REPORT ACT-62 AUGUST, 1985

COORDINATED SCIENCE LABORATORY
APPLIED COMPUTATION THEORY GROUP

THE GENERAL MAXIMUM MATCHING ALGORITHM OF MICALI AND VAZIRANI

PAUL A. PETERSON

REPORT T-163

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GENERAL MAXIMUM MATCHING ALGORITHM OF MICALI AND VAZIRANI

PAUL A. PETERSON

AUGUST 1985

SUPPORTED BY THE EASTMAN KODAK COMPANY

TABLE OF CONTENTS

I. PROBLEM STATEMENT AND PRELIMINARY DEFINITIONS

This paper presents a simple explanation and computer implementation of the efficient algorithm of Micali and Vazirani that finds a maximum matching in a general graph in $O(\sqrt{|V|} \cdot |E|)$ time. Several of the passages below describing the algorithm have been lifted directly from the original description(1).

The precise statement 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.

If an edge is contained in M it is said to be "matched" otherwise it is said to be "unmatched". If a matched edge is made unmatched or an unmatched edge is made matched, then the matching of the edge is said to "inverted".

Figure 1 shows the result of one possible complete execution of the algorithm, where matched edges are drawn wiggly and unmatched edges are drawn straight (this way of distinguishing matched and unmatched edges is consistent throughout this paper).

(a) (b)

FIGURE 1

Figure 1-a is a general graph with no matching and Figure 1-b is the same graph with a maximum matching; there are, of course,

other maximum matchings possible.

The algorithm was developed by breaking a partially matched graph into simple components and making an analysis of their properties. The most important components are defined as follows:

exposed vertex: A vertex is exposed if all edges incident at it are unmatched.

alternating path: A path is alternating if its edges are alternately in M and not in M

augmenting path: A path is augmenting if it is an alternating path between two exposed vertices.

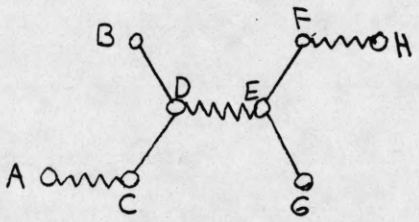

FIGURE 2

In Figure 2 the path (ACDEFH) is an alternating path, and the path (BDEG) is an augmenting path because it is an alternating path between the exposed vertices B and G.

In 1957 Berge proved the following theorem:

> A matching in a graph G is maximum if and only if there is no augmenting path in G with respect to M. (2)

This means that if in G there are no more augmenting paths with respect to M, then M is a maximum matching and, similarily, if M is maximum, then no augmenting paths exist. If there is an augmenting path in G, however, then there is a matching of cardinality $|M|+1$ obtained from M by simply inverting the matching of the edges in the augmenting path. Subsequent sections will call this process "increasing the matching". Since Figure 2 contained the augmenting path (BDEG) its matching can be increased as shown

in Figure 3; in fact, Figure 3 shows a maximum matching since no more augmenting paths are available.

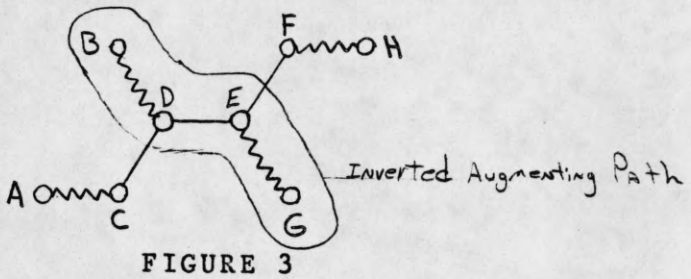

FIGURE 3

II. OVERVIEW OF THE ALGORITHM : (1) PHASES
(2) DEFINITIONS
(3) SUBROUTINE DESCRIPTIONS

(1) PHASES

The primary goal in obtaining a maximum matching is the discovery of augmenting paths. This algorithm finds sets of augmenting paths in "phases", where a "phase" is defined as the process of finding a maximal set of disjoint minimum length augmenting paths in the graph, and increasing the matching along these paths. As shown by Hopcroft and Karp(3), only $O(\sqrt{|V|})$ such phases are needed for finding a maximum matching.

(2) DEFINITIONS

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

level: The level of 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 an exposed vertex.

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

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

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

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

blossom: A blossom is a circuit of odd length, say 2k + 1, that has k matched edges.

Note that since an augmenting path P always has odd length every edge in P is a bridge. Also note that, given a bridge (u,v), it can be proved that the length of the minimum length augmenting path containing the bridge is equal to the tenacity of the bridge.

(3) SUBROUTINE DESCRIPTIONS

The algorithm consists of three main subroutines: SEARCH, BLOSS-AUG, and FINDPATH. Figure 4 has been provided to aid in the description of the function of each of these subroutines.

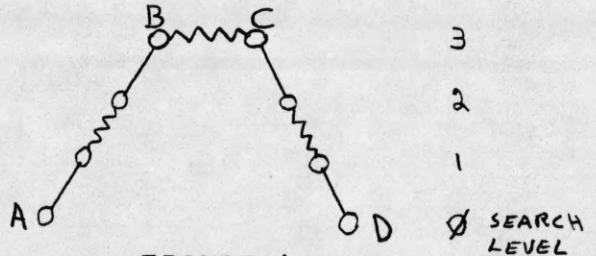

FIGURE 4

The function of SEARCH is to search simultaneously from all exposed vertices ("parallel" Breadth First Search) until it discovers bridges(i), the bridges at search level i. In Figure 4 SEARCH would start from the exposed vertices A and D to find the bridge (B,C) at search level 3. SEARCH then calls BLOSS-AUG for every bridge in bridges(i), passing the vertices of the bridge as a parameter.

Subroutine BLOSS-AUG performs a "Double Depth First Search" (DDFS) in order to find the exposed vertices of the augmenting path containing the bridge. In Figure 4 BLOSS-AUG would perform a depth first search from B to find A concurrently with a depth first search from C to find D.

Once the exposed vertices A and D are found, BLOSS-AUG calls the subroutine FINDPATH twice, once with parameters B and A, and once with parameters C and D. FINDPATH performs a simple Depth First Search to find the exact vertices of the alternating path between its two input vertices. The augmenting path is the concatenation of the two paths found by FINDPATH. Thus the exact vertices of the augmenting path are found and the matching may now be increased.

SEARCH continues to pass bridges to BLOSS-AUG until bridges(i) is depleted, at which time the phase comes to an end.

III. SUBROUTINE SEARCH : (1) FINDING A BRIDGE (2) PREDECESSORS

(1) FINDING A BRIDGE

Assuming that the only alternating paths in a graph G are simple paths (blossoms are considered later), then the discovery of a bridge indicates the discovery of an augmenting path.

To determine whether an edge (u,v) is a bridge SEARCH examines and compares the evenlevel and oddlevel of both u and v. If (u,v) meets the criteria for a bridge, then it is added to the set bridges(i), where i is the search level at which the bridge was found.

During the execution of a phase SEARCH simultaneously grows

Breadth First Search trees (parallel BFS) rooted at the exposed vertices of G in order to find the level of each vertex (i.e., to find the evenlevel of outer vertices and the oddlevel of inner vertices). In order to do so SEARCH starts with search level 0 and grows the parallel BFS trees by incrementing the search level by 1 each time.

SEARCH scans an edge at most once (in one of the two directions). Later subroutines may scan the edge in the opposite direction but if so will mark the edge "used" to prohibit SEARCH from scanning it again (i.e., the edge will not be scanned again in the present phase).

At the start of a phase the evenlevel and oddlevel of each vertex of G are initialized to infinity to signify that no alternating path of any length has been found yet. Then the evenlevel of each exposed vertex is initialized to 0.

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

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

Figure 5 shows the (evenlevel,oddlevel) labelings of each of the vertices in two searched graphs (the corresponding search levels are on the right). Figure 5-a shows the search beginning

at the exposed vertex, A, of a graph, and Figure 5-b shows the search of a graph with more than one exposed vertex--where the search begins simultaneously (parallel BFS) from the exposed vertices X, Y, and Z.

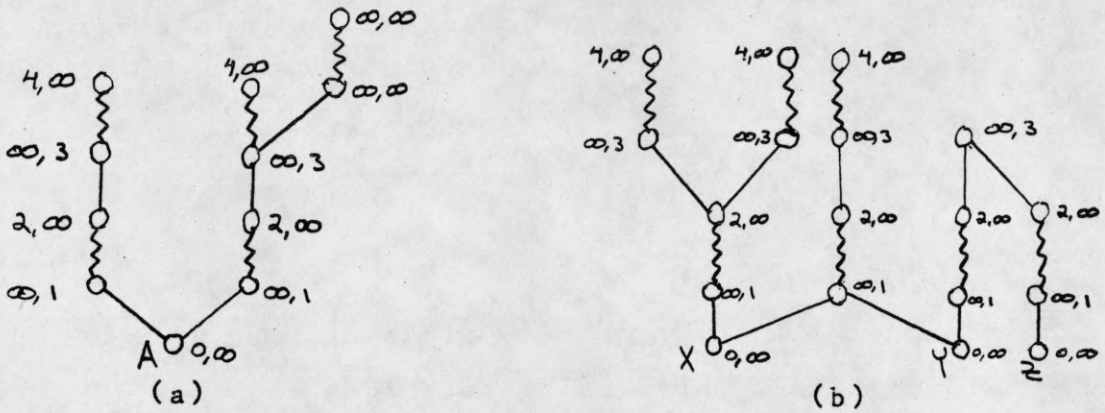

FIGURE 5

While scanning an edge during search level i, SEARCH checks to see whether it is a bridge. When SEARCH discovers that an edge (u,v) is a bridge, it inserts the bridge into the set bridges(i). If, at the end of search level i, the set bridges(i) is non-empty, then SEARCH has discovered at least one augmenting path. (Note that at the present time we are not considering blossoms.) SEARCH passes the vertices of a bridge in bridges(i) to a subroutine, BLOSS-AUG, which finds the exact vertices of the augmenting path and increases the matching. SEARCH continues to pass to BLOSS-AUG the vertices of every bridge in bridges(i) until the set is depleted, thus assuring that a maximal set of minimum length disjoint augmenting paths is found.

Since at least one augmenting path has been found the depletion of the set bridges(i) indicates the end of a phase. To begin the next phase the search level is reinitialized to 0, all previous marks and labelings attached to the vertices are

cleared, the evenlevel and oddlevel of each vertex in the graph are reinitialized as stated above, and SEARCH is invoked again.

If, at search level i, no edges are discovered to be bridges (i.e. at the end of search level i SEARCH finds the set bridges(i) to be empty), then SEARCH increments the search level by 1 and resumes examination of those vertices whose level corresponds to the new search level.

If instead, at the start of a phase, the matching is already maximum, then no augmenting paths can be found and the algorithm will halt; SEARCH recognizes the termination of the algorithm when SEARCH reaches a search level i such that no vertices have a corresponding level of i.

Figure 6 shows the discovery of bridges. Figure 6-a shows the discovery of the one bridge (A,B) at search level 3. Figure 6-b shows the discovery of the two bridges (W,X) and (Y,Z) at search level 4.

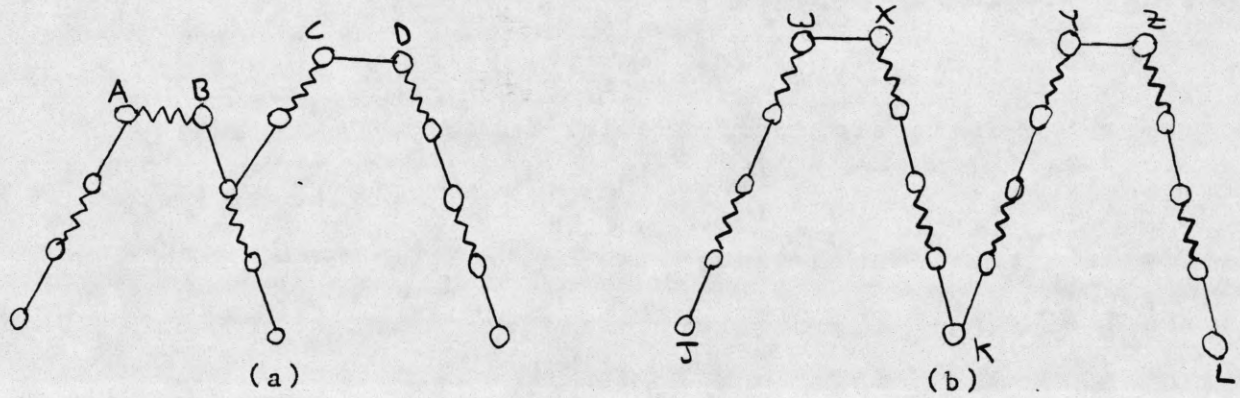

FIGURE 6

Note in Figure 6-a that the edge (A,B) is a bridge because it is the first edge (i.e., encountered at the smallest search level) whose vertices both had a finite oddlevel; the edge (C,D) will

never become a bridge: the present phase will come to an end before C and D are labeled. Also note that in Figure 6-b if the bridge (W,X) is choosen from bridges(i) first (the choice is arbitrary i.e., (Y,Z) could also have been choosen first) the matching will be increased along (J...WX...K); the vertex K, however, will no longer be exposed and (K...YZ...L) will no longer be an augmenting path. Therefore, there are conditions under which SEARCH could pass a bridge to BLOSS-AUG which is not actually part of an augmenting path. This condition is accounted for in the subsequent section of IV.(4) ERASING.

(2) PREDECESSORS AND ANOMONLIES

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

predecessor: Let u be a vertex of G which is not exposed. 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.

Let "predecessors(u)" denote the set of the predecessors of u, and "ancestor" the transitive (but non-reflexive) closure of the relation predecessor. Note that the level of each vertex in predecessors(u) is lower than level(u).

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).

Let "anomalies(u)" be the set of the anomalies of u.

In Figure 7, S and T are the predecessors of U, U is the predecessor of W, and V is an anomaly of U.

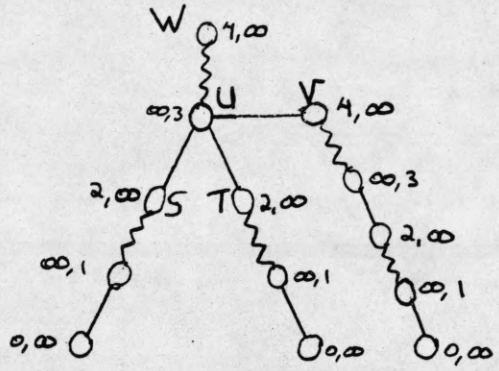

FIGURE 7

IV. BLOSS-AUG : (1) INTRODUCTION AND DESCIPTION
(2) TREE OVERLAP
(3) DCV AND BARRIER
(4) ERASING

(1) INTRODUCTION AND DESCRIPTION

Since SEARCH begins simultaneously from all exposed vertices to discover bridges, it is impossible to distinguish exactly which exposed vertices correspond to any particular bridge. Therefore, SEARCH calls subroutine BLOSS-AUG, passing to BLOSS-AUG the vertices of a bridge, so that BLOSS-AUG may find the exposed vertices of the augmenting path containing the bridge.

Let Length(y,z) be the length of the shortest alternating path from y to z. It is helpful to note the following fact: for every bridge(u,v) discovered at a search level i, there are exposed vertices x1 and x2 such that Length(u,x1) = Length(v,x2) = i (i.e., the total length of the augmenting path is 2i+1). Note that this will not be true for blossoms (blossoms are considered later in the paper).

BLOSS-AUG performs a Double Depth First Search to discover x1 and x2. The DDFS concurrently grows two Depth First Search

trees, TL and TR (denoting the left and right DFS trees), rooted at u and v respectively.

In order to grow the DFS trees, BLOSS-AUG uses two variables, vl and vr, which represent the vertices from which TL and TR are grown. When BLOSS-AUG is called, these variables are initialized as vl:=u and vr:=v. Note that, after this initialization, level(vl)=level(vr).

BLOSS-AUG has the following special feature: when a DFS tree, say TL, is being grown, then the DDFS seeks only the "unused" predecessors(vl) for growing TL. When the DDFS selects a predecessor of vl, say p, it marks p "used" and sets vl:=p (TR is grown in the same manner). Note that, due to the way predecessors are formed, the level of vl is now smaller than it previously was. In light of this, TL is grown if level(vl) >= level(vr) and TR is grown otherwise; in this manner the DFS trees are grown concurrently.

BLOSS-AUG also labels the vertices during the DDFS. The DDFS creates the "father" of a vertex p where, if p is a selected unused predecessor of w, the father(p)=w. Also, the vertices of TL are marked "left" and those of TR are marked "right" so that, once the exposed vertices are found, FINDPATH can find the complete augmenting path.

(2) TREE OVERLAP

During the DDFS, the two trees may find two different exposed vertices, thus allowing BLOSS-AUG to call FINDPATH. The search may not be so simple, however, for the two trees may meet at a vertex m. Then, clearly, only one of the trees can

claim m and the exposed vertex reachable from it. So, first TL is allowed to claim m (m is marked "left"). Furthermore, TR backs up (via father(vr)) and tries to find a vertex as deep as m, thus enabling the DDFS to proceed. However, if TR fails, then TR must claim m (the mark on m is changed to "right"). Now, TL backs up (via father(vl)) and tries to find a vertex as deep as m so that the DDFS can proceed. (If TL cannot find a vertex as deep as m then a blossom has been discovered. We handle the discussion of blossoms in section VI.)

(3) DCV AND BARRIER

BLOSS-AUG uses two other variables whose functions need explanation. These are DCV (Deepest Common Vertex) and Barrier. At any stage, DCV points to the deepest vertex (i.e., the vertex with the smallest level) which has been discovered by both TL and TR. Before the first time that such a vertex is discovered, DCV is undefined. (Note that if a blossom is discovered DCV will be its base.)

Barrier accomplishes the following task: suppose TL and TR meet at a vertex m. Furthermore, suppose that TR backs up all the way and fails to find another vertex as deep as m; however, TL is able to accomplish this task. Subsequently, TL and TR meet again. This time, TR should not back up above m. This task of limiting TR's backing up is accomplished by barrier. Barrier is initialized to v (recall, this is a vertex of bridge(u,v)), and each time TR fails during backtracking, barrier is shifted to the current DCV.

(4) ERASING

As mentioned in the discussion of Figure 6-b, SEARCH may pass to BLOSS-AUG a bridge which is no longer part of an augmenting path. This condition, however, will arise only if BLOSS-AUG has found and inverted at least one non-disjoint augmenting path within the same phase. In order that BLOSS-AUG not search for a non-existent augmenting path, every augmenting path that is found and inverted is immediately erased from the graph. This ensures the disjointness of every minimum augmenting path at a search level(i) in a phase.

The discussion of erasing vertices from the graph is carried out in three parts: the involvement of predecessor vertices, an ideal implementation, and the actual implementation. To avoid confusion, however, it should first be noted that the erasure of a vertex is confined to a single phase; that is, all marks of "erased" placed on a vertex are removed at the start of a new phase.

(4.1) PREDECESSORS AND ERASING

In order to completely ensure the disjointness of augmenting paths within a phase BLOSS-AUG erases two types of vertices. (1) BLOSS-AUG marks "erased" those vertices which are part of an augmenting path. (2) BLOSS-AUG will also mark a vertex w "erased" if every vertex in predecessors(w) is marked "erased". If w meets condition (2) then the set predecessors(w) is called "empty". Note that after such a vertex w has been erased the vertices y that were previously adjacent to w must be examined (or possible re-examined) to determine if the erasure of w has caused the set predecessors(y) to become "empty" and thus result in y

also needing to be erased.

(4.2) IDEAL IMPLEMENTATION

After the matching has been increased along P, every vertex in P is marked "erased". A straight-forward implementation would keep a list L of every erased vertex and remove from every set of predecessors and anomalies every vertex in L. If, in this process of removal, a set of predecessors(w) is discovered to be "empty", then w is added onto L. This process would be continued until there are no more vertices in L.

(4.3) ACTUAL IMPLEMENTATION

Since checking every set of predecessors after each erasure would be very time consuming, erasing is performed concurrently with the rest of the BLOSS-AUG routine. As before, when an augmenting path P is discovered, every vertex in P is marked "erased". Erasing of other vertices is then incorporated into BLOSS-AUG via the discovery of "unused" predecessors. When the DDFS attempts to select an "unused" predecessor, say p, of a vertex w, it first checks whether p has been marked "erased". If p is marked "erased" then it is deleted from the set predecessors(w) and another unused predecessor of w is sought. If the deletion of p from the set predecessors(w) causes the set to become "empty", then w is also marked "erased" and the center of activity (i.e., vl or vr) is moved to the father. If at any time one of the vertices of the bridge is marked "erased" then BLOSS-AUG terminates (no augmenting path can be found).

V. FINDPATH

After BLOSS-AUG has discovered the exposed vertices x1 and x2 of the augmenting path containing the bridge(u,v), subroutine FINDPATH is called. The purpose of FINDPATH is to find the exact vertices of the alternating path from u to x1 and from v to x2 (FINDPATH is called twice) so that the two paths may be concatenated and the matching increased along the entire augmenting path.

FINDPATH is passed two vertices, "high" and "low" (it is also passed a blossom as explained in section VI.(9)). High and low are such that level(high) >= level(low) and they both belong to a common augmenting path. FINDPATH returns the portion between high and low of one such path.

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

1) When the center of activity is at a vertex w only the predecessors of w are considered to continue the search.

2) A predecessor of w, say y, is selected only if the "left"/"right" mark of y is the same as that of high and the level(y) >= level(low).

3) When y is selected w is made the father of y.

Once the search succeeds in finding low FINDPATH constructs the path from high to low by reversing the father chain from low to high.

VI. BLOSSOMS :

Thus far this description of the algorithm has ignored the occurrence of blossoms. We shall now concern ourselves with the discovery, labeling, shrinking, and opening of blossoms.

(1) BLOSSOMS AND PHASES

When SEARCH discovers a bridge(u,v) there are two possible consequences:

1) The bridge corresponds to the discovery of an augmenting path.

2) The bridge corresponds to the discovery of a blossom.

In either case the bridge is passed to BLOSS-AUG. It is the function of BLOSS-AUG not only to find the exposed vertices of an augmenting path but also to detect the presence of blossoms. If BLOSS-AUG discovers that a bridge corresponds to a blossom, then it "shrinks" the blossom into a macronode by a special labeling procedure and then exits.

Note that the discovery of blossoms does not mark the end of a phase; i.e., if all bridges at search level(i) correspond to blossoms, then the search level is nevertheless incremented and the phase will not come to an end until a search level is reached for which there is at least one augmenting path.

(2) BLOSSOMING CONDITION

Blossoming Condition: For a bridge(u,v) 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).

Among the z's of the Blossoming Condition let b be the vertex whose level is maximum. The blossom B is the set of vertices w such that

1) w does not belong to any other blossom when B is formed.

2) b is an ancestor of w.

3) Either w=u or w=v or w is an ancestor of u or w is an ancestor of v.

Furthermore, b is designated to be the "base" of B and u and v the "peaks" of B. Figure 8 shows an example of a blossom, B1, where B1 = {H,I,F,G,D,E}, the peaks of B1 are H and I, the base(B1)=C, and the z's of the Blossoming Condition are A, B, and C.

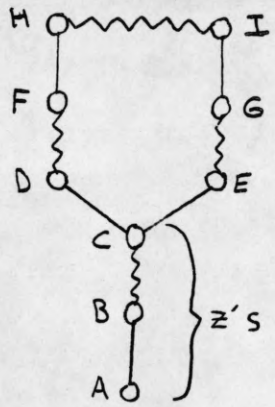

FIGURE 8

(3) DETECTION

As mentioned in section IV.(2), BLOSS-AUG detects a blossom when its two DFS trees meet at a vertex m such that TL and TR are unable to find another vertex as deep as m.

(4) CONSTRUCTION AND LABELING

Once a blossom has been detected the set of vertices in B is constructed. First, the "left"/"right" mark on DCV is removed. Then, B consists of all vertices that are marked "left" or "right" from the DDFS. Also, the peaks of B are given by PeakL:=root of TL, PeakR:=root of TR, and (as mentioned in section IV.(3)) the base of B by b:=DCV.

(5) SETTING THE OTHERLEVEL

Given a bridge(u,v) corresponding to a blossom B and a vertex w belonging to B it is helpful to note that if w is inner(outer) there is a minimum even(odd) length alternating path containing (u,v) from w to a free vertex. This fact allows the otherlevel of every vertex w in B to be set by otherlevel(w):=tenacity(u,v) - level(w). Figure 9 shows an example of a blossom before and after the otherlevel of each vertex in the blossom is set. Each vertex is labeled with its (evenlevel, oddlevel).

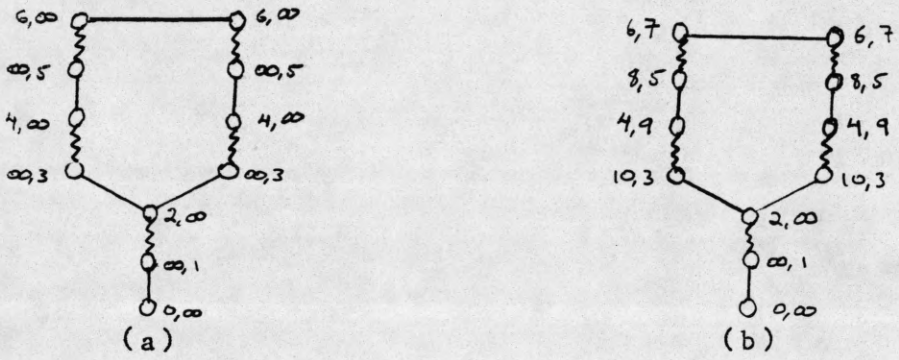

FIGURE 9

Once BLOSS-AUG has set the otherlevel of the vertices in B it must check for any bridges it might have formed. There can be only two types of newly discovered bridges: those having both

endpoints in B and those having only one endpoint in B.

For bridges(s,t) such that both s and t belong to B the Blossoming Condition clearly holds, i.e., (s,t) is not the bridge of an augmenting path but rather corresponds to a new blossom B′. Since every vertex that would be in B′ is already contained in B, the blossom B′ would be empty. 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. Conversly, each anomaly t of each inner vertex s of B defines a newly discovered bridge (s,t). So, BLOSS-AUG computes the tenacity, say 2j+1, of each such bridge (obtained from each set of anomalies of each vertex in B) and inserts it in bridges(j). Note that if the present search level is i, then j>i.

Figure 10-a shows the labeling of a graph with a blossom discovered from the bridge (U,V) before the otherlevel has been set. Figure 10-b shows the same graph after the otherlevel has been set; note that when the otherlevel of S is set, the bridge (S,T) is discovered.

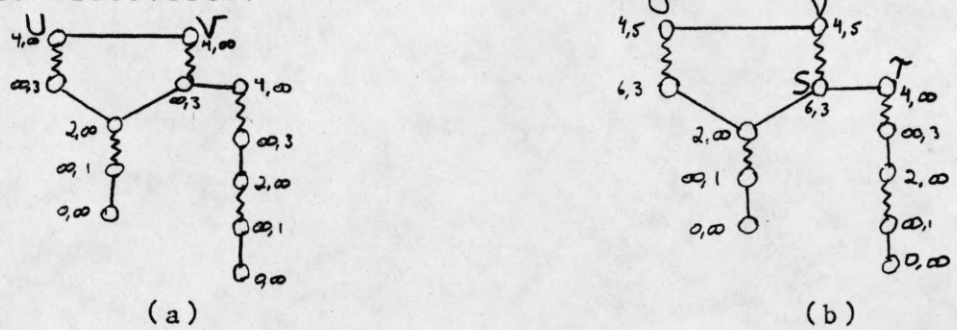

FIGURE 10

Also, after BLOSS-AUG sets the otherlevel of the vertices in B, SEARCH will continue to grow its parallel BFS trees from any

appropriate vertex, even if that vertex is contained in a blossom. It may be helpful to note that for the outer vertices in B the otherlevel is actually unnecessary for growing the paralell BFS trees; it is rather for the inner vertices in B that the otherlevel is used when SEARCH continues growing its paralell BFS trees.

Figure 11 shows the continuation of the paralell BFS even after the discovery of a blossom. Figure 11-a shows the discovery of a blossom at search level(3) from the bridge (U,V). Figure 11-b shows the same graph after the otherlevel of each vertex in the blossom has been set. Figure 11-c shows the continuation of the paralell BFS from the evenlevel setting of U (i.e., the otherlevel of U) and the eventual discovery of the bridge (U′,V′) at search level(5).

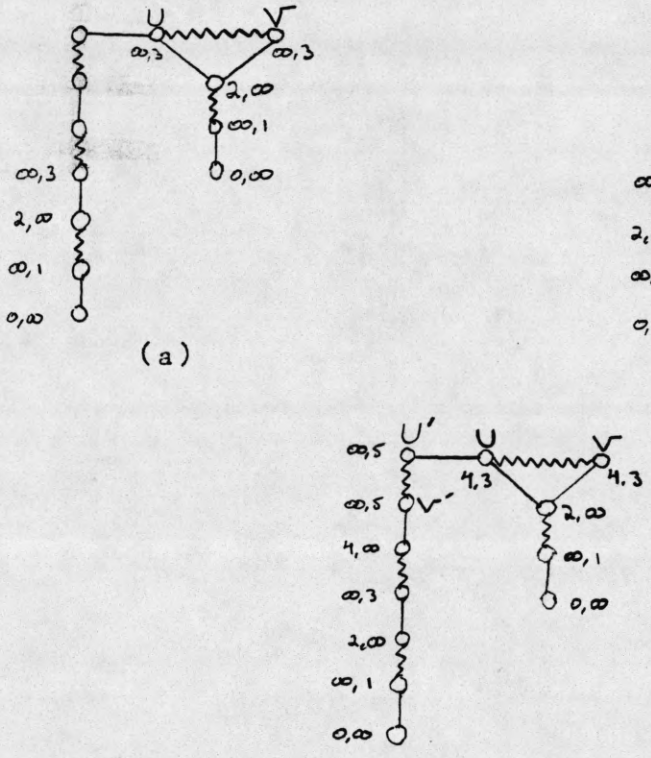

FIGURE 11

(6) FACTS ABOUT BLOSSOMS

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 iff 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 oddlength alternating path from b to free vertex.

5) As a consequence of fact 2, a peak of a blossom B does not necessarily belong to B. This is illustrated in Figure 14.

6) Since at each search level i, SEARCH scans the edges in arbitrary order, the set bridges(i) is formed in an arbitrary order. Consequently, blossoms are not algorithm-independent structures. This is illustrated in Figure 14.

7) If a vertex w belongs to a blossom B whose bridge is (u,v), and w is also contained in a minimum augmenting path P, then
i) if w is outer, then P contains the base of B.
ii) if w is inner P contains u, v, and the base of B.

(7) SOME EXAMPLES OF BLOSSOMS

EXAMPLE (1)

Figure 12 shows the formation of a blossom, B1, discovered from the bridge (F,G) at search level 4. B1 = {F,G,D,E}, the peaks of B1 are F and G, and the base of B1 is C.

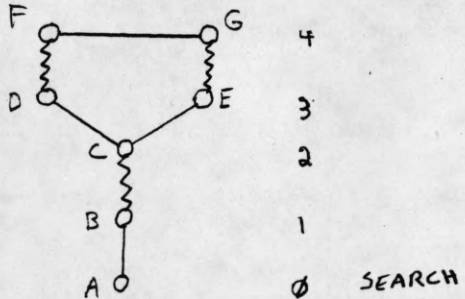

FIGURE 12

EXAMPLE (2)

Figure 13 shows the formation of a blossom, B1, discovered from the bridge (L,M) at search level 6.

B1 = {L,M,J,K,G,H,I,D,E,F}, the peaks of B1 are L and M, and the base of B1 is C.

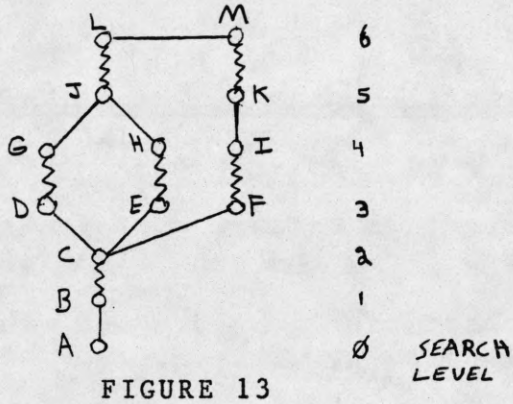

FIGURE 13

EXAMPLE (3)

Since SEARCH passes bridges to BLOSS-AUG in an arbitrary order there are certain blossoms which can be processed in more than one way. Figure 14 shows an example of this where SEARCH has discovered the two bridges (I,J) and (J,K).

CASE (i)

If BLOSS-AUG processes the bridge (I,J) before (J,K) then the blossoms formed are:

B1 = {I,J,F,G}
The peaks of B1 are I and J and its base is D.

B2 = {K,H,D,E,B,C}
The peaks of B2 are J and K and its base is A.

CASE (ii)

If BLOSS-AUG processes the bridge (J,K) before (I,J) then the blossoms formed are:

B1 = {J,K,G,H,D,E,B,C}
The peaks of B1 are J and K and its base is A.

B2 = {I,F}
The peaks of B2 are I and J and its base is A.

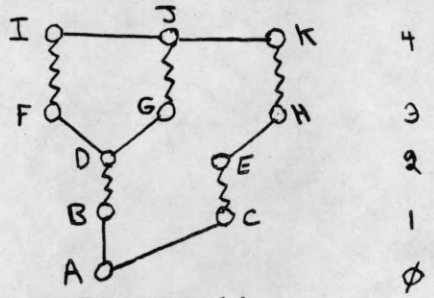

FIGURE 14

(8) EMBEDDED BLOSSOMS

An embedded blossom is a blossom whose base also belongs to a blossom. It is important to note that a blossom can be embedded several times, i.e., within more than one other blossom. Section VI.(7), EXAMPLE(3),CASE(i) illustrates an embedded blossom.

In order that BLOSS-AUG be able to handle blossoms and embedded blossoms its DDFS has one other special feature:

When the search is conducted from a vertex w to scan an edge (w,p), if p belongs to a blossom B1, then the DDFS shifts the center of activity (i.e., v1 or vr) to base*(B1).

In order to define the function base*(.) the partial order "<" is introduced on the bases of blossoms:

If B1 and B2 are blossoms, then
base(B1) < base(B2) iff base(B1) belongs to B2.

Furthermore, the reflexive and transitive closure of < will be denoted by "<*". Then,

$$\text{base*(B1)} \overset{\text{def}}{=} \text{base(B) iff base(B1) } <^* \text{ base(B)}$$
and there is no B′ such that
base(B) < base(B′).

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

In their description of the algorithm Micali and Vazirani

state without proof that "because of the special structure of blossoms, a path compression is sufficient to bound by O(|E|) the work done to base* in a phase". That is, for blossom B1 if

base(B1) < base(B2) < base(B3) < ... < base(B) and

base*(B1) = base(B),

then base* of B2, B3,... is set to base(B) too. Since then, Gabow and Tarjan have stated that their algorithm for incremental tree set union gives the O(|E|) time bound directly(4).

(9) OPENING A BLOSSOM

In order for FINDPATH to determine all vertices between high and low it must be able to "open" any blossoms which are in the path between high and low. To accomplish this FINDPATH is passed a blossom B as a parameter (along with the vertices high and low). Note that when BLOSS-AUG calls FINDPATH the blossom passed is "undefined".

It considers shrunk all blossoms other than B: assume that the center of activity (i.e. vl or vr) is at a vertex t not belonging to B; it can be shown that if t belongs to some other blossom B', then only base(B')=b is considered to continue the search. If the center of activity is transferred to base(B')=b then t is made the father of b.

Thus the path high=x(1)...x(m)=low that FINDPATH finds is only a "generalized path", i.e., it may not be a legal alternating path from high to low. If x(j) belongs to a blossom B'≠B, then x(j+1) = base(B'), and FINDPATH must find an alternating path through B' from x(j) to x(j+1).

To find such a path FINDPATH calls a subroutine OPEN which then recursively calls FINDPATH. There are two ways in which OPEN may call FINDPATH:

(i) If x(j) is outer then OPEN calls FINDPATH with parameters x(j), x(j+1), B´.

(ii) If x(j) is inner, then OPEN makes two calls to FINDPATH. Say x(j) is marked "left" (mark received when B´ was formed). Then the first call finds a path, P1, from PeakL(B´) to x(j) and the second a path, P2, from PeakR(B´) to base(B´)=x(j+1). It should be noticed that P1 and P2 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 $P1^{-1} \circ P2$.

Figure 15 shows a <u>portion</u> of a graph. In this portion there are two blossoms, B1 and B2. B1 = {K,L,H,I} and base(B1)=F; B2 ={N,O,M,J,F,G,D,E} and base(B2)=C. Note that B1 is embedded within B2.

FINDPATH is called with parameters high=P, low=A, and B="undefined" (i.e., all blossoms must be considered shrunk). The generalized path returned will be PHFCBA. Since $H \in B1$ and $F \in B2$, 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.

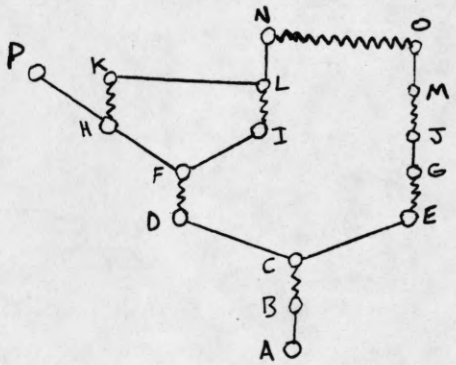

FIGURE 15

References

(1) S. Micali and V. Vazirani, "An $O(\sqrt{|V|} \cdot |E|)$ Algorithm for Finding Maximum Matching in General Graphs," Proc. 21st Annual Symp. on Found. Comp. Sci. (1980), pp. 17-27.

(2) C. Berge, "Two Theorems in Graph Theory," Proc. Nat. Acad. Sci. U.S.A., 43 (1957), pp. 842-844.

(3) J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ Algorithm For Maximum Matching in Bipartite Graphs," SIAM J. on Comp. 2, December 1973, pp. 225-231.

(4) H.N. Gabow and R.E. Tarjan, "A Linear-Time Algorithm for a Special Case of Disjoint Set Union," Proc. 15th Annual ACM Symp. on Theory of Computing (1983), pp. 246-251.

Routine SEARCH

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

(1)  For each exposed 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) ∪ {(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) ∪ {v}.
         (c) (handle anomalies)
             If oddlevel(u) < i then
              anomalies(u):=anomalies(u) ∪ {v}.

(4)  If i is odd then
     for each v with oddlevel(v)=i and v ∉ 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,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: "v ∉ B" stands for "vertex v does not belong to any blossom" i.e., blossom(v) = undefined.

Subroutine BLOSS-AUG (w1,w2:vertices)

```
(0)  (initialization) If w1 and w2 belong to the same blossom
     then go to step (5) (neither is an augmentation possible,
     nor can a new blossom be created).
     For each vertex v:
      clear each "left"/"right" mark,
      clear each f(v).
     If w1 ∈ B then v1:=base*(B)
               else v1:=w1.
     If w2 ∈ B then vr:=base*(B)
               else vr:=w2.
     Mark v1 "left" and vr "right".
     f(v1):=undefined, DCV:=undefined, and barrier:=vr.

(1.1)If v1 and vr are exposed vertices then
     P:=(FINDPATH(w1,v1,undefined))^-1 ∘ FINDPATH(w2,vr,undefined).
     Increase the mathing along P and go to  step (5).

(1.2)(v1 and vr are not both exposed vertices)
     If level(v1) >= level(vr)
      then go to step (2.1)
      else go to step (3.1).

(2.1)If v1 has no more "unused" ancestor edges then
     if f(v1) is undefined
      then go to step (4) (create a new blossom)
      else v1:=f(v1) and go to step (1.1).

(2.2)(v1 has "unused" ancestor edges) Choose an "unused" ancestor
     edge v1-e-u.
     If u is marked "erased"
      then delete u from predecessors(v1)
           if v1 has "unused" ancestor edges
            then go to step (2.2)
            else (v1 has no more "unused" ancestor edges)
             if v1=w1 then go to step (5)
                      else mark v1 "erased", v1=f(v1),
                           and go to step (1.1)
      else (u is not marked "erased")
       Mark e "used".
       If u∈ B then u:=base*(B).
       (a) If u is unmarked
            then mark u "left", f(u):=v1, v1:=u,
                 and go to step (1.1).
       (b) Otherwise (u is marked)
           if u=barrier or u = vr
            then go to step (1.1)
            else mark u "left", vr:=f(vr), v1:=u, DCV:=u,
                 and go to step (1.1).
```

```
(3.1)If vr has no more "unused" ancestor edges then
     if vr=barrier
      then vr:=DCV, barrier:=DCV, mark vr "right", v1:=f(v1),
           and go to step (1.1).
      else vr:=f(vr) and go to step (1.1).

(3.2)(vr has "unused" ancestor edges) Choose an "unused" ancestor
     edge vr--e-->u.
     If u is marked "erased"
      then delete u from predecessors(vr)
           if vr has "unused" ancestor edges
            then go to step (3.2)
            else (vr has no more "unused" ancestor edges)
             if vr=w2 then go to step (5)
                      else mark vr "erased", vr=f(vr),
                           and go to step (1.1)
      else (u is not marked "erased")
       Mark e "used".
       If u∈B then u:=base*(B).
       (a)If u is unmarked then mark it "right", f(u):=vr, vr:=u,
          and go to step (1.1).
       (b)Otherwise (u is marked)
          if u=v1 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):=w1, peakR(B):=w2, 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) ∪ {(u,v)}.
           Mark (u,v) "used".

(5)  Return to SEARCH.


Note: "f(v)" stands for the "father" of vertex v.
```

```
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--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) <= 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)...x(m), where x(1)=high and x(m)=low)
     For j:=1 to m-1 do:
      if blossom(x(j)) ≠ 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.
```

```
PROGRAM MXMATCH(INPUT, OUTPUT);
(* THIS PROGRAM FINDS A MAXIMUM MATCHING IN A GENERAL GRAPH          *)
(* USING THE ALGORITHM OF M. MICALI AND V. VAZIRANI.                 *)

  CONST
    NUMBEROFVERTICES = 20;

  TYPE
    VPTR = ^ VREC;
    VREC = RECORD
             V: INTEGER;
             NEXTV: VPTR
           END;
    VERPTR = ^ VERREC;
    VERREC = RECORD
               VERTEX, EVNLVL, ODDLVL, BLOSSNUM, FATHER: INTEGER;
               VERVISITED, EXPOSED, ERASED: BOOLEAN;
               MARK:
                 (RIGHT, LEFT, NULL);
               ADJHEAD, PREDHEAD, ANOMHEAD: VPTR;
               NEXTVER: VERPTR
             END;
    BRPTR = ^ BRREC;
    BRREC = RECORD
              B1, B2: INTEGER;
              NEXTBRIDGE: BRPTR
            END;
    BLOSSREC = RECORD
                 PEAKL, PEAKR, BASE,BSTAR: INTEGER;
                 BLOSSHEAD: VPTR;
               END;
  VAR
    VERTEX: ARRAY [1.. NUMBEROFVERTICES] OF VERPTR;
    EDGEVISITED: ARRAY [1.. NUMBEROFVERTICES, 1.. NUMBEROFVERTICES] OF
       BOOLEAN;
    EDGEUSED: ARRAY [1.. NUMBEROFVERTICES, 1.. NUMBEROFVERTICES] OF
       BOOLEAN;
    EDGEMATCHED: ARRAY [1.. NUMBEROFVERTICES, 1.. NUMBEROFVERTICES] OF
       BOOLEAN;
    BRIDGE:AARRAY [0.. NUMBEROFVERTICES] OF BRPTR;
    BLOSSARRAY: ARRAY [1.. NUMBEROFVERTICES] OF BLOSSREC;
    PATHHEAD: VPTR;
    INFINITE, SEARCHLVL, BLOSSCOUNT: INTEGER;
    AUGMENTED, DONE: BOOLEAN;

  FUNCTION LEVEL(I: INTEGER): INTEGER;
(* FUNCTION "LEVEL" RETURNS THE LEVEL OF VERTEX "I"                  *)

    BEGIN
      IF VERTEX[I]^.EVNLVL < VERTEX[I]^.ODDLVL
      THEN
        LEVEL := VERTEX[I]^.EVNLVL
      ELSE
        LEVEL := VERTEX[I]^.ODDLVL;
    END (*LEVEL*);
```

```
  FUNCTION REVERSE(HEADPTR, ENDPTR: VPTR): VPTR;
(* FUNCTION "REVERSE" REVERSES THE LIST FROM "HEADPTR" TO NIL.          *)
(* AFTER THE LIST HAS BEEN REVERSED, "REVERSE" POINTS TO THE HEAD OF    *)
(* THE LIST AND THE POINTER AT THE END OF THE REVERSED LIST IS          *)
(* REPLACED BY "ENDPTR".                                               *)

    VAR
      TEMP1, TEMP2: VPTR;

    BEGIN
      IF (HEADPTR <> NIL) AND (HEADPTR^.NEXTV <> NIL)
      THEN
        BEGIN
          TEMP1 := HEADPTR^.NEXTV;
          TEMP2 := HEADPTR^.NEXTV;
          HEADPTR^.NEXTV := ENDPTR;
          REPEAT
            TEMP2 := TEMP2^.NEXTV;
            TEMP1^.NEXTV := HEADPTR;
            HEADPTR := TEMP1;
            TEMP1 := TEMP2;
          UNTIL TEMP2 = NIL;
        END
      ELSE
        HEADPTR^.NEXTV := ENDPTR;
      REVERSE := HEADPTR;
    END (*REVERSE*);


  PROCEDURE BRIDGEINSERTION(LEVEL, V1, V2: INTEGER);
(* PROCEDURE "BRIDGEINSERTION" STACKS A DISCOVERED BRIDGE, V1--V2,      *)
(* ONTO A LIST ACCESSED BY ROW "LEVEL" IN THE ARRAY "BRIDGE".           *)

    VAR
      TEMP: BRPTR;

    BEGIN
      TEMP := BRIDGE[LEVEL];
      NEW(BRIDGE[LEVEL]);
      BRIDGE[LEVEL]^.B1 := V1;
      BRIDGE[LEVEL]^.B2 := V2;
      BRIDGE[LEVEL]^.NEXTBRIDGE := TEMP;
    END (*BRIDGEINSERTION*);
```

```
  PROCEDURE INPUTDATA;
(* PROCEDURE "INPUTDATA" CREATES AN ADJACENCY LIST FOR ALL VERTICES    *)
(* IT MARKS ALL VERTICES "EXPOSED", AND SETS THE VARIABLE "INFINITE"   *)
(* EQUAL TO THE NUMBER OF VERTICES.                                    *)

    VAR
      I: INTEGER;
      APOINTER, LASTAPTR: VPTR;

    BEGIN
      FOR I := 1 TO NUMBEROFVERTICES DO
        BEGIN
          NEW(VERTEX[I]);
          NEW(APOINTER);
          VERTEX[I]^.ADJHEAD := APOINTER;
          VERTEX[I]^.EXPOSED := TRUE;
          REPEAT
            READ(APOINTER^.V);
            EDGEMATCHED[I,APOINTER^.V]:=FALSE;
            LASTAPTR := APOINTER;
            NEW(APOINTER^.NEXTV);
            APOINTER := APOINTER^.NEXTV;
          UNTIL EOLN;
          LASTAPTR^.NEXTV := NIL;
          READLN;
        END;
      INFINITE := NUMBEROFVERTICES;
    END (*INPUTDATA*);

  PROCEDURE OUTPUTDATA;
(* PROCEDURE "OUTPUTDATA" PRINTS OUT THE ADJACENCY LIST AND INDICATES *)
(* WHETHER ADJACENT VERTICES ARE MATCHED OR UNMATCHED.                *)

    VAR
      I: INTEGER;
      APOINTER: VPTR;

    BEGIN
      FOR I := 1 TO NUMBEROFVERTICES DO
        BEGIN
          APOINTER := VERTEX[I]^.ADJHEAD;
          WHILE APOINTER <> NIL DO
            BEGIN
              WRITE(I: 4);
              IF EDGEMATCHED[I, APOINTER^.V]
              THEN
                WRITE(' IS MATCHED WITH ')
              ELSE
                WRITE(' IS NOT MATCHED WITH ');
              WRITELN(APOINTER^.V: 4);
              APOINTER := APOINTER^.NEXTV;
            END;
          WRITELN('**********END VERTEX', I: 4, '**********');
        END;
    END (*OUTPUTDATA*);
```

```
  PROCEDURE BLOSSAUG(W1, W2: INTEGER);
(* PROCEDURE "BLOSSAUG" USES THE VERTICES OF A BRIDGE,"W1" AND "W2"   *)
(* TO DETERMINE WHETHER A BLOSSUM OR AN AUGMENTING PATH EXISTS AND,   *)
(* IN EITHER CASE, DETERMINES THE CORRESPONDING SET OF VERTICES VIA   *)
(* A DOUBLE DEPTH FIRST SEARCH WHICH UTILIZES TWO SUBTREES, TLEFT     *)
(* AND TRIGHT.                                                        *)
(* IF AN AUGMENTING PATH IS FOUND, THEN THE MATCHING IS INCREASED     *)
(* ALONG IT.  IF A BLOSSUM IS FOUND, THEN THE VERTICES OF THE BLOSSUM *)
(* ARE MARKED ACCORDINGLY, A LIST OF THE BLOSSUMS RELEVANT DATA IS    *)
(* IS STORED IN AN ARRAY (BLOSSARRAY), AND BLOSSAUG ENDS.  BLOSSAUG   *)
(* ALSO HANDLES THE ERASING OF VERTICES.  IF "W1" OR "W2" IS ERASED,  *)
(* THEN BLOSSAUG ENDS.                                                *)

    VAR
      I, VL, VR, DCV, BARRIER: INTEGER;
      BLOSSDONE: BOOLEAN;


    FUNCTION BASESTAR(BLOSSUM: INTEGER): INTEGER;
(* FUNCTION "BASESTAR" RETURNS THE BASE* OF A BLOSSUM. "BASESTAR" IS  *)
(* IMPLEMENTED BY A PATH COMPRESSION WHICH IS UPDATED WHEN A BLOSSUM   *)
(* IS FORMED.  "BASESTAR" IS LISTED AS A SEPERATE SUBROUTINE FOR      *)
(* PROGRAM FLEXIBILITY, THAT IS, IT MAY BE ALTERED AND IMPLEMENTED BY *)
(* A DISJOINT SET UNION TREE AS DESCRIBED IN THIS PAPER.              *)

      BEGIN
        BASESTAR:=BLOSSARRAY[BLOSSUM].BSTAR;
      END (*BASESTAR*);


    FUNCTION UNUSEDPRED(I: INTEGER): INTEGER;
(* FUNCTION "UNUSEDPRED" RETURNS AN UNUSED PREDECESSOR OF VERTEX "I"  *)
(* IF "UNUSEDPRED" DISCOVERS NO UNUSED PREDECESSORS, THEN IT RETURNS  *)
(* A VALUE OF Ø.  IF IT DISCOVERS AN UNUSED PREDECESSOR IS            *)
(* MARKED "ERASED", THEN IT DELETES THE PREDECESSOR FROM THE LIST     *)
(* OF "I"'S PREDECESSORS; IF ALL PREDECESSORS ARE DELETED IN THIS WAY *)
(* THEN "UNUSEDPRED" RETURNS A VALUE OF -1, INDICATING THAT VERTEX    *)
(* "I" SHOULD BE ERASED; IF VERTEX "I" SHOULD BE ERASED BUT "I"="W1"  *)
(* OR "I"="W2", THEN "UNUSEDPRED" RETURNS A VALUE OF -2, INDICATING   *)
(* THAT "BLOSSAUG" SHOULD END.                                        *)

      VAR
        LASTPPTR, PPOINTER: VPTR;
        DONE: BOOLEAN;

      PROCEDURE DELETEPRED;
(* PROCEDURE "DELETEPRED" REMOVES THE CURRENT VERTEX FROM "I"'S LIST  *)
(* OF PREDECESSORS.                                                   *)
        BEGIN
          IF PPOINTER = VERTEX[I]^.PREDHEAD
          THEN
            VERTEX[I]^.PREDHEAD := PPOINTER^.NEXTV
          ELSE
            LASTPPTR^.NEXTV := PPOINTER^.NEXTV;
        END (*DELETEPRED*);
```

```
BEGIN (*UNUSEDPRED*)
  UNUSEDPRED := 0;
  DONE := FALSE;
  PPOINTER := VERTEX[I]^.PREDHEAD;
  LASTPPTR := PPOINTER;
  IF PPOINTER = NIL THEN
    BEGIN
      UNUSEDPRED := - 1;
      VERTEX[I]^.ERASED := TRUE;
      DONE := TRUE;
      IF (I = W1) OR (I = W2) THEN
        BEGIN
          BLOSSDONE := TRUE;
          UNUSEDPRED := - 2;
        END;
    END;
  WHILE NOT DONE DO
    BEGIN
      IF VERTEX[PPOINTER^.V]^.ERASED
      THEN
        BEGIN
          DELETEPRED;
          IF VERTEX[I]^.PREDHEAD = NIL
          THEN
            BEGIN
              UNUSEDPRED := - 1;
              VERTEX[I]^.ERASED := TRUE;
              DONE := TRUE;
              IF (I = W1) OR (I = W2) THEN
                BEGIN
                  BLOSSDONE := TRUE;
                  UNUSEDPRED := - 2;
                END;
            END;
        END
      ELSE
        IF EDGEUSED[I, PPOINTER^.V]
        THEN
          BEGIN
            LASTPPTR := PPOINTER;
            PPOINTER := PPOINTER^.NEXTV;
            IF PPOINTER = NIL THEN
              BEGIN
                DONE := TRUE;
              END;
          END
        ELSE
          BEGIN
            UNUSEDPRED := PPOINTER^.V;
            EDGEUSED[I, PPOINTER^.V] := TRUE;
            EDGEUSED[PPOINTER^.V, I] := TRUE;
            DONE := TRUE;
          END;
    END;
END (*UNUSEDPRED*);
```

```
    FUNCTION FINDPATH(HIGH, LOW, B: INTEGER; ENDPTR: VPTR): VPTR;
(* FUNCTION "FINDPATH" RETURNS A LIST OF THE VERTICES IN THE          *)
(* ALTERNATING PATH BETWEEN "HIGH" AND "LOW".  WHEN THE FUNCTION IS   *)
(* COMPLETED, "FINDPATH" POINTS TO "HIGH" (THE HEAD OF THE LIST) AND  *)
(* THE END POINTER IS "ENDPTR".  "FINDPATH" ALSO DETERMINES WHETHER   *)
(* THE ALTERNATING PATH IS LEGAL, THAT IS, IT OPENS ANY BLOSSUMS      *)
(* WITHIN THE PATH.                                                   *)

      VAR
        TEMP, FRONT, LASTFRONT, TEMPHEAD: VPTR;
        V, U: INTEGER;
        DONE: BOOLEAN;


      FUNCTION OPEN(ENTRANCE, BASE: INTEGER; ENDPTR: VPTR): VPTR;
(* FUNCTION "OPEN" RECURSIVELY CALLS "FINDPATH" TO DETERMINE THE      *)
(* ALTERNATING PATH THROUGH A BLOSSUM.                                *)

        VAR
          B: INTEGER;

        BEGIN
          B := VERTEX[ENTRANCE]^.BLOSSNUM;
          IF ODD(LEVEL(ENTRANCE))
          THEN
            IF VERTEX[ENTRANCE]^.MARK = LEFT
            THEN
              OPEN := REVERSE(FINDPATH(BLOSSARRAY[B].PEAKL, ENTRANCE, B
                  , NIL), FINDPATH(BLOSSARRAY[B].PEAKR, BASE, B, ENDPTR))
            ELSE
              OPEN := REVERSE(FINDPATH(BLOSSARRAY[B].PEAKR, ENTRANCE, B
                  , NIL), FINDPATH(BLOSSARRAY[].PEAKL, BASE, B, ENDPTR))
          ELSE
            OPEN := FINDPATH(ENTRANCE, BASE, B, ENDPTR);
        END (*OPEN*);


      FUNCTION UNVISITEDPRED(I: INTEGER): INTEGER;
(* FUNCTION "UNVISITEDPRED" RETURNS AN UNVISITED PREDECESSOR OF       *)
(* VERTEX "I".                                                       *)

        VAR
          PPOINTER: VPTR;
          DONE: BOOLEAN;

        BEGIN
          UNVISITEDPRED := Ø;
          PPOINTER := VERTEX[I]^.PREDHEAD;
          DONE := FALSE;
          WHILE (PPOINTER <> NIL) AND (NOT DONE) DO
            BEGIN
```

```
        IF NOT EDGEVISITED[I, PPOINTER^.V]
        THEN
          BEGIN
            EDGEVISITED[I, PPOINTER^.V] := TRUE;
            EDGEVISITED[PPOINTER^.V, I] := TRUE;
            UNVISITEDPRED := PPOINTER^.V;
            DONE := TRUE;
          END
        ELSE
          PPOINTER := PPOINTER^.NEXTV;
      END;
  END (*UNVISITEDPRED*);


BEGIN (*FINDPATH*)
  NEW(TEMP);
  TEMP^.NEXTV := ENDPTR;
  DONE := FALSE;
  IF HIGH = LOW
  THEN
    BEGIN
      TEMP^.V := HIGH;
      FINDPATH := TEMP;
    END
  ELSE
    BEGIN
      V := HIGH;
      WHILE NOT DONE DO
        BEGIN
          U := UNVISITEDPRED(V);
          WHILE U = Ø DO
            BEGIN
              V := VERTEX[V]^.FATHER;
              U := UNVISITEDPRED(V);
            END;
          IF VERTEX[V]^.BLOSSNUM <> B THEN
            BEGIN
              EDGEVISITED[U, V] := FALSE;
              EDGEVISITED[V, U] := FALSE;
              U := BLOSSARRAY[VERTEX[V]^.BLOSSNUM].BASE;
            END;
          IF U = LOW
          THEN
            DONE := TRUE
          ELSE
            IF (VERTEX[U]^.VERVISITED) OR (LEVEL(U) <= LEVEL(LOW))
            THEN
            ELSE
              BEGIN
                VERTEX[U]^.VERVISITED := TRUE;
                VERTEX[U]^.FATHER := V;
                V := U;
              END;
        END;
```

```
(*DONE=TRUE*)
            TEMP^.V := LOW;
            WHILE V <> HIGH DO
              BEGIN
                NEW(FRONT);
                FRONT^.NEXTV := TEMP;
                FRONT^.V := V;
                TEMP := FRONT;
                V := VERTEX[V]^.FATHER;
              END;
            NEW(FRONT);
            FRONT^.V := HIGH;
            FRONT^.NEXTV := TEMP;
            FINDPATH := FRONT;
            TEMPHEAD := FRONT;
            LASTFRONT := FRONT;
            WHILE FRONT^.V <> LOW DO
              BEGIN
                IF VERTEX[FRONT^.V]^.BLOSSNUM <> B
                THEN
                  BEGIN
                    IF FRONT = TEMPHEAD
                    THEN
                      BEGIN
                        TEMPHEAD := OPEN(FRONT^.V, FRONT^.NEXTV^.V,
                           FRONT^.NEXTV^.NEXTV);
                        FINDPATH := TEMPHEAD;
                      END
                    ELSE
                      LASTFRONT^.NEXTV := OPEN(FRONT^.V, FRONT^.NEXTV^.
                         V, FRONT^.NEXTV^.NEXTV);
                    FRONT := FRONT^.NEXTV^.NEXTV;
                    WHILE LASTFRONT^.NEXTV <> FRONT DO
                      LASTFRONT := LASTFRONT^.NEXTV;
                    IF LASTFRONT^.V = LOW THEN
                     FRONT:=LASTFRONT;
                  END
                ELSE
                  BEGIN
                    LASTFRONT := FRONT;
                    FRONT := FRONT^.NEXTV;
                  END;
              END;
          END;
      END (*FINDPATH*);
```

```
    PROCEDURE INCREASEMATCHING;
(* PROCEDURE "INCREASEMATCHING" INCREASES THE MATCHING ALONG THE      *)
(* AUGMENTING PATH CORRESPONDING TO THE BRIDGE (W1,W2).               *)

      VAR
        PATHHEAD, TEMP1P, TEMP2P: VPTR;
        MATCH: BOOLEAN;

      BEGIN
        PATHHEAD := REVERSE(FINDPATH(W1, VL, 0, NIL), FINDPATH(W2, VR,
           0, NIL));
        TEMP1P := PATHHEAD;
        TEMP2P := PATHHEAD^.NEXTV;
        MATCH := TRUE;
        VERTEX[TEMP1P^.V]^.ERASED := TRUE;
        VERTEX[TEMP1P^.V]^.EXPOSED := FALSE;
        WHILE TEMP2P <> NIL DO
          BEGIN
            IF MATCH
            THEN
              BEGIN
                EDGEMATCHED[TEMP1P^.V, TEMP2P^.V] := TRUE;
                EDGEMATCHED[TEMP2P^.V, TEMP1P^.V] := TRUE;
               MATCH := FALSE;
              END
            ELSE
              BEGIN
                MATCH := TRUE;
                EDGEMATCHED[TEMP1P^.V, TEMP2P^.V] := FALSE;
                EDGEMATCHED[TEMP2P^.V, TEMP1P^.V] := FALSE;
              END;
            VERTEX[TEMP2P^.V]^.ERASED := TRUE;
            TEMP1P := TEMP2P;
            TEMP2P := TEMP2P^.NEXTV;
          END;
        VERTEX[TEMP1P^.V]^.EXPOSED := FALSE;
      END (*INCREASEMATCHING*);
```

```
    PROCEDURE CREATEBLOSSUM;
(* PROCEDURE "CREATEBLOSSUM" LABLES EVERY VERTEX IN THE BLOSSUM,      *)
(* STORES A LIST OF THE BLOSSUM'S RELEVANT DATA IN THE ARRAY          *)
(* "BLOSSARRAY", UPDATES THE BASE* OF THE BLOSSUM, SETS THE OTHERLEVEL*)
(* OF THE VERTICES IN THE BLOSSUM, AND CHECKS TO SEE WHETHER ANY NEW  *)
(* NEW BRIDGES HAVE BEEN FORMED.                                      *)

      VAR
        TEMPPTR, LASTPTR, ANOMPTR: VPTR;
        TEMP, I: INTEGER;

      BEGIN
        BLOSSCOUNT := BLOSSCOUNT + 1;
        VERTEX[DCV]^.MARK := NULL;
        BLOSSARRAY[BLOSSCOUNT].PEAKL := W1;
        BLOSSARRAY[BLOSSCOUNT].PEAKR := W2;
        BLOSSARRAY[BLOSSCOUNT].BASE := DCV;
        IF VERTEX[BLOSSARRAY[BLOSSCOUNT].BASE]^.BLOSSNUM=0
         THEN
          BLOSSARRAY[BLOSSCOUNT].BSTAR:=DCV
         ELSE
          BLOSSARRAY[BLOSSCOUNT].BSTAR:=BLOSSARRAY
            [VERTEX[BLOSSARRAY[BLOSSCOUNT].BASE]^.BLOSSNUM].BSTAR;
        NEW(TEMPPTR);
        BLOSSARRAY[BLOSSCOUNT].BLOSSHEAD := TEMPPTR;
        FOR I := 1 TO NUMBEROFVERTICES DO
          IF VERTEX[I]^.MARK <> NULL THEN
            BEGIN
              VERTEX[I]^.BLOSSNUM := BLOSSCOUNT;
              TEMPPTR^.V := I;
              LASTPTR := TEMPPTR;
              NEW(TEMPPTR^.NEXTV);
              TEMPPTR := TEMPPTR^.NEXTV;
              IF ODD(LEVEL(I))
              THEN
                BEGIN (* I.E. WORKING WITH INNER VERTEX *)
                  VERTEX[I]^.EVNLVL := (2 * SEARCHLVL) + 1 - VERTEX[I]^
                     .ODDLVL;
                  ANOMPTR := VERTEX[I]^.ANOMHEAD;
                  WHILE ANOMPTR <> NIL DO
                    BEGIN
                      TEMP := (VERTEX[I]^.EVNLVL + VERTEX[ANOMPTR^.V]^.
                         EVNLVL) DIV 2;
                      BRIDGEINSERTION(TEMP, I, ANOMPTR^.V);
                      EDGEUSED[I, ANOMPTR^.V] := TRUE;
                      EDGEUSED[ANOMPTR^.V, I] := TRUE;
                      ANOMPTR := ANOMPTR^.NEXTV;
                    END
                END
              ELSE
                VERTEX[I]^.ODDLVL := (2 * SEARCHLVL) + 1 - VERTEX[I]^.
                   EVNLVL;
            END;
        LASTPTR^.NEXTV := NIL;
      END (*CREATEBLOSSUM*);
```

```
    PROCEDURE RIGHTSIDE;
(* PROCEDURE "RIGHTSIDE" GROWS THE SUBTREE TRIGHT OF"BLOSSAUG"'S     *)
(* DOUBLE DEPTH FIRST SEARCH.                                        *)

      VAR
        U: INTEGER;

      BEGIN
        U := UNUSEDPRED(VR);
        WHILE U = - 1 DO
          BEGIN
            VR := VERTEX[VR]^.FATHER;
            U := UNUSEDPRED(VR);
          END;
        IF NOT BLOSSDONE
        THEN
          BEGIN
            IF U = 0
            THEN
              BEGIN
                IF VR = BARRIER
                THEN
                  BEGIN
                    VR := DCV;
                    BARRIER := DCV;
                    VERTEX[VR]^.MARK := RIGHT;
                    IF VERTEX[VL]^.FATHER <> 0 THEN
                      VL := VERTEX[VL]^.FATHER
                    ELSE
                      BEGIN
                        CREATEBLOSSUM;
                        BLOSSDONE := TRUE;
                      END
                  END
                ELSE
                  VR := VERTEX[VR]^.FATHER;
              END
            ELSE
              BEGIN
                IF VERTEX[U]^.BLOSSNUM > 0 THEN
                  BEGIN
                    U := BASESTAR(VERTEX[U]^.BLOSSNUM);
                  END;
                IF VERTEX[U]^.MARK = NULL THEN
                  BEGIN
                    VERTEX[U]^.MARK := RIGHT;
                    VERTEX[U]^.FATHER := VR;
                    VR := U;
                  END
                ELSE
                  IF U = VL THEN
                    DCV := U;
              END;
          END;
      END (*RIGHTSIDE*);
```

```
    PROCEDURE LEFTSIDE;
(* PROCEDURE "LEFTSIDE" GROWS THE SUBTREE TLEFT OF "BLOSSAUG"'S       *)
(* DOUBLE DEPTH FIRST SEARCH.                                         *)

      VAR
        U: INTEGER;

      BEGIN (*LEFTSIDE*)
        U := UNUSEDPRED(VL);
        WHILE U = - 1 DO
          BEGIN
            VL := VERTEX[VL]^.FATHER;
            U := UNUSEDPRED(VL);
          END;
        IF NOT BLOSSDONE
        THEN
          BEGIN
            IF U = 0
            THEN
              BEGIN
                IF VERTEX[VL]^.FATHER = 0
                THEN
                  BEGIN
                    CREATEBLOSSUM;
                    BLOSSDONE := TRUE;
                  END
                ELSE
                  VL := VERTEX[VL]^.FATHER
              END
            ELSE
              BEGIN
                IF VERTEX[U]^.BLOSSNUM > 0 THEN
                  U := BASESTAR(VERTEX[U]^.BLOSSNUM);
                IF VERTEX[U]^.MARK = NULL
                THEN
                  BEGIN
                    VERTEX[U]^.MARK := LEFT;
                    VERTEX[U]^.FATHER := VL;
                    VL := U;
                  END
                ELSE
                  IF (U <> BARRIER) OR (U = VR) THEN
                    BEGIN
                      VERTEX[U]^.MARK := LEFT;
                      VR := VERTEX[VR]^.FATHER;
                      VL := U;
                      DCV := U;
                    END;
              END;
          END;
      END (*LEFTSIDE*);
```

```
BEGIN (*BLOSSAUG*)
  BLOSSDONE := FALSE;
  IF (VERTEX[W1]^.BLOSSNUM = VERTEX[W2]^.BLOSSNUM) AND (VERTEX[W1]^
     .BLOSSNUM <> 0)
  THEN
    BLOSSDONE := TRUE
  ELSE
    BEGIN
      FOR I := 1 TO NUMBEROFVERTICES DO
        BEGIN
          VERTEX[I]^.MARK := NULL;
          VERTEX[I]^.FATHER := 0;
        END;
      IF VERTEX[W1]^.BLOSSNUM > 0
      THEN
        VL := BASESTAR(VERTEX[W1]^.BLOSSNUM)
      ELSE
        VL := W1;
      IF VERTEX[W2]^.BLOSSNUM > 0
      THEN
        VR := BASESTAR(VERTEX[W2]^.BLOSSNUM)
      ELSE
        VR := W2;
      VERTEX[VL]^.MARK := LEFT;
      VERTEX[VR]^.MARK := RIGHT;
      DCV := 0;
      BARRIER := VR;
    END;
  IF VR = VL THEN
    BLOSSDONE := TRUE;
(*END BLOSSAUG INITIALIZATION*)
  WHILE NOT BLOSSDONE DO
    BEGIN
      IF VERTEX[VL]^.EXPOSED AND VERTEX[VR]^.EXPOSED
      THEN
        BEGIN
          INCREASEMATCHING;
          AUGMENTED := TRUE;
          BLOSSDONE := TRUE;
        END
      ELSE
        IF LEVEL(VL) >= LEVEL(VR)
        THEN
          LEFTSIDE
        ELSE
          RIGHTSIDE;
    END;
END (*BLOSSAUG*);
```

```
  PROCEDURE INITSEARCH;
(* PROCEDURE "INITSEARCH" INITIALIZES THE VERTICES LABELS AND THE     *)
(* PROGRAMS GLOBAL VARIABLES IN ORDER TO START A NEW PHASE.           *)

    VAR
      I, J: INTEGER;

    BEGIN
      FOR I := 1 TO NUMBEROFVERTICES DO
        BEGIN
          IF VERTEX[I]^.EXPOSED
          THEN
            VERTEX[I]^.EVNLVL := 0
          ELSE
            VERTEX[I]^.EVNLVL := INFINITE;
          VERTEX[I]^.ODDLVL := INFINITE;
          VERTEX[I]^.BLOSSNUM := 0;
          VERTEX[I]^.PREDHEAD := NIL;
          VERTEX[I]^.ANOMHEAD := NIL;
          BRIDGE[I] := NIL;
          VERTEX[I]^.VERVISITED := FALSE;
          VERTEX[I]^.ERASED := FALSE;
        END;
      BRIDGE[0] := NIL;
      FOR I := 1 TO NUMBEROFVERTICES DO
        FOR J := 1 TO NUMBEROFVERTICES DO
          BEGIN
            EDGEVISITED[I, J] := FALSE;
            EDGEUSED[I, J] := FALSE;
          END;
      SEARCHLVL := - 1;
      AUGMENTED := FALSE;
      BLOSSCOUNT := 0;
    END (*INITSEARCH*);
```

```
  PROCEDURE SEARCH;
(* PROCEDURE "SEARCH" PERFORMS A PARALLEL BREADTH FIRST SEARCH IN      *)
(* ORDER TO DISCOVER BRIDGES.  IT ALTERNATELY SEARCHES FOR EVEN AND    *)
(* ODD LEVEL VERTICES, MAKING LISTS OF THEIR PREDECESSORS AND          *)
(* ANOMALIES AS IT GOES.                                               *)

    VAR
      I: INTEGER;


    PROCEDURE INSERT(I: INTEGER; VAR HEADPTR: VPTR);
(* PROCEDURE "INSERT" WILL STACK VERTEX "I" ONTO A LIST WHERE          *)
(* "HEADPTR" POINTS TO THE HEAD OF THE LIST.  "INSERT" IS USED TO ADD  *)
(* ONTO A VERTEX'S LIST OF PREDECESSORS AND ANOMALIES.                 *)

      VAR
        TEMP: VPTR;

      BEGIN
        TEMP := HEADPTR;
        NEW(HEADPTR);
        HEADPTR^.V := I;
        HEADPTR^.NEXTV := TEMP;
      END (*INSERT*);


    PROCEDURE CALLBRIDGES(LEVEL: INTEGER);
(* PROCEDURE "CALLBRIDGES" CALLS PROCEDURE "BLOSSAUG" FOR EVERY        *)
(* BRIDGE FOUND AT THE CURRENT SEARCH LEVEL.                           *)

      VAR
        V1, V2: INTEGER;

      BEGIN
        WHILE BRIDGE[LEVEL] <> NIL DO
          BEGIN
            V1 := BRIDGE[LEVEL]^.B1;
            V2 := BRIDGE[LEVEL]^.B2;
            BRIDGE[LEVEL] := BRIDGE[LEVEL]^.NEXTBRIDGE;
            IF (NOT VERTEX[V1]^.ERASED) AND (NOT VERTEX[V2]^.ERASED)
            THEN
             BLOSSAUG(V1, V2);
          END;
      END (*CALLBRIDGES*);
```

```
    PROCEDURE EVENSEARCH(I: INTEGER);
(* PROCEDURE "EVENSEARCH" SEARCHES FOR VERTICES WHOSE LEVEL EQUALS     *)
(* THE SEARCH LEVEL WHEN THE SEARCH LEVEL IS EVEN.                     *)

      VAR
        APOINTER: VPTR;
        TEMP: INTEGER;

      BEGIN
        APOINTER := VERTEX[I]^.ADJHEAD;
        WHILE APOINTER <> NIL DO
          BEGIN
            IF (NOT EDGEUSED[I, APOINTER^.V]) AND (NOT EDGEMATCHED[I,
               APOINTER^.V])
            THEN
              IF VERTEX[APOINTER^.V]^.EVNLVL < INFINITE
              THEN
                BEGIN
                  TEMP := VERTEX[APOINTER^.V]^.EVNLVL;
                  TEMP := TEMP + VERTEX[I]^.EVNLVL;
                  TEMP := TEMP DIV 2;
                  BRIDGEINSERTION(TEMP, I, APOINTER^.V);
                END
              ELSE
                BEGIN
                  IF VERTEX[APOINTER^.V]^.ODDLVL = INFINITE THEN
                    VERTEX[APOINTER^.V]^.ODDLVL := SEARCHLVL + 1;
                  IF VERTEX[APOINTER^.V]^.ODDLVL = SEARCHLVL + 1 THEN
                    INSERT(I, VERTEX[APOINTER^.V]^.PREDHEAD);
                  IF VERTEX[APOINTER^.V]^.ODDLVL < SEARCHLVL THEN
                    INSERT(I, VERTEX[APOINTER^.V]^.ANOMHEAD);
                END;
            APOINTER := APOINTER^.NEXTV;
          END;
      END (*EVNSEARCH*);
```

```
    PROCEDURE ODDSEARCH(I: INTEGER);
(* PROCEDURE "ODDSEARCH" SEARCHES FOR VERTICES WHOSE LEVEL EQUALS      *)
(* THE SEARCH LEVEL WHEN THE SEARCH LEVEL IS ODD.                      *)
      VAR
        APOINTER: VPTR;
        TEMP: INTEGER;

      BEGIN
        IF VERTEX[I]^.BLOSSNUM = 0 THEN
          BEGIN
            APOINTER := VERTEX[I]^.ADJHEAD;
            WHILE NOT EDGEMATCHED[I, APOINTER^.V] DO
              BEGIN
                APOINTER := APOINTER^.NEXTV;
              END;
            IF VERTEX[APOINTER^.V]^.ODDLVL = SEARCHLVL THEN
              BEGIN
                TEMP := VERTEX[APOINTER^.V]^.ODDLVL;
                TEMP := TEMP + VERTEX[I]^.ODDLVL;
                TEMP := TEMPDIV 2;
                BRIDGEINSERTION(TEMP, I, APOINTER^.V);
              END;
            IF VERTEX[APOINTER^.V]^.ODDLVL = INFINITE THEN
              BEGIN
                VERTEX[APOINTER^.V]^.EVNLVL := SEARCHLVL + 1;
                INSERT(I, VERTEX[APOINTER^.V]^.PREDHEAD);
              END;
      END (*ODDSEARCH*);

    BEGIN (*SEARCH*)
      WHILE NOT AUGMENTED AND NOT DONE DO
        BEGIN
          SEARCHLVL := SEARCHLVL + 1;
          DONE := TRUE;
          IF ODD(SEARCHLVL) THEN
           BEGIN
            FOR I:=1 TO NUMBEROFVERTICES DO
             IF VERTEX[I]^.ODDLVL = SEARCHLVL THEN
              BEGIN
               DONE:=FALSE;
               ODDSEARCH(I);
              END;
           END
          ELSE
           BEGIN
            FOR I:=1 TO NUMBEROFVERTICES DO
             IF VERTEX[I]^.EVNLVL = SEARCHLVL THEN
              BEGIN
               DONE:=FALSE;
               EVENSEARCH(I);
              END;
           END;
          CALLBRIDGES(SEARCHLVL);
        END;
    END (*SEARCH*);
```

```
BEGIN (*MXMATCH*)
  DONE := FALSE;
  INPUTDATA;
  OUTPUTDATA;
  WHILE NOT DONE DO
    BEGIN
      INITSEARCH;
      SEARCH;
    END;
  OUTPUTDATA;
END (*MXMATCH*).
```