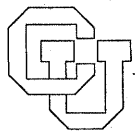
Using Blossoms to Find Maximum Matchings on Graphs

Harold Gabow

CU-CS-056-74 September 1974



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

USING BLOSSOMS TO FIND MAXIMUM MATCHINGS ON GRAPHS

by Harold Gabow

University of Colorado

Abstract

A matching on a graph is a set of edges, no two of which share a vertex. The problem discussed is to find a matching with the greatest number of edges possible. An algorithm is presented that implements the concept of blossoms, as developed by Edmonds, completely and efficiently. The computation time is proportional to V^3. The algorithm can be used in an algorithm that computes maximum weighted matchings efficiently.

1. Introduction

Finding a maximum matching on a graph is an interesting representative of a class of integer programming problems that can be solved efficiently [5]. It has applications in operations research. For example, the following is a maximum matching problem: A squadron commander must divide his men into teams of two. Certain teams are not acceptable because the two men do not get along. Choose the greatest number of acceptable teams.

A number of algorithms have been given for computing maximum matchings [1,2,3,6,10,11]. The most successful follow the basic approach given by Edmonds [3]. In that approach, the notion of blossoms is central. This paper describes a maximum matching algorithm that takes advantage of the complete theoretical significance of blossoms, in an efficient manner. Because of this, the algorithm generalizes easily to weighted matching and other problems [7,8].

The algorithm has a worst-case computation time that is $O(V^3)$, where V is the number of vertices. This bound is the best currently known. The bound has been achieved by several algorithms [1,6,10]. However, these algorithms implement restricted aspects of blossoms, and so do not generalize easily.

Sections 2 and 3 of this paper present background material: definitions from graph theory, and a summary of Edmonds' method for matching with blossoms. The next three sections describe the algorithm: Section 4 presents a data structure for matching with blossoms; its use is illustrated in Section 5, which describes blossom expansion; Section 6 presents a data structure for choosing edges. The algorithm is not

described in complete detail because of its length. Instead, examples and sample Algol procedures are given. The entire algorithm can be reconstructed from these. Also, a complete description of the algorithm can be found in [7].

Finally in Section 7, the efficiency and applications of the algorithm are discussed.

2. Preliminaries

This section summarizes some well-known definitions and results. A graph consists of a finite set of vertices and a finite set of edges. An edge is an unordered set of two distinct vertices. The edge containing vertices v and w is denoted (v,w) or (w,v). Vertices v and w are adjacent, and edge (v,w) is incident to v and to w.

If the order of vertices in edge (v,w) is significant, it is a directed edge. The directed edge (v,w) goes from v to w; the head of (v,w) is w, and the tail is v.

Throughout this paper, G denotes a given graph; V denotes the number of vertices in G; E denotes the number of edges in G.

A path in G is an ordered list of vertices $(v_1,v_2,\ldots,v_n)$, such that no vertex occurs more than once in the list, and (v_i,v_{i+1}) is an edge, for $1 \leq i < n$. The path joins v_1 to v_n.

A matching M on G is a set of edges, no two of which share a vertex. A vertex is matched if it is in some edge of the matching; otherwise it is unmatched. M is a maximum matching if no matching on G contains more edges than M. Figure 1 shows two matchings on a graph G1 (Matched edges are drawn wavy). In Fig. 1(a) there are two unmatched vertices, 17 and 18; in Fig. 1(b) the matching is maximum, as all vertices are matched.

An alternating path is a path $(v_1,\ldots,v_n)$ such that one of every two consecutive edges (v_{i-1},v_i) and (v_i,v_{i+1}) is matched, for $1 < i < n$. An augmenting path is an alternating path that joins two unmatched vertices.

If $(v_1,\ldots,v_{2n})$ is an augmenting path, a new matching M' is obtained from M by replacing the matched edges (v_{2i},v_{2i+1}), $1 \leq i < n$, with the unmatched edges (v_{2i-1},v_{2i}), $1 \leq i \leq n$. We say M is augmented to M', since M' contains one more edge than M. In Fig. 1(a), (17,16,15,12,7,8,

9,13,14,18) is an augmenting path. Augmenting gives the matching in Fig. 1(b).

A fundamental fact is this: A matching M has an augmenting path if and only if M is not maximum. (For a proof, see [2] or [3]). As a result, a maximum matching can be found by repeatedly searching for an augmenting path and augmenting the matching.

A tree T is a graph with a distinguished vertex r, such that there is a unique path P(v,r) joining any vertex v to r. Vertex r is the root of T. If path P(v,r) contains vertex w, then vertex v is a descendent of w.

An equivalent, recursive way to define a tree is as follows: A tree T is a finite set of vertices, with a distinguished root vertex r, such that the other vertices of T (if any) are partitioned into disjoint trees $T_1, T_2, \ldots, T_m$, where $m \geq 0$. There are edges (r, r_i), where r_i is the root of T_i, for $1 \leq i \leq m$. The trees T_i are the subtrees of r.

In an ordered tree, the order of subtrees $T_1, \ldots, T_m$ is significant. In a labeled tree each vertex contains some information, called its label.

3. Matching with Blossoms

This section describes the basic algorithm for computing a maximum matching using blossoms. The algorithm was discovered by Edmonds [3]. The algorithm is illustrated by showing how it constructs the maximum matching in Figure 1 for graph G1.

The algorithm begins with all vertices in the given graph G unmatched. It searches for an augmenting path. When such a path is found, the matching is augmented. Then a new search begins, for an augmenting path in the new matching. The search-augment process is repeated, until finally no augmenting path is found. At this point the algorithm halts with a maximum matching.

The search is complicated by the fact that the graph is repeatedly transformed. Under certain conditions, a collection of vertices is replaced by a single vertex, called a blossom. This forms a new graph. A blossom can be absorbed, along with other vertices and blossoms, into a new blossom, adding further complexity. Also, under certain conditions, a blossom is replaced by its constituent vertices and blossoms. This forms a new graph, perhaps different from any previously used.

Throughout this paper, we call the graph currently being searched the working graph. We refer to vertices of the working graph as either vertices or blossoms.

Figure 2 shows the augmenting paths found in constructing the maximum matching on graph G1. A blossom b1 forms in search S5; it belongs to the augmenting path found in this search. Blossoms b2, b3, and b4 form in subsequent searches. In search S9, blossoms b4 and b3 are expanded into their constituent vertices.

A search works by constructing a number of trees made up of alternating paths. An alternating tree is defined as a tree such that (i) the root is an unmatched vertex u, and (ii) any path in the tree, P(v,u), from a vertex v to the root u, is an alternating path. A vertex v is called outer if either v is the root u or path P(v,u) starts with a matched edge. Otherwise, if P(v,u) starts with an unmatched edge, vertex v is called inner.

A search constructs an alternating tree for each unmatched vertex u. This is done by scanning edges, and adding edges and vertices to the tree. Eventually, the search scans an edge joining two outer vertices v_1, v_2 that are in different trees. The alternating paths $P(v_1,u_1)$ and $P(v_2,u_2)$, together with edge (v_1,v_2), form an augmenting path between vertices u_1 and u_2. At this point, the search terminates by augmenting the matching.

Figure 3 shows the alternating trees grown in search S9. The tree in Figure 3(f) contains five outer and four inner vertices. In Figure 3(g), an edge joining the two trees, (9, 13), is scanned, and an augmenting path is found.

A search constructs alternating trees in four different steps. Now we describe these steps, called initialize, grow, blossom, and expand.

Initialize: A search begins by making every unmatched vertex the root of an alternating tree. Figure 3(a) illustrates this step for search S9.

Grow: Suppose the search scans an edge (x,y), where vertex x is outer in some alternating tree, and vertex y is not in any alternating tree.

Then a grow step is executed. The alternating tree containing x is extended by two edges, (x,y) and (y,v). Here (y,v) is the matched edge containing vertex y. (Edge (y,v) exists, since otherwise y is the root of an alternating tree.) Vertex y is made an inner vertex, and v is made outer.

Figure 3(e)-(f) illustrates a grow step. In Figure 3(e), vertex 12 is outer, and vertex 8 is not in any tree. When edge (12,8) is scanned, a grow step is executed, as shown in Figure 3(f).

Blossom: Suppose the search scans an edge (x,y), where x and y are outer vertices in the same alternating tree. Then a blossom step is executed. Figure 4 illustrates this step. The paths P(x,u) and P(y,u) join at an outer vertex j. In the blossom step, all vertices up to and including j in paths P(x,u) and P(y,u) are replaced by a single vertex b. This new vertex is called a blossom. Blossom b is adjacent to any vertex that was previously adjacent to a constituent vertex of b. In the new working graph, b is an outer vertex.

The four blossoms steps executed in searches S1-S9 are shown in Figure 5. For example in Figure 5(b), edge (7,b1) is scanned. This edge comes from edge (7,3) in the original graph. (This is indicated by the "3" at the end of edge (7,b1)). Blossom b2 is formed, and replaces vertices 7, 6, b1, 4, and 5.

The rationale for blossom steps is that any matching on the new working graph gives a matching on the original working graph. Suppose in the new working graph, the matched edge incident to b corresponds to (v,v'). Here v is a vertex in b, and v' is not in b. Referring to Fig. 4, there is an alternating path P(v,j) from v to j, that starts with a

matched edge. (If v is an outer vertex, path P(v,j) is the beginning of P(v,u). If v is inner, path P(v,j) consists of a path from v to x (or y), plus P(x,j) (or P(y,j)). For example, $P(x_1,j)$ consists of edges (x_1,x), (x,y), and path P(y,j).) If we rematch the edges in P(v,j) (switching the matched and unmatched edges), then v is no longer matched with a vertex in b. This allows edge (v,v') to be matched. It gives the desired matching on the original working graph.

When a search terminates by augmenting, the next search begins in the current working graph. The blossoms are not "undone"; they remain as vertices. Blossoms do get "undone" in the fourth type of step.

Expand: Suppose a blossom b is made inner in a grow step of some search. (Note this search is not the one in which blossom b is formed. Blossom b is outer when it forms.) Blossom b may be expanded during this search.

The expand step replaces b by its constituent vertices. This necessitates two changes. First, the matching on the constituent vertices is changed. This accounts for the augments done since the formation of blossom b. Second, some constituent vertices are placed in the tree to replace b. This prevents losing the portion of the tree "hanging" from b.

Figure 3(b)-(c) shows the expansion of blossom b4. The matching is not changed; the three constituent vertices of b4 are placed in the tree. Figure 3(d)-(e) shows the expansion of blossom b3. The matching is changed from the original one (see Figure 5(c)) to reflect the augment in search S7 (see Figure 2). Also, vertex b2 is placed in the tree.

Expand steps are necessary to make sure inner blossoms do not "hide" augmenting paths from the search. For example, in the working graph of Figure 3(d), the augmenting path in Figure 3(g) corresponds to (17, 10, 11, b3, 12, b3, 13, 14, 18). This path is not detected as an augmenting path. When blossom b3 is expanded, the augmenting path is no longer hidden.

An inner blossom can be expanded at any point in the search. There are two situations where expansion is unnecessary. In the first case, an augmenting path is found before expanding; clearly expansion is unnecessary. In the second case, the inner blossom gets absorbed in a new outer blossom, in a blossom step; expansion is unnecessary since no augmenting paths are "hidden." If neither case occurs, the blossom gets expanded in the search.

This concludes our description of the basic algorithm. A complete discussion of the underlying theory is given in [3]. The flowchart in Fig. 6 summarizes the search procedure. The rest of the paper describes an efficient implementation of the flowchart.

4. A Data Structure for Blossoms

This section presents a data structure that allows blossoms to be formed, manipulated, and expanded efficiently. Data specifying blossoms is stored in five parallel arrays, called BLOSSOM, MATE, LABEL, NEXT, and LAST. We begin with some general remarks. Then we describe each array and illustrate its use.

The algorithm begins by numbering the vertices and edges of the given graph G. The vertices are numbered from 1 to V. Each of the five arrays in the data structure has V entries, one for each vertex. A vertex number is used as an array index, designating the entry for that vertex. Usually we identify a vertex and its number.

The edges of G are numbered from 1 to 2E. Each edge (x,y) is given two consecutive numbers. The first number is associated with the directed edge (x,y), the second number with the directed edge (y,x). (Note the edges of G themselves have no associated direction.) Edge numbers are stored in the MATE and LABEL arrays. The direction associated with an edge number allows additional information to be saved. Usually we identify an edge and its number.

We give a method for extending the numbering of the vertices to include blossoms. The blossom index, defined below, identifies a blossom, and is used as an array index.

Definition 1: The blossom index for a blossom b, denoted i(b), is an integer between 1 and V, defined recursively as follows:

(i) Suppose b is a vertex in G, the original graph. Then i(b) is the number of vertex b.

(ii) Suppose b is a blossom formed from vertices x, x_1, ... ,

x_{2r+1}, y, y_1, ... , y_{2s+1}, j, as in Fig. 4. Then i(b) = i(j).

(Note j is the vertex where paths P(x,u) and P(y,u) join together.) For example, in Fig. 5, i(b4) = i(b3) = 9.

These indices can be used to identify blossoms, since in a given working graph, distinct blossoms have distinct indices. This gives the first use for blossom indices. It is illustrated by the BLOSSOM array. This array specifies which blossom contains a given vertex in the current working graph.

Definition 2: Let v be a vertex in G. The entry BLOSSOM[v] is the index of the blossom containing v in the current working graph. So for the blossom b with v ε b, BLOSSOM[v] = i(b).

For example, in the working graph of Fig. 5(d), vertices 1-11 are contained in blossom b4, so their BLOSSOM value is 9.

The second use for blossom indices is as array indices. This is illustrated by the MATE and LABEL arrays. These arrays are maintained as follows. When the algorithm begins, every vertex is a blossom index. MATE and LABEL information for a vertex v is stored in entry number v. When a blossom b forms, MATE and LABEL information for b is stored in entry number i(b). The information about the vertex numbered i(b) is no longer needed.

Now we describe the MATE array in detail. It specifies the matching on the current and previous working graphs.

Definition 3: Let v be a vertex in G. Entry MATE[v] is the number of a directed edge (x,y), defined as follows. Let W be the last working graph in which v is the index of a blossom b. Then vertex x ε b, and (x,y) corresponds to the matched edge incident to b in W. (If b is unmatched, then MATE[v]=0).

As an example, consider vertex 1 in Fig. 5. It is a blossom index in Fig. 5(a)-(b). In Fig. 5(a), MATE[1] = (1,9); in Fig. 5(b), MATE[1] = (3,4). This entry does not change in Fig. 5(c)-(d), since vertex 1 is no longer a blossom index. Table 1 gives the MATE entries for the vertices in blossom b4.

Now we describe the LABEL array. It specifies the structure of the alternating trees in the current working graph, and of blossoms in previous working graphs.

Definition 4: Let v be a vertex in G. Entry LABEL[v] is the number of a directed edge (x,y), defined as follows. Let W be the last working graph in which v is the index of a blossom b.

(i) Suppose b is an outer vertex in W. Then (x,y) corresponds to the first unmatched edge in path P(b,u). More precisely, if $P(b,u) = (b, b_1, b_2, \ldots, u)$, then edge (x,y) corresponds to (b_1,b_2). Also, vertex $x \in b_1$. (If b is an unmatched outer vertex, P(b,b) has no edges. In this case LABEL[v] = 0).

(ii) Suppose b is an inner vertex in W. If b is absorbed in a new blossom in the next working graph, then (x,y) corresponds to the first unmatched edge in path P'(b,u). Otherwise, b is an inner vertex in the current working graph; LABEL[v] is an arbitrary negative number.

(iii) Suppose b is neither outer nor inner in W. Then b is a vertex in the current working graph that is not in a tree. Entry LABEL[v] is an arbitrary negative number.

For example, consider Fig. 5(b). Blossom b1 is outer, so LABEL[1] = (4,5). Vertex 4 is inner. Before the blossom step, LABEL[4] < 0; after it, LABEL[4] = (3,7). Table 1 gives the LABEL entries for vertices in blossom b4.

The following pseudo-Algol code illustrates the use of the MATE and LABEL arrays. It shows how the matching is updated when an augmenting path is formed.

```
procedure augment (integer value ea);
comment Parameter ea is an edge that completes an augmenting path.
        Thus ea joins two outer vertices v1, v2, and an augmenting
        path is formed by paths P(v1,u1), P(v2,u2), and edge ea;
for side := 1,2 do
comment Rematch two "sides" of the augmenting path;
    begin
    b1 := BLOSSOM[tail of ea]; comment The tail of a directed edge
          (x,y) is x, and the head is y;
    MATE[b1] := ea; comment Match one end of ea;
    e := LABEL[b1]; comment e steps through the edges to be matched;
    while e ≠ 0 do
        begin
        comment Get the two ends of edge e;
        b1 := BLOSSOM[head of e];
        b2 := BLOSSOM[tail of e];
        comment Match edge e;
        MATE[b2] := e;
        MATE[b1] := edge opposite to e;
        comment Advance e along the path;
        e := LABEL[b1];
        end;
    ea = edge opposite to ea;
    end;
```

Now we describe the NEXT and LAST arrays. These arrays keep track of which vertices belong to which blossoms in the current and previous working graphs. They are best described in terms of a tree corresponding to a blossom, defined as follows.

Definition 5: The blossom tree of a blossom b, T(b), is a labeled, ordered tree, defined recursively as follows:

(i) Suppose b is a vertex in G. Then T(b) consists of a root only. The root is labeled with the number of b.

(ii) Suppose b is a blossom formed from vertices x, x_1,..., x_{2r+1}, y, y_1,..., y_{2s+1}, j, as in Fig. 4. For blossom tree T(j), let $\overline{T}(j)$ denote the subtrees of the root, in order. Then the blossom tree T(b) has a root labeled i(b). The subtrees of the root, in order, are

$$T(x),\ T(x_1),\ldots,\ T(x_{2r+1}),\ T(y),\ T(y_1),\ldots,\ T(y_{2s+1}),\ \overline{T}(j).$$

The vertices of T(b) are labeled by the vertices (of G) in b. (Since i(b) = i(j), no vertex is lost by replacing T(j) with $\overline{T}(j)$.) The trees for blossoms b1-b4 are illustrated in Fig. 7.

The NEXT and LAST arrays derive from the preorder lists of blossom trees. The preorder list of an ordered tree is a list of its vertices, defined recursively as follows: the root of the tree is listed first; next come the preorder lists for the subtrees of the root, in subtree order [9]. Thus the preorder list L(b) for the blossom tree T(b) is

$$i(b),\ L(x),\ L(x_1),\ldots,\ L(x_{2r+1}),\ L(y),\ L(y_1),\ldots,\ L(y_{2s+1}),\ \overline{L}(j),$$

where the notation is analogous to that of Definition 5.

The NEXT array stores the preorder lists of all blossom trees.

Definition 6: Let v be a vertex in G. Entry NEXT[v] contains the vertex after v in the preorder list of T(b), where b is the blossom containing v in the current working graph. (If v is last in the list, NEXT[v] = 0.)

Thus the preorder list for T(b) is

$$i(b), NEXT[i(b)], NEXT^2[i(b)], \ldots, NEXT^m[i(b)],$$

where m is such that $NEXT^{m+1}[i(b)] = 0$. Table 1 gives the NEXT entries for blossom b4.

The LAST array indicates the ends of preorder lists.

Definition 7: Let v be a vertex in G. Entry LAST[v] contains the last vertex (in the preorder list of T(b)) that is a descendent (in T(b)) of v. Here b is the blossom containing v in the current working graph.

Thus the subtree of v in T(b), in preorder, is

$$v, NEXT[v], \ldots, NEXT^m[v],$$

where $NEXT^m[v] = LAST[v]$. Table 1 gives the LAST entries for blossom b4.

The following code illustrates the use of NEXT and LAST, by showing how these arrays are updated when a new blossom is formed.

```
procedure set_N_L (integer value x,y,j);
comment Parameters x and y are vertices in the original graph, and j is
        a blossom index. As in Fig. 4, edge (x,y) completes the new
        blossom, and j is the blossom where the two paths join together;
begin
rear := j; comment rear indicates the end of the part of the preorder
                list constructed so far. Initially, only j is in the
                list;
```

```
save := NEXT[j]; comment Save the preorder list of j, for later inser-
                         tion in the list;
for side := BLOSSOM[x], BLOSSOM[y]do
comment  Process the vertices in both paths;
     begin
     b1 := side; comment b1 steps through the indices of the outer
                         vertices of the path,
     while b1≠j do
          begin
          b2 := BLOSSOM[head of MATE[b1]]; comment b2 steps through the
                                                   indices of the inner
                                                   vertices of the path;
          NEXT[rear] := b1; comment Add b1's list;
          NEXT[LAST[b1]] := b2; comment Add b2's list;
          rear := LAST[b2];
          b1 := BLOSSOM[head of LABEL[b1]];
          end;
     end;
comment  Now add j's list;
if save = 0 then LAST[j] := rear  else[NEXT]rear  := save;
end;
```

5. Expanding Blossoms

This section discusses expand steps in greater detail. A method for expanding blossoms is described. The use of the blossom arrays in expanding is illustrated.

We consider an example that is representative of the general case: A blossom b forms, as shown in Fig. 4, in the search S_1. This blossom is rematched in one or more augments. Then in the search S_2, blossom b is expanded. Before expansion, b is an inner vertex. The two edges incident to b in the alternating tree are the unmatched edge (y_1,y_1'), and the matched edge (x_1,x_1'). Vertices x_1 and y_1 are shown in Fig. 4; vertices x_1' and y_1' are not in blossom b.

As noted in Section 3, the expand step changes the MATE and LABEL arrays. Now we discuss these changes.

Before expanding, the MATE array stores the original matching on the constituent vertices of b (this matching is shown in Fig. 4, ignoring vertex j). The expand step must change the matching on b, so vertex x_1 is no longer matched with a vertex of b. (This allows edge (x_1,x_1') to be matched.) This can be done by rematching the path $P(x_1,j)$. The MATE array must be changed so these edges are matched:

$$(x,y),\ (y_1,y_2),\ (y_3,y_4),\ \ldots\ (y_{2s+1},j).$$

The LABEL array must be changed so blossom b is replaced in the alternating tree. To do this, vertices y_1 and x_1 in the tree must be joined by an alternating path. This can be done since in the new matching on blossom b, every vertex is joined to x_1 by an alternating path. In the example, the desired path is

$$(y_1,y_2,\ \ldots,\ y_{2s+1},j,\ x_{2r+1},\ \ldots,\ x_2,x_1).$$

Adding this path to the tree makes alternate vertices outer, starting

with y_2 and ending with x_2. New LABEL entries must be made for these vertices.

Now we give a method for expanding blossoms that accomplishes these changes. It consists of the three procedures described below.

The first procedure computes the new vertices in the working graph,

$$x, x_1, \ldots, x_{2r+1}, y, y_1, \ldots, y_{2s+1}, j.$$

These vertices may themselves be blossoms, so their constituent vertices are computed. The procedure also gives a negative sign to all LABEL entries for new vertices. This effectively removes these vertices from the tree (see Definition 4). However, the previous LABEL information is preserved (as negative edge numbers), for the two remaining procedures.

The second procedure updates MATE, by rematching path $P(x_1, j)$. It resembles the augment procedure in Section 4, although the negative edge numbers in LABEL are used. There is another addition: Entries in LABEL are changed to be compatible with the new matching. When the procedure is finished, the updated LABEL array defines alternating paths (in the new matching) from the new vertices to x_1. (However, the LABEL entries are still negative.)

The third procedure updates the LABEL array, adding the alternating path between y_1 and x_1 to the tree. This path is computed from the LABEL array, as in the augment procedure, using negative edge numbers. Alternate vertices along this path are made outer by assigning new (positive) LABEL values.

Complete details of the expand procedures can be found in [7]. Below, we give code for the first procedure. It illustrates how several blossom arrays are used together.

```
procedure  start-expand (integer value b);
comment  Parameter b is the index of the blossom to expand;
begin
comment  The procedure consists of two main loops.  In each loop,
         variables b1 and b2 step through the indices of the constituent
         vertices of blossom b.  This two-pass organization is not re-
         quired - the two loops can be combined into one.  This is not
         done here for the sake of clarity;
comment  Pass 1:  Step through the constituent vertices of b, to find the
         last one (in the preorder list);
b1 := NEXT[b];
b2 := NEXT[LAST[b1]];
comment  b2 is always one blossom ahead of b1;
e := eb := LABEL[b2];
comment  Referring to Fig. 4, eb is the edge (x,y) that formed blossom b.
         Variable e steps through the unmatched edges in blossom b,
for side := 1, 2 do
comment  Step through both "sides" of b;
     begin
     while LABEL[b2] = e do
     comment b1 and b2 are in the current side.  Advance;
          begin
          e := edge opposite to LABEL[b1];
          b1 := NEXT[LAST[b2]];
          b2 := NEXT[LAST[b1]];
          end;
     comment b1 is not in the current side.  Try the other one;
```

```
        e := edge opposite to eb;
        end;
    first := b1;
    comment  first is now the vertex following b in the preorder list for
             blossom j (see Fig. 4).  If no such vertex exists, first = 0;
    b1 := NEXT[b]; comment Return b1 to the first constituent vertex of b;
    comment  Now update arrays for blossom j;
    NEXT[b] := first;
    if first = 0 then LAST[b] := b;
    LABEL[b] := -LABEL[b];
    comment  Pass 2:  Update arrays for the other constituent blossoms b1 of b;
    while b1 ≠ first do
        begin
        b2 := NEXT[LAST[b1]];
        NEXT[LAST[b1]] := 0;
        LABEL[b1] := -LABEL[b1];
        comment Update BLOSSOM for all vertices  in blossom b1;
        i := b1;
        while i ≠ 0 do
            begin
            BLOSSOM[i] := b1;
            i := NEXT[i];
            end;
        b1 := b2;  comment Advance;
        end;
    end;
```

We conclude this section by mentioning another way the expand procedures are used. After the final search of the algorithm, a maximum matching has been constructed. However, the working graph may contain unexpanded blossoms. These blossoms must be expanded so the MATE array is correct. The last step of the algorithm does this, using the first two expand procedures.

For example, after the final search S9 in Fig. 3, the working graph contains a blossom b2. This blossom is expanded into its constituent vertices, 7,6,b1,4,5; then blossom b1 is expanded. The final matching obtained is shown in Fig. 1(b).

6. A Data Structure for Scanned Edges

This section describes a data structure for choosing edges in the search. (This step of the algorithm is shown in the flowchart of Fig. 6.) The EDGE data structure allows edges to be recorded, and later, chosen, efficiently.

We say a blossom b gets labeled when it is made outer in an initialize, grow, blossom, or expand step (The term derives from making a new LABEL entry for b). A label step is one of these four steps.

Inside the flowchart box for a label step, the edges incident to a newly labeled blossom are examined. If an edge can be used in a subsequent label step, it is recorded in the EDGE data structure. Later it is chosen (in the choose box) for processing.

Below we classify the edges that must be recorded. The notation of the flowchart is used: (x,y) is an edge; vertex x is in a newly labeled blossom b; vertex y is in a blossom c. The tree types of edges are characterized as follows.

Type 1: Blossom c is outer. Edge (x,y) can be used in a subsequent blossom or augment step. Hence it must be recorded. In Fig. 3(d), (16,17) is a Type 1 edge, as is (9,13) in Fig. 3(f).

Type 2: Blossom c is not in a tree. Edge (x,y) can be used in a subsequent grow step, so it must be recorded. In Fig. 3(b), (12,15) is a Type 2 edge.

Type 3: Blossom c is inner, and contains more than one vertex of G.

Edge (x,y) cannot be used in a label step, as long as blossom c is not expanded. Suppose blossom c gets expanded. Then vertex y is contained in a new blossom c', and edge (x,y) may now be Type 2. So this edge must be recorded. In Fig. 3(d), (12,8) and (13,8) are Type 3 edges. Figure 3(e) shows how (12,8) becomes a Type 2 edge.

If vertex x is in a newly labeled blossom, then all edges (x,y) that can be used in subsequent label steps (in the current search) are of Types 1-3. This motivates the following definition of EDGE. The EDGE array has an entry for each vertex of G.

Definition 8: Let y be a vertex in G. Entry EDGE[y] is either a directed edge (x,y), or a list of directed edges (x,y), defined as follows. In the current working graph, let y be in blossom c.

(i) Suppose either c is not in a tree, or c is inner and contains more than one vertex of G (as in Types 2-3). Then EDGE[y] is a directed edge (x,y), where vertex x is in an outer blossom. (If no such edge exists, EDGE[y] = 0).

(ii) Suppose c is outer (as in Type 1). Then EDGE[y] is a list of directed edges (x,y), where vertex x is in an outer blossom.

(iii) Suppose c is inner and consists of one vertex (y) of G. Then EDGE[y] = 0.

For example, in Fig. 5(b) before blossom b2 forms, EDGE[7] is the list of edges (10,7), (3,7).

This definition is a slight simplification of the actual data structure. A list of edge numbers (x,y) cannot be stored in one word of a random access machine. Actually, EDGE[y] is the head of a list of

edges, and each word of the list contains one edge. For convenience, we overlook this point.

When vertex y is in Type 2 or 3, entry EDGE[y] is a single edge. Sometimes several edges (x_1,y), (x_2,y),..., qualify as EDGE[y]. (For example, in Fig. 3(d), (13,8) and (12,8) both qualify as EDGE[8]). All candidate edges are equivalent: After any sequence of grow, blossom, and expand steps, either all or none of the edges (x_i,y) are Type 2. The choice of EDGE[y] is arbitrary. The important fact is that some edge (x_i,y) exists.

The following code illustrates how the EDGE data structure is used to implement the main loop of the algorithm, shown in the flowchart.

```
procedure search;
comment Search calls these procedures: initialize, grow, blossom and
        expand. Comments to a procedure statement describe the operation
        of that procedure;
begin
found := false; comment Boolean variable found is set true if an aug-
                        menting path is found;
initialize; comment Make each unmatched vertex the (outer) root of an
                    alternating tree. No other vertex is in a tree.
                    Scan the new outer vertices and make EDGE entries;
while EDGE contains a non-zero entry do
    begin
    y := an entry index with EDGE[y] ≠ 0; comment If there are several
                                                  candidates for y, choose
                                                  arbitrarily;
```

```
c := BLOSSOM[y];
if LABEL[c]< 0 then
     begin comment c is not outer;
     d := BLOSSOM[head of MATE[c]]; comment d is the blossom matched
                                                with c;
     if LABEL[d] < 0 then
          begin comment c and d are not in a tree;
          e := EDGE[y]; comment e is a Type 2 edge;
          grow(e); comment Make c inner and d outer.  If blossom c
                           contains only vertex y (LAST[c] := c),
                           set EDGE[y] := 0.  Scan edges from vertices
                           of d, and make EDGE entries;
          end
     else begin comment c is inner and d is outer.  EDGE[y] is Type 3;
          expand (c); comment Update the five blossom arrays.  For
                              new inner blossoms containing only one
                              vertex y', ser EDGE[y'] := 0.  For new
                              outer blossoms, scan edges and make
                              EDGE entries;
          end;
     end
else begin comment c is outer;
     for each edge e in the list EDGE[y] do
          begin comment e is Type 1;
          blossom (e); comment Compute the join blossom j (see Fig. 4).
                               If j = c (e is contained in blossom c),
                               do nothing.  If j does not exist
```

```
                                (e completes an augmenting path)
                                augment the matchine and reset found.
                                Otherwise (e forms a new blossom b),
                                update the five blossom arrays, scan
                                edges from the vertices of b that
                                were previously inner, and make EDGE
                                entries;
                    if found then go to exit;
                    end;
                EDGE[y] := 0; comment Delete the list of edges for y;
                end;
            end;
        comment If the while loop is exhausted,no  augmenting path exists;
        exit : end;
```

Note how the steps illustrated in Fig. 3(b)-(c) are executed by the search procedure. When vertex 17 is labeled and scanned, (Fig. 3(a)), entry EDGE[10] is set to (17,10). The first time through the while loop, entry EDGE[10] is chosen, and the grow step of Fig. 3(b) is done. The second time through the loop, EDGE[10] is chosen again, and blossom b4 is expanded (Fig. 3(c)). In the expand step, EDGE[10] is set to 0.

In general, an edge can be chosen many times in the while loop, causing a sequence of grow and expand steps. The sequence of steps done for an edge has the following form: zero or more expand steps, followed by a grow step, followed by zero or more expand steps.

7. Efficiency and Applications

This section discusses the efficiency of the algorithm, from theoretical and practical points of view. It also indicates applications and extensions of the method.

The execution time of the algorithm is bounded by $O(V^3)$. This bound results from doing at most V/2 searches, each requiring time $O(V^2)$. We briefly discuss how the most time-consuming portions of the search, the blossom and expand steps, achieve the $O(V^2)$ time bound.

In the search procedure of Section 6, the blossom procedure is called (at most) once for each edge. At most V/2 of these calls form a new blossom (since a new blossom decreases the number of vertices in the working graph by at least two). At most one call does an augment. The remaining calls take no action (as the Type 1 edge chosen is not in the working graph). Thus the blossom procedure requires at most time

$$(V/2) * O(V) + O(V) + O(V^2) = O(V^2).$$

The expand procedure is called at most V/2 times in a search. (Only blossoms formed in previous searches can be expanded, and there are at most V/2 of these.) Thus the expand procedure requires at most time

$$(V/2) * O(V) = O(V^2).$$

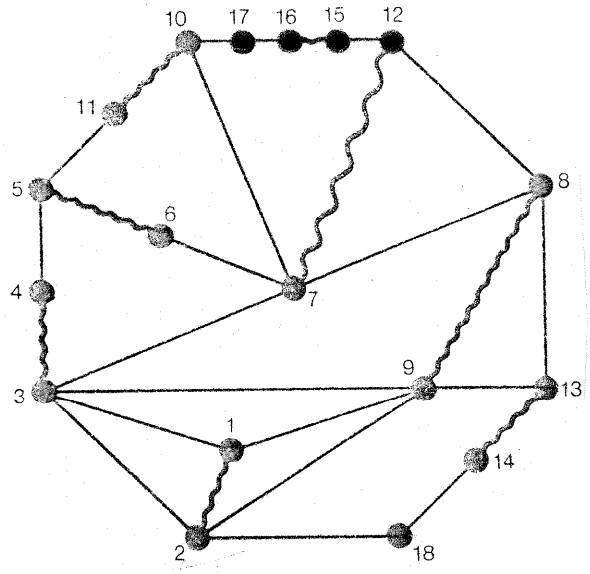
The other portions of the search procedure can be done in time $O(V)$. Thus a search requires at most time $O(V^2)$, and the algorithm executes in time $O(V^3)$.

Experiments were made to compare the blossom algorithm with a simpler method for maximum matching in time $O(V^3)$ [7]. The two algorithms were implemented in Algol W on the IBM 360/165. For "random" graphs with up to 1000 edges, the simpler algorithm was one to two orders of magnitude faster. This factor could be reduced by more careful programming of some

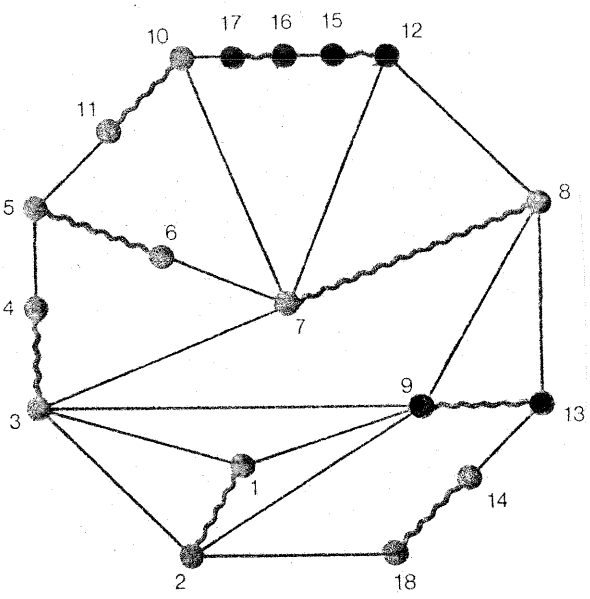
$O(V^3)$ portions of the algorithm. However, the blossom algorithm does not appear to be best for practical matching problems.

The main application of the blossom algorithm is in computing maximum matchings on weighted graphs. In a weighted graph, each edge has a numerical weight. The problem is to find a matching with maximum weight. An algorithm has been developed that takes time $O(V^3)$ [7,8,10]. This algorithm incorporates the blossom algorithm. It works by maintaining numerical quantities that indicate when grow, blossom, and expand steps should be done. (In the special case where all edges have equal weight, the weighted matching algorithm is identical to the blossom algorithm.)

The data structure for blossoms extends to related graph problems. An example is the problem of computing a minimum edge cover on a graph. Here a cover is defined as a set of edges meeting every vertex of the graph at least once. An $O(V^3)$ algorithm incorporating the data structure for blossoms is being developed.



(a)



(b)

Figure 1. Matchings on G1.

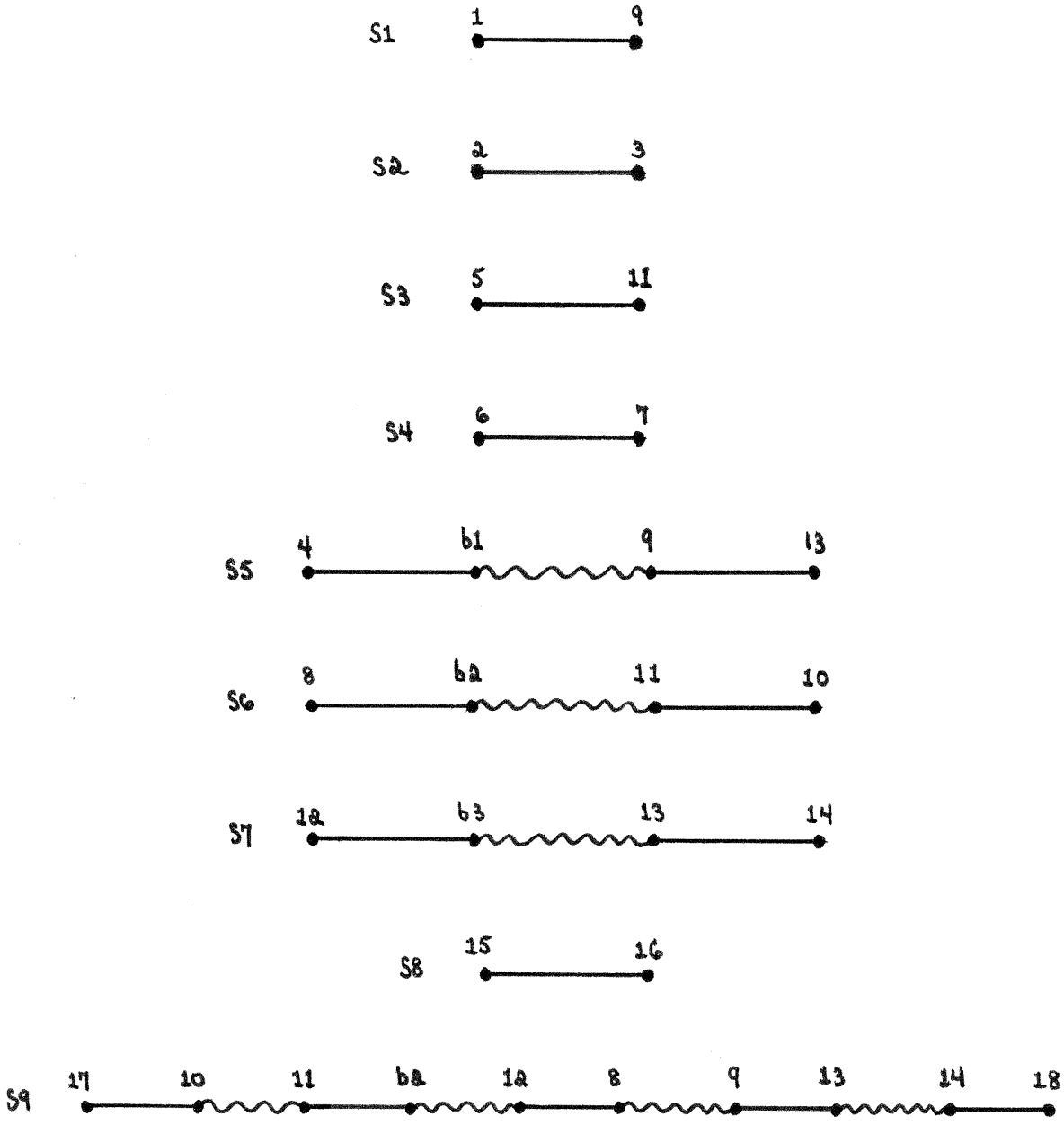


Figure 2. Augmenting paths found in G1.

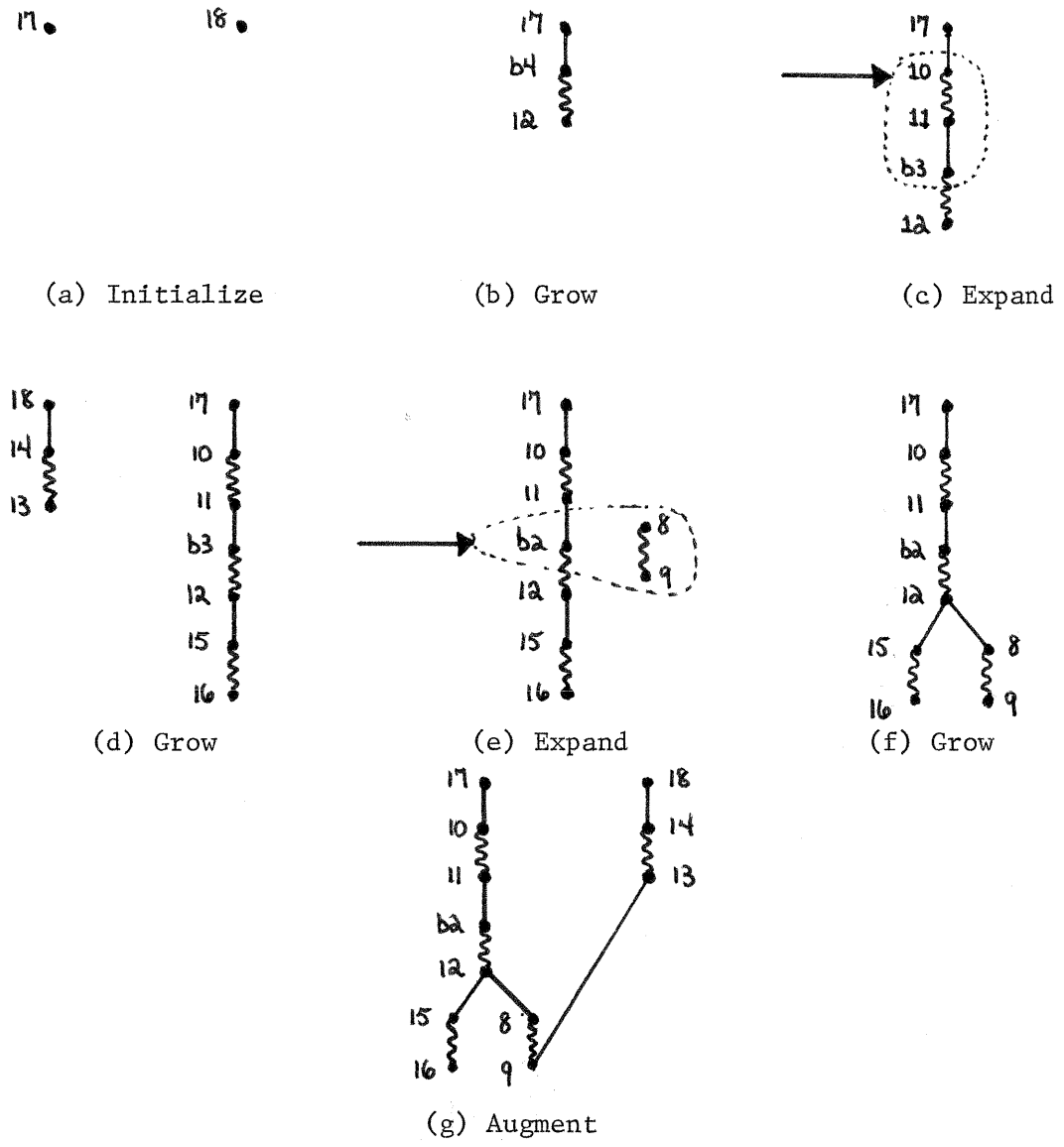


Figure 3. Search S9.

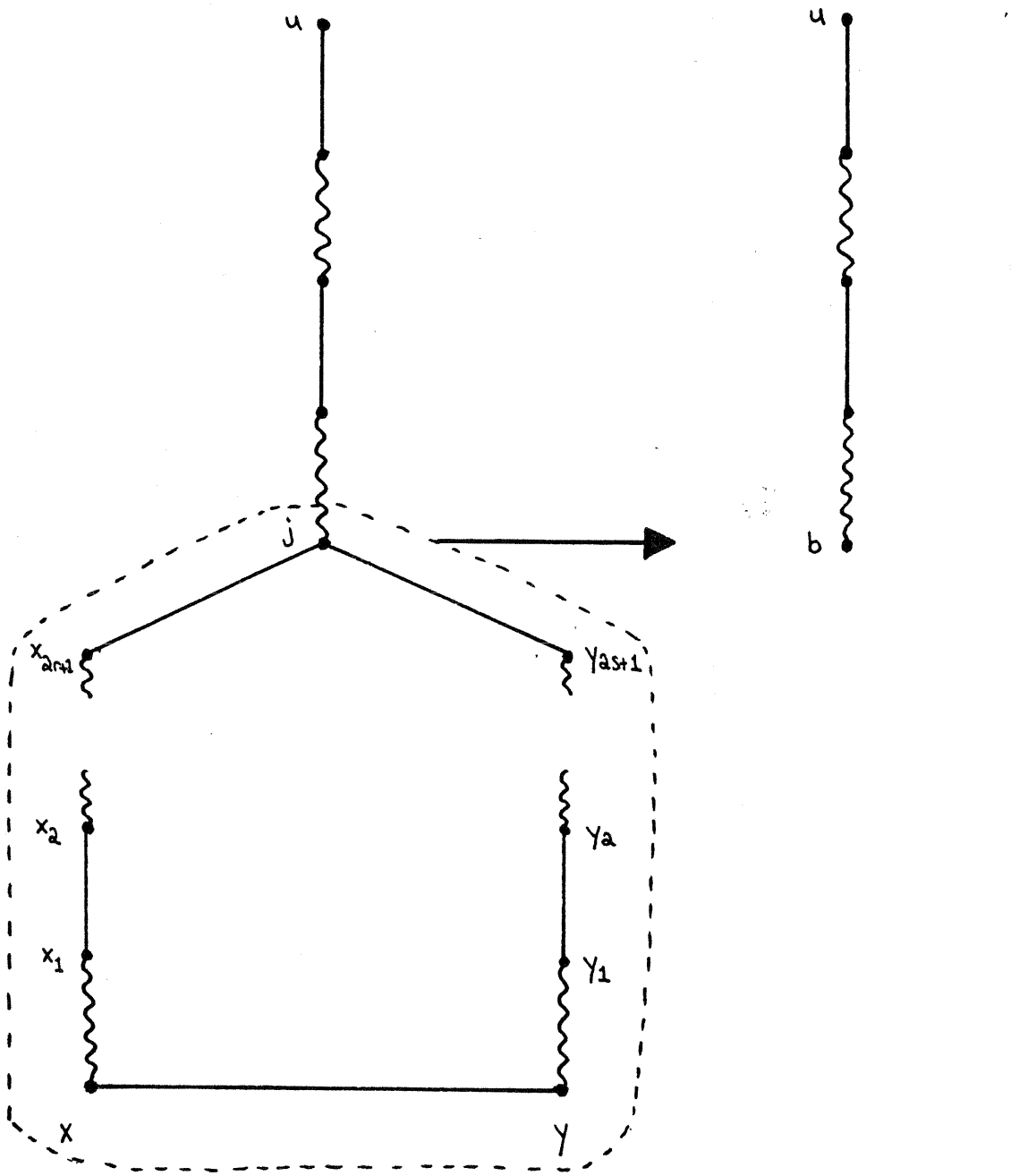


Figure 4. Blossom Step.

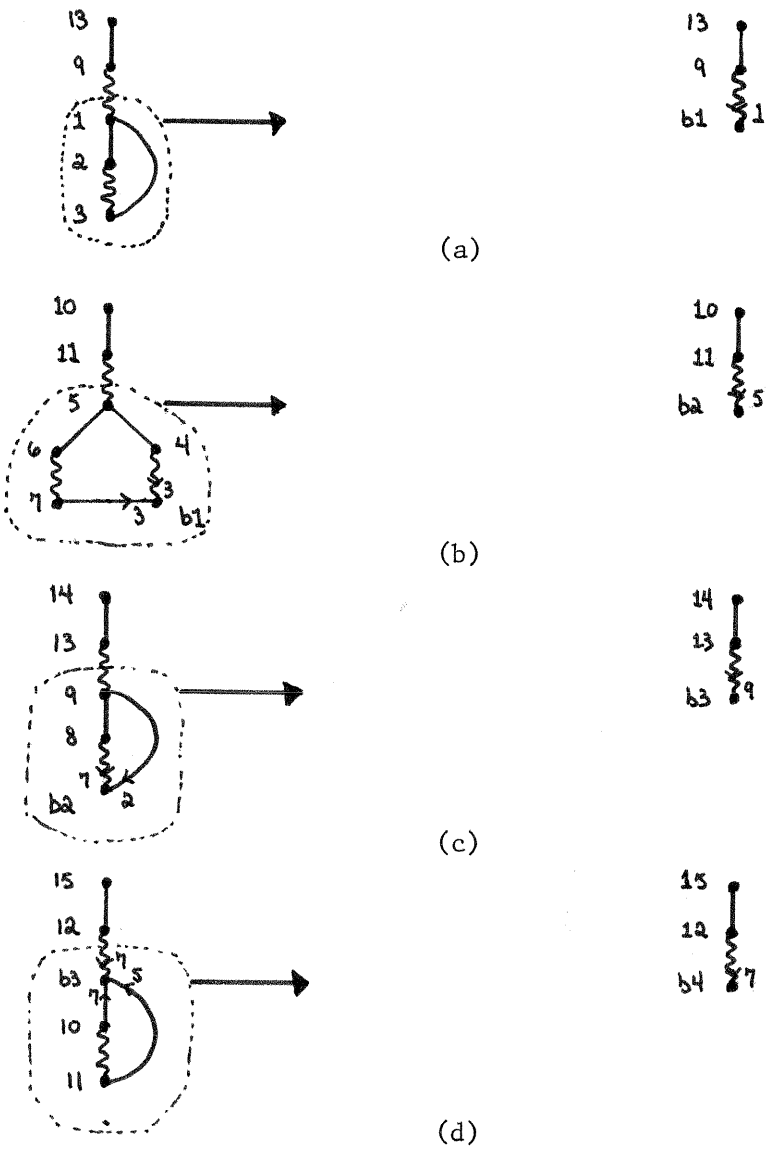


Figure 5. Formation of blossoms b1-b4.

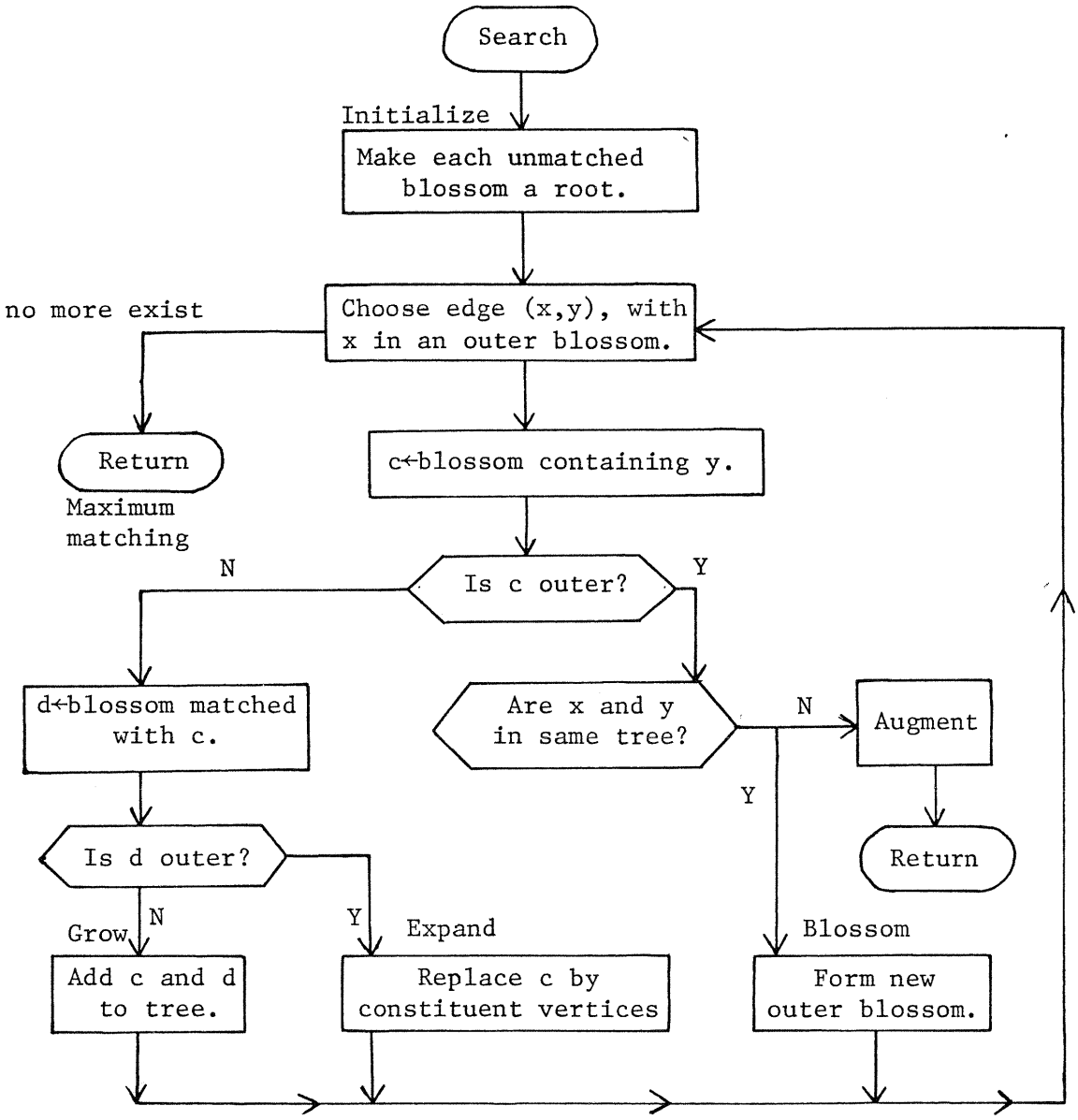
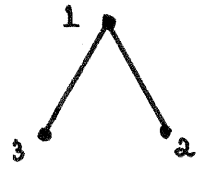
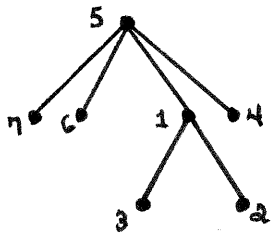


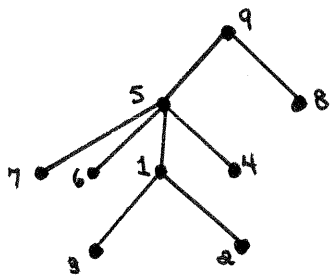
Figure 6. Basic flowchart



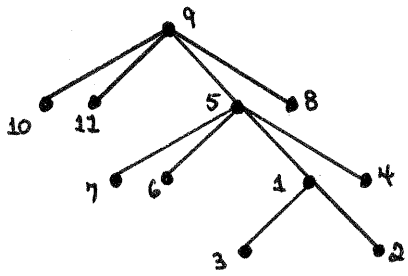
(a) b1



(b) b2



(c) b3



(d) b4

Figure 7. Blossom trees for b1-b4.

Vertex	MATE	LABEL	NEXT	LAST
1	(3,4)	(4,5)	3	2
2	(2,5)	(3,1)	4	2
3	(3,2)	(2,1)	2	3
4	(4,3)	(3,7)	8	4
5	(7,8)	(8,9)	7	4
6	(6,7)	(7.3)	1	6
7	(7,6)	(6,5)	6	7
8	(8,7)	(2,9)	0	8
9	(7,12)	(12,15)	10	8
10	(10,11)	(11,5)	11	10
11	(11,10)	(10,7)	5	11

Table 1.

Array entries for blossom b4.

References

[1] Balinski, M.L., 1967. "Labelling to obtain a maximum matching," in R.C. Bose and T.A. Dowling, ed., Combinatorial Mathematics and Its Applications, University of North Carolina Press, North Carolina, pp. 585-602, 1967.

[2] Berge, C., 1957. "Two theorems in graph theory," Proceedings of the National Academy of Science, Vol. 43, pp. 842-844, 1957.

[3] Edmonds, J., 1963. "Paths, trees and flowers," Canadian Journal of Mathematics, Vol. 17, pp. 449-467, 1965.

[4] Edmonds, J., 1965. "Maximum matching and polyhedron with 0,1 - vertices," Journal of Research of the Natural Bureau of Standards, Vol. 69B, pp. 125-130, 1965.

[5] Edmonds, J. and Johnson, E.L. 1970. "Matching: A well-solved class of integer linear programs," Proceedings of the Calgary International Conference on Combinatorial Structures and their Applications, Gordon and Breach, N.Y., pp. 89-92, 1970.

[6] Gabow, H., 1972. "An efficient implementation of Edmonds' algorithm, algorithm for maximum matching on graphs," Submitted for publication.

[7] Gabow, H., 1973. "Implementations of algorithms for maximum matching on non-bipartite graphs," Ph.D. dissertation, Stanford University, 1973.

[8] Gabow, H. and Lawler, E.L., 1973. "An efficient algorithm for maximum matching on weighted graphs," in preparation.

[9] Knuth, D., 1968. The Art of Computer Programming, Vol. 1, "Fundamental Algorithms," Addison-Wesley, Reading, Mass., 1968.

[10] Lawler, E.L., 1973. Combinatorial Optimization Theory, to be published.

[11] Witzgall, D. and Zahn, C.T. Jr., 1964. "Modification of Edmonds' algorithm for maximum matching of graphs," Journal of Research of the National Bureau of Standards, Vol. 69B, pp. 91-98, 1965.