Algorithmica (1988) 3: 511-533

Algorithmica
© 1988 Springer-Verlag New York Inc.

The General Maximum Matching Algorithm of Micali and Vazirani[1]

Paul A. Peterson[2,3] and Michael C. Loui[2]

Abstract. We give a clear exposition of the algorithm of Micali and Vazirani for computing a maximum matching in a general graph. This is the most efficient algorithm known for general matching. On a graph with n vertices and m edges this algorithm runs in $O(n^{1/2}m)$ time.

Key Words. Matching, Graph algorithm, Combinatorial optimization.

1. Introduction. Let $G=(V, E)$ be a finite, undirected graph with vertices V and edges E. Let $n=|V|$ and $m=|E|$. A *matching* is a set M of edges of G such that no two edges of M are incident on the same vertex. A *maximum matching* is a matching of maximum cardinality.

The computation of maximum matchings is a fundamental problem of combinatorial optimation. The classic assignment problem of operations research can be formulated as a matching problem on bipartite graphs (Bondy and Murty, 1976). The scheduling of tasks of multiprocessor computers (Fujii *et al.*, 1969) and the scheduling of transmissions on packet radio networks (Hajek, 1984) can be modeled as matching problems on general graphs.

For bipartite graphs the best maximum matching algorithm, due to Hopcroft and Karp (1973), runs in $O(n^{1/2}m)$ time. For general graphs a straightforward implementation of the maximum matching algorithm of Edmonds (1965) runs in $O(n^4)$ time (Papadimitriou and Steiglitz, 1982). Progressively more efficient general matching algorithms have been designed with the following running times:

$O(n^3)$	(Gabow, 1976),
$O(nm)$	(Kameda and Munro, 1974),
$O(n^{5/2})$	(Even and Kariv, 1975),
$O(n^{1/2}m)$	(Micali and Vazirani, 1980).

We give a clear exposition of the important—but extremely complicated—algorithm of Micali and Vazirani, which is the most efficient known for maximum matching in general graphs. The efficiency of this algorithm depends on the new algorithm for incremental tree set union of Gabow and Tarjan (1985). We hope

[1] Work on this paper has been supported by the Office of Naval Research under Contract N00014-85-K-0570 and by the Eastman Kodak Company.
[2] Department of Electrical and Computer Engineering and Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA.
[3] Current address: NCR Corporation, 1601 South Main Street, Dayton, OH 45479, USA.

Received June 13, 1986; revised July 9, 1987. Communicated by C. K. Wong.

that this exposition will reach a wider audience than did the original paper (Micali and Vazirani, 1980), which appeared only in a conference proceedings.

We assume that the reader understands basic graph theory (Bondy and Murty, 1976) and the breadth-first search and depth-first search techniques. Section 2 reviews basic definitions for matching. Section 3 gives an overview of the algorithm of Micali and Vazirani. Sections 4–6 describe the principal subroutines of the algorithm in detail, assuming that they encounter no blossoms. Section 7 modifies the subroutines to handle blossoms. The presence of blossoms complicates general matching algorithms. Section 8 establishes that the algorithm runs in $O(n^{1/2}m)$ time. The appendix presents the entire algorithm formally; it corrects several minor errors in the original paper of Micali and Vazirani (1980).

2. Basic Definitions.

Let M be a matching in a graph. With respect to M we define the terms *matched, free, mate, exposed, alternating path,* and *augmenting path.*

An edge e is *matched* if $e \in M$, *free* if $e \notin M$. A vertex v is *matched* if some matched edge is incident on v, *exposed* if no matched edge is incident on v. If $(v, w) \in M$, then v is the *mate* of w, and vice versa. (The pairs (v, w) and (w, v) denote the same edge.)

A *blossom* is a circuit of odd length, say $2k+1$, that has k matched edges. Since bipartite graphs have no circuits of odd length, bipartite matching algorithms need not handle blossoms.

A path $(v_1, v_2, \ldots)$ is *alternating* if the edges $(v_1, v_2), (v_2, v_3), \ldots$ are alternately in M and not in M. For vertices v and w, let $Length(v, w)$ be the length of the shortest alternating path from v to w. A path is *augmenting* if it is alternating, and the first and last vertices are exposed. An augmenting path always has odd length. An augmenting path with $2k+1$ edges has k matched edges and $k+1$ free edges.

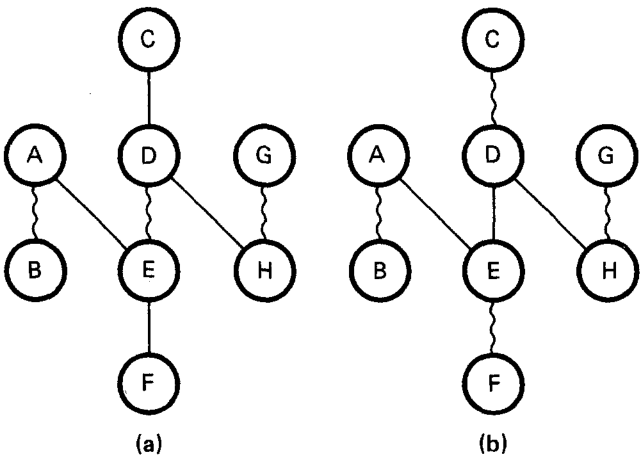

Fig. 1

Let $E(P)$ be the edges of an augmenting path P for M. The symmetric difference

$$M' = E(P) \oplus M$$

comprises the edges in $E(P)$ or in M but not in both. M' is a matching with one more edge than M. Call M' the result of *increasing M along P*. In Figure 1 straight lines denote free edges, wavy lines matched edges. Increasing the matching along augmenting path (C, D, E, F) in Figure 1(a) yields the new matching of Figure 1(b).

From this discussion it is apparent that if M has an augmenting path, then M is not maximum. Conversely, Berge (1957) established that if M has no augmenting paths, then M is maximum. Bondy and Murty (1976) and Papadimitriou and Steiglitz (1982) gave clear proofs of this fundamental fact.

3. Overview of the Algorithm

3.1. Phases. To obtain a maximum matching, the algorithm proceeds in a sequence of *phases*. At the beginning of each phase the algorithm has a matching M. During each phase the algorithm finds a maximal set of vertex-disjoint minimum length augmenting paths for M and increases the matching along each of these paths. Hopcroft and Karp (1973) proved that $O(n^{1/2})$ of these phases suffice for finding a maximum matching, even in nonbipartite graphs. Each phase of the algorithm runs in $O(m)$ time. Consequently, the running time of the algorithm is $O(n^{1/2}m)$.

3.2. Definitions. Fix a matching M in the graph. The *even-level* of a vertex v is the length of the minimum even length alternating path for M from an exposed vertex to v, if any, $+\infty$ otherwise. Write $EvenLevel(v)$ for the even-level of v. The *odd-level* of a vertex v is the length of the minimum odd length alternating path for M from an exposed vertex to v, if any, $+\infty$ otherwise. Write $OddLevel(v)$ for the odd-level of v. Define the *level* of v to be

$$Level(v) = \min\{EvenLevel(v), OddLevel(v)\}.$$

Vertex v is *outer* if $Level(v)$ is even, *inner* if $Level(v)$ is odd. Observe that if v is exposed, then $Level(v) = EvenLevel(v) = 0$, hence v is outer. If v is outer (resp. inner), then the *other-level* of v is its odd-level (even-level).

An edge (s, t) is a *bridge* if both $EvenLevel(s)$ and $EvenLevel(t)$ are finite, or if both $OddLevel(s)$ and $OddLevel(t)$ are finite. (We caution the reader than our use of the word *bridge* differs from its use in pure graph theory, namely, an edge whose removal leaves a disconnected graph.) Since every augmenting path P has odd length, every edge in P is a bridge. Furthermore, if M has no blossoms, then every bridge lies on an augmenting path. Define

$$tenacity(s, t) = \min\{EvenLevel(s) + EvenLevel(t),$$
$$OddLevel(s) + OddLevel(t)\} + 1.$$

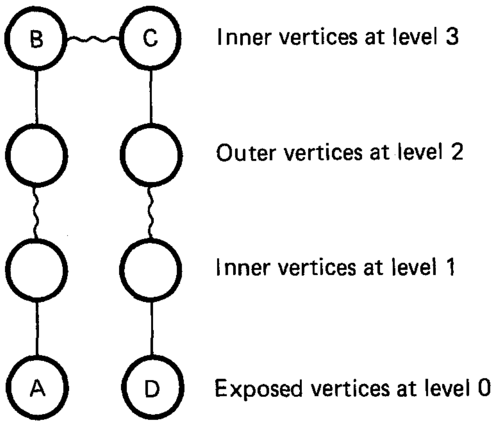
B
C
A
D
Inner vertices at level 3
Outer vertices at level 2
Inner vertices at level 1
Exposed vertices at level 0

Fig. 2

The length of the minimum length augmenting path containing a bridge (s, t) is $tenacity(s, t)$.

3.3. Description of Subroutines. To explain the fundamental ideas of the algorithm clearly, we assume here and in Sections 4–6 that no matching constructed by the algorithm has blossoms. Section 7 presents the modifications to handle blossoms.

The algorithm has three principal subroutines: SEARCH, BLOSS-AUG, and FINDPATH. This description will refer to Figure 2.

During each phase SEARCH conducts a breadth-first search for augmenting paths. It searches simultaneously from all exposed vertices until it discovers a nonempty set $Bridges(i)$ of bridges at search level i. In Figure 2 SEARCH would start from the exposed vertices A and D to find the bridge (B, C) at search level 3. For every bridge in $Bridges(i)$, SEARCH calls BLOSS-AUG with the bridge as the input.

BLOSS-AUG performs a double depth-first search in order to find the exposed vertices of an augmenting path containing the bridge. In Figure 2 BLOSS-AUG would perform a depth-first search from B to find A concurrently with a depth-first search from C to find D. After finding exposed vertices A and D, BLOSS-AUG calls FINDPATH twice, once with inputs B and A and once with inputs C and D.

FINDPATH performs a simple depth-first search to find the exact vertices of the alternating paths between its two input vertices. The augmenting path is the concatenation of the two paths found by FINDPATH.

For each augmenting path found by BLOSS-AUG and FINDPATH, the algorithm increases the matching along the augmenting path and marks these vertices "erased." Since BLOSS-AUG considers only unerased vertices, all augmenting paths discovered during the same phase are vertex-disjoint. SEARCH continues to pass bridges to BLOSS-AUG until $Bridges(i)$ is exhausted. Then the phase ends.

This algorithm borrows some ideas from the algorithm of Even and Kariv (1975), which also operates in phases. SEARCH resembles the first stage of the algorithm of Even and Kariv, and the double depth-first search of BLOSS-AUG corresponds to the third stage.

4. Subroutine SEARCH

4.1. Finding a Bridge. At the beginning of each phase the algorithm erases all marks and labels used during the preceding phase and initializes the even-level and odd-level of every vertex to $+\infty$, signifying that it has not yet found any alternating path. Then it sets the even-level of every exposed vertex to 0 and starts SEARCH with search level 0.

SEARCH conducts a breadth-first search for augmenting paths, starting simultaneously from all exposed vertices at search level 0. In general, SEARCH finds vertices at level $i+1$ only after it finds all vertices at level i. SEARCH scans each edge at most twice, once in each direction. We describe this process in detail.

When the search level i is even, SEARCH considers each vertex v such that $EvenLevel(v) = i$. For each vertex u adjacent to v such that (u, v) is free, if $OddLevel(u) = +\infty$, then SEARCH sets $OddLevel(u) = i+1$. When the search level i is odd, SEARCH considers each matched vertex v such that $OddLevel(v) = i$. Let u be the mate of v. SEARCH sets $EvenLevel(u) = i+1$.

Figure 3 shows the state of SEARCH, which started from exposed vertices A, B, and C simultaneously, after search level 2. Vertices are labeled with their current values of *EvenLevel* and *OddLevel*. SEARCH will examine vertices H, I, J, and K at search level 3.

For each edge (u, v) encountered at the current search level i, SEARCH determines whether (u, v) is a bridge by comparing the even-level and odd-level of both u and v. If (u, v) is a bridge, then it is added to the set $Bridges(i)$ of

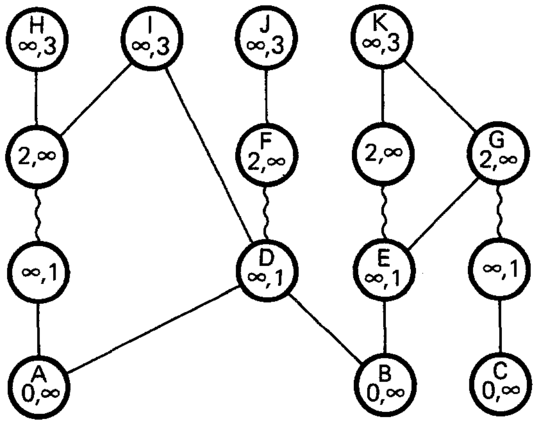

Fig. 3

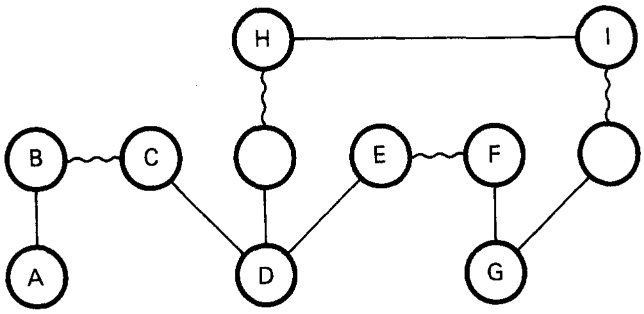

Fig. 4

bridges found at search level i. If SEARCH encounters no bridges at search level i, then it begins search level $i+1$.

Unless SEARCH encounters a blossom, considered in Section 7, the discovery of a bridge yields an augmenting path. If at the end of search level i the set *Bridges*(i) is nonempty, then for each bridge (s, t) in *Bridges*(i) in succession, SEARCH calls BLOSS-AUG with input (s, t). After these calls to BLOSS-AUG the current phase ends; SEARCH does not proceed to search level $i+1$ during the current phase.

If at the start of a phase the current matching is maximum, then no augmenting paths exist. SEARCH recognizes this condition when it reaches search level i and finds no vertices at level i.

In Figure 4 SEARCH discovers bridges (B, C) and (E, F) at search level 1. Edge (H, I) will not be a bridge in the current phase because the phase terminates after increasing the matching along the augmenting paths through bridges at search level 1. Suppose SEARCH calls BLOSS-AUG first with (B, C), then with (E, F). If the matching is increased along the path (A, B, C, D), then (D, E, F, G) would no longer be an augmenting path. Thus SEARCH could pass a bridge to BLOSS-AUG that is not a part of an augmenting path. Section 5.3 discusses this situation further.

4.2. Predecessors and Anomalies. During the execution of SEARCH, call vertex y a *predecessor* of a matched vertex z if y is adjacent to z and either

(1) z is inner, and $OddLevel(z) = EvenLevel(y) + 1$; or
(2) z is outer, and y is the mate of z.

If SEARCH first discovered z by examining the neighbors of y, then y is a predecessor of z. Call (y, z) a *predecessor edge* of z. Let *Predecessors*(z) denote the set of predecessors of vertex z. SEARCH inserts y into *Predecessors*(z) when it scans edge (y, z). Vertex w is an *ancestor* of z if either w is a predecessor of z or w is a predecessor of another ancestor of z.

Vertex y is an *anomaly* for vertex z if z is inner, y is outer, y is adjacent to z, but y is not the mate of z, and $EvenLevel(y) > OddLevel(z)$. Let *Anomalies*$(z)$ denote the set of anomalies of z. SEARCH adds y to the set *Anomalies*(z) when it reaches y. Informally, y is an anomaly for z because when SEARCH considers the neighbors of y during the breadth-first search, z already has a finite value for odd-level. Section 7 will use anomalies to construct augmenting paths through blossoms.

In Figure 3 A and B are predecessors of D, D is the predecessor of F, and G is an anomaly of E.

5. Subroutine BLOSS-AUG

5.1. Depth-First Search Processes. The input to BLOSS-AUG is a bridge (s, t). BLOSS-AUG finds the exposed vertices x and x' of an augmenting path containing the bridge. If (s, t) was discovered by SEARCH at level i, then

$$Length(s, x) + Length(t, x') = 2i,$$

and the augmenting path has length $2i+1$.

To find x and x', BLOSS-AUG performs two depth-first searches concurrently, starting from s and t simultaneously, using the predecessors of vertices: the left depth-first search process LEFTDFS starts from s; the right depth-first search process RIGHTDFS starts from t. Lets variables v_L and v_R denote the current vertices of LEFTDFS and RIGHTDFS, respectively. Initially $v_L = s$ and $v_R = t$. Note that initially $Level(v_L) = Level(v_R)$.

In general, LEFTDFS proceeds if $Level(v_L) \geq Level(v_R)$, and RIGHTDFS proceeds otherwise. LEFTDFS examines only the predecessor edges of v_L that have not yet been used. When LEFTDFS selects a predecessor u of v_L, it defines $Parent(u) = v_L$, marks edge (v_L, u) "used," marks vertex u "left," and sets $v_L = u$. By definition of the predecessors, the level of v_L is smaller than before. RIGHTDFS proceeds in the same manner, marking the chosen predecessor vertex "right" instead of "left." If LEFTDFS and RIGHTDFS reach two different exposed vertices, then BLOSS-AUG calls FINDPATH, which constructs the complete augmenting path.

5.2. Backtracking. LEFTDFS and RIGHTDFS may meet at a vertex w. Only one process may claim w and the exposed vertex reachable from w. First, LEFTDFS claims w and marks w "left." Then RIGHTDFS tries to find a vertex as deep as w, backtracking via *Parent* if necessary. If RIGHTDFS fails to find another vertex as deep as w, then RIGHTDFS claims w and replaces the "left" mark on w by "right." Now LEFTDFS backtracks via *Parent* and tries to find a vertex as deep as w. If LEFTDFS cannot find another vertex as deep as w, then a blossom has been discovered. This situation is handled in Section 7.

LEFTDFS and RIGHTDFS use variables *DCV* (Deepest Common Vertex) and *Barrier*. *DCV* is the deepest vertex—i.e., the vertex with the smallest level—discovered by both LEFTDFS and RIGHTDFS. Before the first time that a common vertex is reached, *DCV* is undefined. If a blosson is discovered, then *DCV* will be the base.

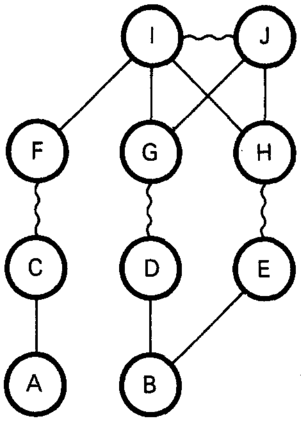

Fig. 5

Barrier prevents unproductive backtracking to keep the time for one phase to $O(m)$. Suppose LEFTDFS and RIGHTDFS meet at vertex w. Furthermore, suppose RIGHTDFS fails to find another vertex as deep as w, but LEFTDFS does. Subsequently LEFTDFS and RIGHTDFS meet again. At this time RIGHTDFS should not back up above w. In general, when RIGHTDFS fails during backtracking, *Barrier* is set to the current *DCV*. Later backtracking by RIGHTDFS never passes above *Barrier*. Initially $Barrier = v_R$.

Let us examine Figure 5. Suppose BLOSS-AUG is called with input(I, J). The following actions could occur:

(0) LEFTDFS sets $v_L = I$ and marks I "left"; RIGHTDFS sets $v_R = J$ and marks J "right." Initially $Barrier = J$.
(1) LEFTDFS uses (I, G), marks G "left," and sets $v_L = G$.
(2) RIGHTDFS uses (J, H), marks H "right," and sets $v_R = H$.
(3) LEFTDFS uses (G, D), marks D "left," and sets $v_L = D$.
(4) RIGHTDFS uses (H, E), marks E "right," and sets $v_R = E$.
(5) LEFTDFS uses (D, B), marks B "left," and sets $v_L = B$.
(6) RIGHTDFS uses (E, B), finds $B = v_L$, and sets $DCV = B$.
(7) Since E now has no unused predecessor edges, RIGHTDFS backtracks to H, which also has no unused predecessor edges, then to J. At this point $v_R = J$.
(8) RIGHTDFS uses (J, G), but G is marked "left."
(9) Since $v_R = J = Barrier$ and J has no unused predecessor edges, RIGHTDFS claims *DCV* by setting $v_R = DCV = B$, marks B "right," and forces LEFTDFS to backtrack by setting $v_L = D$. Also, RIGHTDFS sets $Barrier = DCV = B$.
(10) Since D now has no unused predecessor edges, LEFTDFS backtracks to G, which also has no unused predecessor edges, then to I. At this point $v_L = I$.
(11) LEFTDFS uses (I, H), but H is marked "right."
(12) LEFTDFS uses (I, F), marks F "left," and sets $v_L = F$. Eventually LEFTDFS reaches A.

(13) BLOSS-AUG calls FINDPATH to construct the augmenting path from exposed vertex A through bridge (I, J) to exposed vertex B.

If the graph did not have edge (I, F), then at step (12) LEFTDFS would have exhausted all predecessor edges of I, and LEFTDFS would have detected a blossom.

5.3. Erasure. Figure 4 shows that SEARCH may pass to BLOSS-AUG a bridge that is no longer part of an augmenting path. To prevent BLOSS-AUG from searching for a nonexistent augmenting path, the algorithm erases the vertices of each augmenting path as soon as the matching is increased along the path, and LEFTDFS and RIGHTDFS select only unerased vertices. Thus the erasures guarantee the disjointness of the minimum length augmenting paths during one phase. The erasure of a vertex occurs at most once during each phase. All "erased" marks are removed at the start of the next phase.

Subroutine ERASE marks every vertex on an augmenting path "erased." Also, ERASE marks vertex z "erased" if every vertex in $Predecessors(z)$ is marked "erased." For speed, ERASE does not access $Predecessors(z)$ when a predecessor of z is erased. Instead, each vertex z has an integer variable $Count(z)$ whose value is the number of unerased predecessors of z, and ERASE subtracts 1 from $Count(z)$ when it erases a predecessor of z. When ERASE changes $Count(z)$ to 0, it also erases z.

Upon completion of ERASE, every unerased, matched vertex z (with finite $Level(z)$) has at least one unerased predecessor, hence has an alternating path to an exposed vertex. Consequently, if BLOSS-AUG is called with input (s, t), then there is an alternating path from s to an exposed vertex and an alternating path from t to an exposed vertex, but these paths may intersect. By Menger's Theorem (Bondy and Murty, 1976), vertex-disjoint alternating paths exist if and only if for each $i \leq \min\{Level(s), Level(t)\}$, there are at least two distinct ancestors of s or t at level i. Thus if LEFTDFS is unable to find another vertex as deep as DCV, then there is only one ancestor of s or t at $Level(DCV)$, and BLOSS-AUG cannot find an augmenting path through (s, t).

6. Subroutine FINDPATH.

After BLOSS-AUG has discovered the exposed vertices x and x' of the augmenting path containing the bridge (s, t), it calls subroutine FINDPATH twice: once to find the vertices of the alternating path from s to x, once to find the alternating path from t to x'. The concatenation of these two paths with (s, t) produces the augmenting path. If BLOSS-AUG encountered no blossoms, then LEFTDFS and RIGHTDFS have already traversed the desired paths, and FINDPATH would be superfluous. If BLOSS-AUG encountered blossoms, however, then the modifications of Section 7.6 imply that LEFTDFS and RIGHTDFS may not have traversed the desired paths.

The inputs to FINDPATH are vertices $High$ and Low, with $Level(High) \geq Level(Low)$. FINDPATH performs a depth first search from $High$ to find Low. When FINDPATH reaches a vertex for the first time during the phase, it marks the vertex "visited." If the current vertex is v, then FINDPATH considers only the unvisited, unerased predecessors of v to continue the search. FINDPATH selects a predecessor u of v only if the "left" or "right" mark of u is the same as that of *High* and $Level(u) \geq Level(Low)$. Upon selecting u, FINDPATH sets $Parent(u) = v$. If $Parent(u)$ was defined by LEFTDFS or RIGHTDFS, then $Parent(u) = v$ already. When FINDPATH reaches *Low*, FINDPATH uses the *Parent* values to construct the path.

Section 7.7 modifies FINDPATH to handle blossoms.

7. Blossoms and Blooms.

So far we have ignored the occurrence of blossoms. This section describes the discovery, labeling, and opening of blossoms.

7.1. Blossoms and Phases. Every bridge discovered by SEARCH lies on either an augmenting path or a blossom. BLOSS-AUG not only finds the exposed vertices of an augmenting path but also detects the blossoms. If BLOSS-AUG determines that a bridge corresponds to a blossom, then it labels the vertices of the blossom and exits.

The discovery of blossoms does not end a phase. If all bridges at search level i correspond to blossoms, then the search level is incremented. The current phase does not end until either at least one augmenting path is found or the matching is maximum.

7.2. Blooming Condition. The algorithm uses a generalization of blossoms that we call *blooms*. Section 7.5 exhibits blooms that are not blossoms. Let (s, t) be a bridge.

Blooming Condition. There exists a vertex w such that w is an ancestor of both s and t, and no other ancestor of either s or t has the same level as w.

Among the w's of the Blooming Condition such that w does not currently belong to a bloom, let b be the vertex whose level is maximum. The *bloom* B is the set of vertices y such that:

(1) y does not belong to any other bloom when B is formed,
(2) either $y = s$ or $y = t$ or y is an ancestor of s or of t, and
(3) b is an ancestor of y.

Observe that by condition (1), a vertex can belong to at most one bloom. By condition (3), $b \notin B$. Call b the *base* of B, and write $b = base(B)$. Call s and t the *peaks* of B. Figure 6(a) shows a bloom $\{D, E, F, G\}$ whose peaks are F and G and whose base is C.

By induction on the number of blooms that have been formed, we show that for every matched edge (y, z), if y belongs to a bloom B, then $z \in B$ too. Consider the situation when B is formed, and let $b = base(B)$. By condition (1) and the inductive hypothesis, z is not yet in a bloom, and z satisfies condition (1). Because (y, z) is matched, either y is the only predecessor of z or vice versa. It follows that since y satisfies condition (2), z also satisfies condition (2). By the inductive

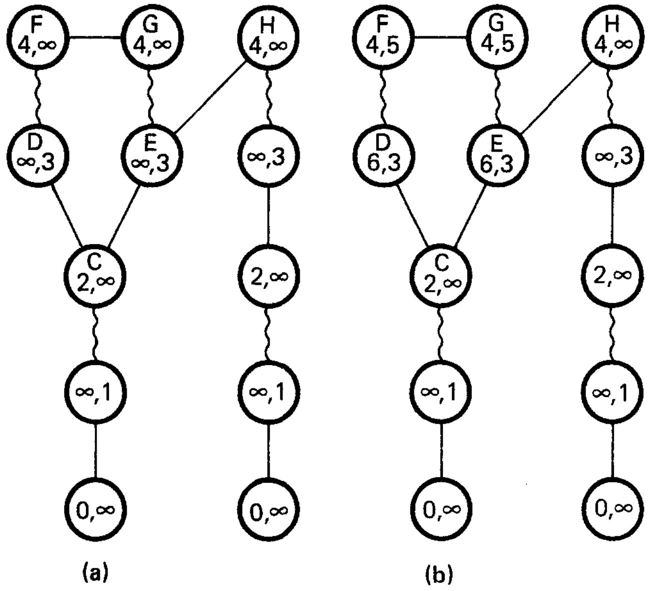

Fig. 6

hypothesis again, since currently b is not in a bloom, the mate of b (if it exists) is not in a bloom. Consequently b must be outer, for otherwise its mate would be an ancestor of both s and t at a higher level. If z is inner, then $b \neq z$, and condition (3) implies that b is also an ancestor of z. If z is outer, then y is the predecessor of z, and condition (3) implies that b is also an ancestor of z. Thus z satisfies condition (3). *Ergo* $z \in B$.

7.3. Detection. Section 5.2 mentioned that BLOSS-AUG detects a bloom when the depth-first search processes LEFTDFS and RIGHTDFS meet at a vertex w such that neither LEFTDFS nor RIGHTDFS can find a different vertex as deep as w.

Upon detecting a bloom B, the algorithm determines the set of vertices in B. B comprises all vertices marked "left" or "right," excluding the *DCV*. The base of B is the *DCV*. Section 7.6 modifies LEFTDFS and RIGHTDFS to ensure that neither the vertices of B nor $base(B)$ belong to a previously formed bloom.

7.4. Setting Other-level. Let y be a vertex in bloom B. If y is inner (outer), then there is an even (odd) length alternating path containing (s, t) from an exposed vertex to y. Thus the other-level of y can be set to

$$tenacity(s, t) - Level(y).$$

Figure 6 shows a bloom—*a fortiori* a blossom—discovered at search level 4 before and after the other-level of each bloom vertex is set; each vertex is labeled with its even-level and odd-level.

Once BLOSS-AUG has set the other-level of each vertex in B, it must check for newly formed bridges. Any new bridge must have at least one vertex in B. For new bridges (y, z) such that both y and z belong to B, the Blooming Condition holds. Thus (y, z) induces not an augmenting path, but a bloom already contained in B. These bridges are ignored. For new bridges (y, z) such that only vertex y is in B, necessarily y is inner. If $EvenLevel(z) < OddLevel(y)$, then z is a predecessor of y, but since the depth-first search processes did not find an augmenting path through z, bridge (y, z) is ignored. If $EvenLevel(z) > OddLevel(y)$, then z is an anomaly of y. In Figure 6(b) (E, H) is a bridge and H is an anomaly of E. Conversely, for every anomaly z of an inner vertex y of B, the edge (y, z) is a bridge. BLOSS-AUG computes $tenacity(y, z) = 2k+1$ and inserts (y, z) into the set $Bridges(k)$. If the present search level is i, then $k > i$ because

$$\begin{aligned} 2k &= EvenLevel(z) + EvenLevel(y) \qquad \text{since } z \text{ is an anomaly of } y \\ &> EvenLevel(y) + OddLevel(y) = 2i+1. \end{aligned}$$

If SEARCH reaches level k, then it will call BLOSS-AUG with input (y, z).

After BLOSS-AUG has set the other-level of the vertices in B, SEARCH can continue the breadth-first search from the inner vertices y of B when it reaches search level $EvenLevel(y)$. Since y is inner, SEARCH does not scan the free edges incident on y until it reaches search level $EvenLevel(y)$. In Figure 7 a bloom is discovered at search level 1, and a bridge of an augmenting path is discovered at search level 3.

7.5. Example. Blooms detected by the algorithm depend on the order in which SEARCH passes bridges to BLOSS-AUG. In Figure 8 SEARCH has discovered bridges (K, L) and (L, M) at search level 6. (The figure omits other edges incident on P.)

If BLOSS-AUG handles (K, L) before (L, M), then the blooms are

$$B_1 = \{K, L, H, I\}$$

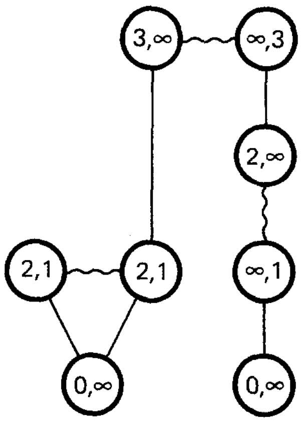

Fig. 7

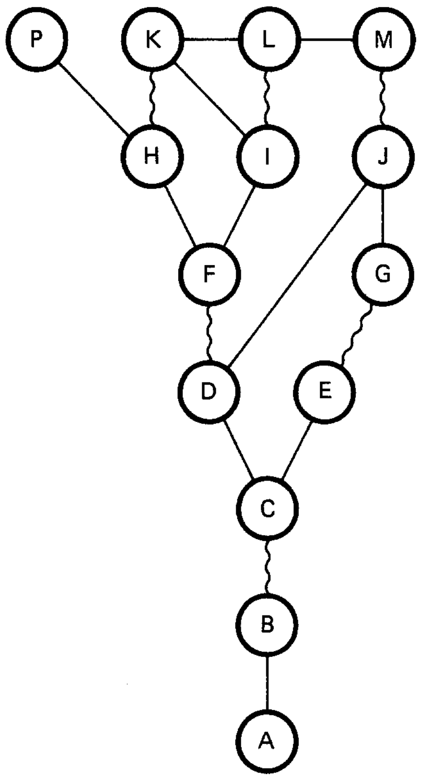

Fig. 8

with peaks K and L and base F and

$$B_2 = \{M, J, F, G, D, E\}$$

with peaks L and M and base C. Notice that a peak of a bloom may not belong to that bloom.

If BLOSS-AUG handles (L, M) before (K, L), then the blooms are

$$B_1' = \{L, M, I, J, F, G, D, E\}$$

with peaks L and M and base C and

$$B_2' = \{K, H\}$$

with peaks K and L and base C.

7.6. Embedded Blooms. An *embedded bloom* is a bloom whose base belongs to another bloom. In Section 7.5 bloom B_1 is embedded in bloom B_2. For blooms B and B', define the partial order $<$ by

$$\mathit{base}(B) < \mathit{base}(B') \qquad \text{if} \quad \mathit{base}(B) \in B'.$$

Let $<^*$ be the reflexive, transitive closure of $<$. Define $base^*(B)$ to be the $base(B^*)$ such that $base(B) <^* base(B^*)$, and $base(B^*)$ does not belong to a bloom.

When LEFTDFS or RIGHTDFS advances to a predecessor that belongs to a bloom B, it changes its current vertex (v_L or v_R) to $base^*(B)$ immediately. This operation guarantees that no vertex considered by LEFTDFS or RIGHTDFS belongs to a previously formed bloom. In essence, this operation has the effect of "shrinking" each bloom into its *base**. There is an augmenting path for the current matching if and only if there is an augmenting path after a blossom is shrunk (Papadimitriou and Steiglitz, 1982; Tarjan, 1983).

To compute *base** we may apply the standard algorithm for the UNION-FIND problem, which uses both path compression and weighted union (Aho *et al.*, 1974; Reingold *et al.*, 1977). For m operations (one for each edge) on n elements (vertices), this algorithm takes $O((m+n)\alpha(m, n))$ time, where α is a very slowly growing function (Tarjan, 1975); in particular, $\alpha(m, n) \leq \log^* n$ for all m and n. For computing *base**, however, we may use the more efficient incremental set union algorithm of Gabow and Tarjan (1985) since the only union operation requires adding a new bloom base. This algorithm runs in $O(m+n)$ time.

7.7. Opening a Bloom. We modify FINDPATH to handle blooms. In general, the inputs to FINDPATH are vertices *High* and *Low* and a bloom *B*. when BLOSS-AUG calls FINDPATH, *B* is undefined. Subsequent recursive calls to FINDPATH will request paths through bloom *B*.

First, FINDPATH finds a sequence of vertices

$$HIGH = x_1, \ldots, x_k = Low$$

from *High* to *Low* via *Predecessors*, assuming that all blooms other than *B* are shrunk. That is, when FINDPATH encounters a vertex $x_j \in B'$, where bloom $B' \neq B$, it jumps to $x_{j+1} = base(B')$. Otherwise, as in Section 6, if $x_j \in B$ or x_j is in no bloom, then FINDPATH selects the predecessor x_{j+1} of x_j having the same "left" or "right" mark as *High*. This sequence of vertices may not be an alternating path.

Second, for each $x_j \in B' \neq B$, FINDPATH calls a new subroutine OPEN to find a path in B' from x_j to x_{j+1}. If x_j is outer, then OPEN calls FINDPATH with inputs x_j, x_{j+1}, B'. If x_j is inner, then OPEN calls FINDPATH twice. Suppose that x_j was marked "left" when B' was formed. The first call to FINDPATH finds a path P_1 from the left peak of B' to x_j, and the second call finds a path P_2 from the right peak of B' to x_{j+1}. Then the path from x_j to x_{j+1} is the concatenation of the reversal of P_1 with P_2.

In Figure 8 bloom $B_1 = \{K, L, H, I\}$ is embedded in bloom $B_2 = \{M, J, F, G, D, E\}$, $base(B_1) = F$, and $base(B_2) = C$. Suppose BLOSS-AUG calls FINDPATH with $High = P$, $Low = A$, and $B =$ empty. FINDPATH first obtains the sequence (P, H, F, C, B, A). Since $H \in B_1$, FINDPATH calls OPEN; since H is inner, OPEN delivers the path (H, K, L, I, F). Next, since $F \in B_2$, FINDPATH calls OPEN; since F is outer, OPEN delivers the path (F, D, C). Upon completion, the original call to FINDPATH produces the path $(P, H, K, L, I, F, D, C, B, A)$.

8. Analysis

8.1. Correctness. Let us verify informally that the algorithm is correct. Consider the situation at the beginning of a phase. If the current matching is maximum, then there is no augmenting path, hence during this phase SEARCH finds no bridges that yield augmenting paths, and the algorithm halts. If the current matching is not maximum, then an augmenting path of some minimum length L exists. It suffices to show that the algorithm finds a maximal set of vertex-disjoint augmenting paths of length L during this phase.

First, we proceed by induction to confirm that SEARCH and BLOSS-AUG compute the even-level and odd-level of each vertex correctly.

Suppose that at an even search level i, SEARCH encounters a vertex u adjacent to a vertex v at level i such that $OddLevel(u)=+\infty$. (The argument for an odd i is identical.) By the inductive hypothesis, i is the length of the shortest alternating path of even length from v to an exposed vertex. Consequently, the shortest odd length alternating path from u to an exposed vertex must have length $i-1$ or $i+1$. If the length were $i-1$, then SEARCH would have set $OddLevel(u)$ at level $i-1$ because the search is breadth first. Therefore SEARCH sets $OddLevel(u)$ to $i+1$ correctly. Furthermore, since every u with $OddLevel(u)=i+1$ is adjacent to a v with $EvenLevel(v)=i$, SEARCH reaches all vertices at level $i+1$ whose odd-level was $+\infty$.

BLOSS-AUG sets the other-level of every vertex in a bloom. Suppose y is an outer vertex in a bloom discovered from a bridge at level k. Then BLOSS-AUG sets $OddLevel(y)$ to $2k+1-EvenLevel(y)$ because there is an alternating path of this odd length from y around the bloom to an exposed vertex. If this value were incorrect, then $OddLevel(y)$ would be $2j+1-EvenLevel(y)$ for some $j<k$. Thus y would have been an ancestor of the vertices of a bridge (s, t) discovered at previous search level j. If (s, t) induced an augmenting path, then we would have ended this phase without reaching search level k. If (s, t) induced a bloom, then BLOSS-AUG would have set $OddLevel(y)$ at level j. Therefore BLOSS-AUG sets $OddLevel(y)$ correctly. Analogously, if y is an inner vertex in a bloom, then BLOSS-AUG sets $EvenLevel(y)$ correctly.

Second, we check that during this phase, the algorithm finds at least one augmenting path of length L.

Since the algorithm determines the vertex levels correctly, for every augmenting path of length L, SEARCH finds a bridge of the path at search level $(L-1)/2$. Let (s, t) be the first bridge of an augmenting path with which SEARCH calls BLOSS-AUG. By Menger's theorem, since there are vertex-disjoint alternating paths from s to an exposed vertex and from t to an exposed vertex, at each level $i\leq\min\{Level(s), Level(t)\}$, there are at least two distinct ancestors of s or t at level i. We must show that LEFTDFS and RIGHTDFS eventually reach different exposed vertices, for then FINDPATH can construct the augmenting path.

Let us examine the operation of LEFTDFS and RIGHTDFS. LEFTDFS (resp. RIGHTDFS) inspects each predecessor u of v_L (v_R). If u belongs to a bloom B, then $base^*(B)$ is the only ancestor of u whose level is $Level(base^*(B))$; thus the shortest alternating path from u to an exposed vertex must pass through $base^*(B)$, and LEFTDFS (RIGHTDFS) may replace u by $base^*(B)$ in its search. Next, LEFTDFS sets v_L (RIGHTDFS sets v_R) to u only if u has neither a "left" nor a "right" mark. If u is marked "left" or "right," then either u is on a current alternating path from s to v_L or from t to v_R, or, since the search processes are depth first, all predecessor edges of u have been used. In the latter case, the only ancestor of u at level $Level(DCV)$ is DCV, which is marked "right" (resp. "left"). Because LEFTDFS (RIGHTDFS) seeks a different ancestor at this level, it need not consider u again. Since there are two or more distinct ancestors of s or t at every level, we conclude that LEFTDFS and RIGHTDFS can find shortest vertex-disjoint alternating paths from s and t to distinct exposed vertices.

Third, we establish that the algorithm finds a maximal set of vertex-disjoint augmenting paths of length L at this phase.

Clearly, the "erased" marks on vertices ensure that the augmenting paths in this phase are vertex-disjoint. To verify that the set of vertex-disjoint augmenting paths is maximal, recall that for every augmenting path of length L, SEARCH finds a bridge of the path and calls BLOSS-AUG with that bridge. LEFTDFS and RIGHTDFS ignore a vertex u only when either u is erased or u already has a "left" or "right" mark. But if u is erased, then u no longer has an alternating path P via a predecessor to an exposed vertex such that P is disjoint from augmenting paths previously discovered during the phase. And if u has a "left" or "right" mark, then u was marked during the current invocation of BLOSS-AUG (to be shown in the next paragraph), and as argued before, u does not lead to a different ancestor at the level of the current DCV. *Ergo*, by the correctness of LEFTDFS and RIGHTDFS, the algorithm determines a maximal set of vertex-disjoint augmenting paths during the phase.

We demonstrate that the only unerased vertices marked "left" or "right" encountered by LEFTDFS and RIGHTDFS during an invocation of BLOSS-AUG are marked during the same invocation. Consider any previous invocation of BLOSS-AUG, and suppose (s, t) was the input. If this invocation yielded a bloom B, then every vertex y marked "left" or "right" during this invocation became part of B; subsequently, LEFTDFS and RIGHTDFS always bypass y when they replace y by $base^*(B)$. If this invocation yielded an augmenting path, then for every vertex y marked "left" or "right" during this invocation, either (1) y was on the augmenting path, or (2) y was marked "left" and was not on the augmenting path, or (3) y was marked "right" and was not on the augmenting path. In case (1) y was erased at the end of the invocation of BLOSS-AUG. In case (2) LEFTDFS backtracked above y because, at some time during its search, the only ancestor of y at level $Level(DCV)$ was $DCV = Barrier$, which was also the only ancestor of t at this level. This DCV was on the augmenting path. The erasure of this DCV eventually caused the erasure of y. In case (3) RIGHTDFS backtracked above y because, at some time during its earch, $DCV = v_L$ was the only ancestor of y at level $Level(v_L)$. Later, either the path from s to v_L was extended into an augmenting path to an exposed vertex, or LEFTDFS backtracked above this v_L because of a new $DCV = Barrier$ at a smaller level. Thus y had at some level only one ancestor, either v_L or the new DCV, and this ancestor was on the augmenting path. Since this ancestor was erased, y itself became erased.

8.2. Time Complexity. Let us analyze the running time of the algorithm. Let m be the number of edges and n be the number of vertices of an input graph. Without loss of generality, assume that the graph has no isolated vertices; that is, every vertex is incident on at least one edge. Consequently, $n \leq 2m$. During each phase the algorithm increases the matching along a maximal set of disjoint minimum length augmenting paths. Thus there are $O(n^{1/2})$ phases (Hopcroft and Karp, 1973). We show that each phase runs in $O(m)$ time.

During one phase, subroutine SEARCH considers each vertex at most twice—once at an odd search level, once at an even search level. (Only inner vertices in blooms are considered twice.) Furthermore, SEARCH scans each edge at most twice, once in each direction. It follows that the total time for operations in SEARCH is $O(m+n)=O(m)$.

During one phase, the depth-first search processes of BLOSS-AUG use each edge at most once. Once a vertex v has a "left" or "right" mark, neither LEFTDFS nor RIGHTDFS considers the predecessor edges of v again—even on subsequent calls to BLOSS-AUG. Section 7.6 established that the total time for *base** computations can be kept to $O(m+n)$. The total time for operations in BLOSS-AUG is $O(m+n)=O(m)$.

During one phase, FINDPATH marks each edge "visited" at most once, each vertex "visited" at most once. The total time for FINDPATH is $O(m+n)=O(m)$. During one phase, ERASE decrements a $Count(z)$ variable once for each predecessor edge joining z to a predecessor of z. The total time for ERASE is at most proportional to the number of edges, $O(m)$. In summary, each phase runs in $O(m)$ time, and the algorithm runs in $O(n^{1/2}m)$ time.

Appendix. This appendix presents the complete matching algorithm. The presentation uses constructs from modern programming languages. Instead of **begin–end** pairs, however, the indentation of blocks of statements specifies the control structures. Each statement ends either at the end of a line or at a semicolon. Straight brackets "[" and "]" delimit comments. The **exit** statement exits the subroutine. Peterson (1985) wrote a program in Pascal to implement this algorithm.

General Matching Algorithm

```
repeat
[New phase: Initialization]
  for each vertex $v$ do
    $EvenLevel(v) := +\infty$; $OddLevel(v) := +\infty$
    $Bloom(v) :=$ undefined [name of bloom to which $v$ belongs]
    $Predecessors(v) :=$ empty; $Successors(v) :=$ empty; $Anomalies(v) :=$ empty
    $Count(v) := 0$ [number of unerased predecessors]
    Mark $v$ "unerased" [for disjoint augmenting paths]
    Mark $v$ "unvisited" [for depth-first search in FINDPATH]
    Delete any "left" or "right" mark [for BLOSS-AUG]
  for each edge $(u, v)$ do
    Mark $(u, v)$ "unused" [for depth-first search in BLOSS-AUG]
    Mark $(u, v)$ "unvisited" [for depth-first search in FINDPATH]
  for $i := 1$ to $|V|$ do
    $Candidates(i) :=$ empty [vertices to be searched at level $i$]
    $Bridges(i) :=$ empty [set of bridges at level $i$]
```

[Each of the sets *Predecessors*, *Successors*, *Anomalies*, *Candidates*, and *Bridges* can be implemented by a linear list such as a queue. The operations on these sets are to insert a new vertex and to select the next vertex in the set. Both operations can be implemented in constant time.]

```
  call SEARCH
until No augmentation occurred
```

SUBROUTINE SEARCH

Effect: Finds all augmenting paths of minimal length and increases the current matching along these paths.
Calls: BLOSS-AUG

```
(S0) [Initialization]
$i := 0$
for each exposed vertex $v$ do
  $EvenLevel(v) := 0$; Insert $v$ into $Candidates(0)$
(S1) [Breadth-first search]
while $Candidates(i)$ is not empty and No augmentation occurred at level $i-1$ do
  if $i$ is even then for each $v$ in $Candidates(i)$ do
    for each unerased neighbor $u$ of $v$ such that $(u, v)$ is free do
      if $EvenLevel(u) < +\infty$
      then  [$(u, v)$ is a bridge, but because $u$ could belong to a blossom, $Level(u)$
            could be greater than $Level(v)$]
            $j := (EvenLevel(u) + EvenLevel(v))/2$
            Insert $(u, v)$ into $Bridges(j)$
      else  if $OddLevel(u) = +\infty$
            then  $OddLevel(u) := i+1$
            if $OddLevel(u) = i+1$
            then  Add 1 to $Count(u)$
                  Insert $v$ into $Predecessors(u)$
                  Insert $u$ into $Successors(v)$
                  Insert $u$ into $Candidates(i+1)$
            if $OddLevel(u) < i$
            then  Insert $v$ into $Anomalies(u)$
  if $i$ is odd then for each $v$ in $Candidates(i)$ such that $v$ belongs to no bloom do
    Let $u$ be the mate of $v$ [$v$ must be matched]
    if $OddLevel(u) < +\infty$
    then  $j := (OddLevel(u) + OddLevel(v))/2$
          Insert $(u, v)$ into $Bridges(j)$
    if $EvenLevel(u) = +\infty$
```

```
        then   $Predecessors(u) := \{v\}$; $Successors(v) := \{u\}$; $Count(u) := 1$
               $EvenLevel(u) := i+1$; Insert $u$ into $Candidates(i+1)$
    for each edge $(s, t)$ in $Bridges(i)$ do
        if $s$ and $t$ are both unerased [Calls to BLOSS-AUG may induce erasure of
          vertices]
        then   call BLOSS-AUG with input $(s, t)$
    $i := i+1$
```

SUBROUTINE BLOSS-AUG

Input: A bridge (s, t)
Effect: Either discovers a bloom or augments the current matching.
Called by: SEARCH
Calls: LEFTDFS, RIGHTDFS, ERASE

```
(B0) [Initialization]
if $s$ and $t$ belong to the same bloom then exit
if $s$ belongs to a bloom then $v_L := base^*(Bloom(s))$ else $v_L := s$
if $t$ belongs to a bloom then $v_R := base^*(Bloom(t))$ else $v_R := t$
Mark $v_L$ "left" and $v_R$ "right"
$DCV :=$ undefined; $Barrier := v_R$
(B1) [Double depth-first search]
while $v_L$ and $v_R$ are not both exposed do
[If $Level(s) = Level(t)$, then except during backtracking, either $Level(v_L) =$
$Level(v_R)$ or $Level(v_L) = Level(v_R) - 1$]
    if $Level(v_L) \geq Level(v_R)$
    then   call LEFTDFS
    else   call RIGHTDFS
    if This call discovers a bloom
    then   goto step (B3)
(B2) [$v_L$ and $v_R$ are both exposed]
call FINDPATH to find a path $P_L$ from $High = s$ to $Low = v_L$ with $B =$ undefined
call FINDPATH to find a path $P_R$ from $High = t$ to $Low = v_R$ with $B =$ undefined
Increase the matching along the path starting with the reversal of $P_L$ through
$(s, t)$ ending with $P_R$
call ERASE to erase all vertices along the augmenting path
exit
(B3) [Creation of a new bloom]
Remove the "right" mark from $DCV$
Create a new bloom $B$ that comprises all vertices marked "left" or "right" during
the current call to BLOSS-AUG
for each $y$ in $B$ do $Bloom(y) := B$
The peaks of $B$ are $s$ and $t$
$base(B) := DCV$
for each $y$ in $B$ do
    $Bloom(y) := B$
[Set the other-level of each vertex in $B$]
```

```
if $y$ is outer
then   $OddLevel(y) := 2i+1-EvenLevel(y)$
if $y$ is inner
then   $EvenLevel(y) := 2i+1-OddLevel(y)$
       Insert $y$ into $Candidates\ (EvenLevel(y))$
       for each $z$ in $Anomalies(y)$ do
         $j := (EvenLevel(y)+EvenLevel(z))/2$ [At this point $j > i$]
         Insert $(y, z)$ into $Bridges(j)$
         Mark $(y, z)$ "used"
```

SUBROUTINE LEFTDFS

Inputs: Vertices s, v_L, v_R, *DCV*, *Barrier*
Effect: One step of left depth-first search process—advances v_L to a predecessor or backtracks or signals the discovery of a bloom.
Called by: BLOSS-AUG

```
(L0) [Search each predecessor edge]
while $v_L$ has "unused" predecessor edges do
  Choose an "unused" predecessor edge $(v_L, u)$ such that $u$ is "unerased"
  Mark $(v_L, u)$ "used"
  if $Bloom(u)$ is defined
  then   $u := base^*(Bloom(u))$
  if $u$ is not marked "left" or "right"
  then   Mark $u$ "left"; $Parent(u) := v_L$; $v_L := u$
         exit
```

[If u is marked "left" or "right," then LEFTDFS is backtracking. At this time $v_R = Barrier = DCV$, and $Level(v_L) > Level(v_R)$. Because the search is depth first, either u is on the alternating path from t to v_R, or all predecessor edges of u have already been used.]

```
(L1) [$v_L$ has no more "unused" predecessor edges]
if $v_L = s$ [Input to BLOSS-AUG]
  then   Signal discovery of a bloom
  else   $v_L := Parent(v_L)$ [Backtrack]
```

SUBROUTINE RIGHTDFS

Inputs: Vertices v_L, v_R, *DCV*, *Barrier*
Effect: One step of right depth-first search process—advances v_R to a predecessor or backtracks.
Called by: BLOSS-AUG

```
(R0) [Search each predecessor edge]
while $v_R$ has "unused" predecessor edges do
  Choose an "unused" predecessor edge $(v_R, u)$ such that $u$ is "unerased"
  Mark $(v_R, u)$ "used"
  if $Bloom(u)$ is defined
  then   $u := base^*(Bloom(u))$
  if $u$ is not marked "left" or "right"
  then  Mark $u$ "right"; $Parent(u) := v_R$; $v_R := u$
        exit
  else  if $u = v_L$ then $DCV := u$
```

[If u is marked "left" or "right," then because the search is depth first, either u is on the alternating path from s to v_L, or all predecessor edges of u have already been used.]

```
(R1) [$v_R$ has no more "unused" predecessor edges]
if $v_R = Barrier$
then  $v_R := DCV$; $Barrier := DCV$; Replace the "left" mark on $v_R$ by "right"
      $v_L := Parent(v_L)$ [Force LEFTDFS to backtrack from $v_L = DCV$]
else  $v_R := Parent(v_R)$ [Backtrack]
```

Subroutine ERASE

Input: Set Y of vertices to be erased [Y can be implemented by a queue]
Effect: Marks all vertices in Y "erased."
Once all predecessors of a vertex z have been erased, z is erased too.
Called by: BLOSS-AUG

```
repeat
  Remove a new vertex $y$ from $Y$
  Mark $y$ "erased"
  for each "unerased" $z$ in $Successors(y)$ do
    Subtract 1 from $Count(z)$
    if $Count(z)$ is now 0 then Insert $z$ into $Y$
until $Y$ is empty
```

Subroutine FINDPATH

Inputs: Vertices *High* and *Low* with $Level(High) \geq Level(Low)$, bloom B
Output: An alternating *Path* from *High* to *Low* through *Predecessors*
Called by: BLOSS-AUG, OPEN
Calls: OPEN, which finds paths through blooms other than B

```
(F0) [Initialization]
if $High = Low$
then $Path := High$; exit
$v := High$

(F1) [Depth-first search to find $Low$]
repeat
  while $v$ has no "unvisited" predecessor edges do $v := Parent(v)$
  if $Bloom(v) = B$ or $Bloom(v) =$ undefined
  then  Choose an "unvisited" predecessor edge $(u, v)$
        Mark $(u, v)$ "visited"
  else  $u := base(Bloom(v))$
  if ($u$ is "unvisited") and ($u$ is "unerased")
    and ($Level(u) > Level(Low)$)
    and ($u$ has same "left"/"right" mark as $High$)
  then  Mark $u$ "visited"; $Parent(u) := v$; $v := u$
until $u = Low$
```

(F2) [Path has been found, except for blooms other than B]

```
Let $Path = x_1, \ldots, x_m$ be the path defined by the $Parent$ pointers from $High$ to $Low$
for $j := 1$ to $m-1$ do
  if $Bloom(x_j)$ is defined and $Bloom(x_j) \neq B$
  then  [At this point $x_{j+1} = base(Bloom(x_j))$]
        Replace $x_j$ and $x_{j+1}$ by output of OPEN($x_j$)
```

SUBROUTINE OPEN

Inputs: Vertex x
Output: An alternating path *Path* from x through $Bloom(x)$ to $base(Bloom(x))$
Called by: FINDPATH
Calls: FINDPATH

```
$B := Bloom(x)$; $b := base(B)$
if $x$ is outer
then  Let $Path$ be the output of FINDPATH on inputs $x, b, B$
if $x$ is inner
then  Let $LeftPeak$ and $RightPeak$ be the peaks of $B$
      if $x$ is marked "left"
      then  call FINDPATH with inputs $LeftPeak, x, B$
            call FINDPATH with inputs $RightPeak, b, B$
            Let $Path$ be the concatenation of these paths
      if $x$ is marked "right"
      then  call FINDPATH with inputs $RightPeak, x, B$
            call FINDPATH with inputs $LeftPeak, b, B$
            Let $Path$ be the concatenation of these paths
```

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

Berge, C. (1957), Two theorems in graph theory, *Proc. Nat. Acad. Sci. U.S.A.* **43**, 842-844.

Bondy, J. A., and Murty, U. S. R. (1976), *Graph Theory with Applications*, Elsevier North-Holland, New York.

Edmonds, J. (1965), Paths, trees and flowers, *Canad. J. Math.* **17**, 449-467.

Even, S., and Kariv, O. (1975), An $O(n^{2.5})$ algorithm for maximum matching in general graphs, *Proc. 16th Ann. Symp. on Foundations of Computer Science*, IEEE, pp. 100-112.

Fujii, M., Kasami, T., and Ninomiya, K. (1969), Optimal sequencing of two equivalent processors, *SIAM J. Appl. Math.* **17**, 784-789.

Gabow, H. N. (1976), An efficient implementation of Edmonds' algorithm for maximum matching on graphs, *J. Assoc. Comput. Mach.* **23**, 221-234.

Gabow, H. N., and Tarjan, R. E. (1985), A linear-time algorithm for a special case of disjoint set union, *J. Comput. System Sci.* **30**, 209-221.

Hajek, B. (1984), Link schedules, flows, and the multichromatic index of graphs, *Proc. 1984 Conf. on Information Sciences and Systems*, Princeton University, Princeton, NJ, pp. 498–502.

Hopcroft, J. E., and Karp, R. M. (1973), An $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM J. Comput.* **2**, 225–231.

Kameda, T., and Munro, I. (1974), An $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs, *Computing* **12**, 91–98.

Micali, S., and Vazirani, V. (1980), An $O(\sqrt{|v|}\,|E|)$ algorithm for finding maximum matching in general graphs, *Proc. 21st Ann. Symp. on Foundations of Computer Science*, IEEE, pp. 17–27.

Papadimitriou, C. H., and Steiglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

Peterson, P. A. (1985), The general maximum matching algorithm of Micali and Vazirani, Tech. Rep. ACT-62, Coordinated Sci. Lab., Univ. Illinois at Urbana-Champaign, Aug. 1985.

Reingold, E. M., Nievergelt, J., and Deo, N. (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ.

Tarjan, R. E. (1975), Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22**, 215–225.

Tarjan, R. E. (1983), *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA.