16th Annual Symp. on Foundations of Computer Science (FOCS), Berkeley, 1975, pp. 100-112.