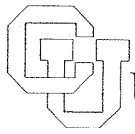
An Efficient Implementation of Edmonds' Algorithm for Maximum Weight Matching on Graphs

Harold Gabow

CU-CS-075-75



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

AN EFFICIENT IMPLEMENTATION OF EDMONDS' ALGORITHM FOR MAXIMUM WEIGHT MATCHING ON GRAPHS

Harold Gabow
University of Colorado

Abstract

A matching on a graph is a set of edges, no two of which share a vertex. If each edge of the graph has a number, called its weight, a maximum weight matching has the greatest total weight possible. An algorithm with run-time $O(V^3)$, where V is the number of vertices, is presented. It is based on Edmonds' algorithm, which has run-time $O(V^4)$. The algorithm uses efficient data structures for blossoms and for choosing edges in a search.

1. Introduction. Maximum weight matching on graphs is the prototype of a number of integer programming problems that can be solved efficiently [4]. Its applications in operations research are illustrated by the following matching problem: Construction workers must be paired up to carry beams. Each pair of workers has a numerical rating of its efficiency. Choose the pairs so the total efficiency rating is as high as possible.

Edmonds has presented an efficient algorithm for maximum weight matching [3]. It is based on an algorithm for maximum cardinality matching [2]. Both algorithms have a worst-case time bound of $O(V^4)$, where V is the number of vertices. We describe implementations of these algorithms with time bound $O(V^3)$. The cardinality matching algorithm has the same time bound as other efficient algorithms [1,5,8], although it is more involved. The weighted matching algorithm has the best time bound currently known.

Section 2 gives definitions from graph theory. Section 3 summarizes Edmonds' method for cardinality matching. Then the implementation of this algorithm is described: Section 4 presents the data structure; Section 5 illustrates it by describing blossom expansion. The next two sections discuss weighted matching: Section 6 summarizes Edmonds' method; Section 7 describes a data structure for weighted edges. Section 8 discusses numerical accuracy, and Section 9 discusses the efficiency of the algorithm.

The algorithm is not stated in complete detail because of its length. Instead, examples and sample procedures are given. A complete statement of the algorithm is in [6].

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 (v,w) is incident to v and to w.

When the order of vertices in an edge is significant, it is a directed edge. The directed edge (v,w) goes from v to w; its head is w, and its 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 tree is a graph with a distinguished vertex r, the root, such that there is a unique path P(v,r) joining any vertex v to r.

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 an edge of M; otherwise it is unmatched. M is a maximum cardinality matching if no matching on G contains more edges. Figure 1(a) shows a matching on a graph G1 (Matched edges are drawn wavy). It is not maximum cardinality, since a matching with more edges exists.

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 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. In Figure 1(a), (12,6,7,9,8,11) is an augmenting path. Augmenting gives a maximum matching.

A fundamental fact is this: A matching M has an augmenting path if and only if M is not maximum cardinality [2]. Thus, a maximum cardinality matching can be found by repeatedly searching for an augmenting path and augmenting the matching.

In a weighted graph, each edge has a real number called its weight. The weight of a matching is the sum of the weights of all matched edges. M is a maximum weight matching if no matching has greater weight. Figure 1(b) gives weights for G1. The matching shown has weight 60. This is not maximum since the maximum cardinality matching has weight 62.

3. Cardinality matching. This section describes Edmonds' algorithm for maximum cardinality matching, using blossoms. The flow chart of Figure 2 is explained. Graph G1 of Figure 1 is used as an example.

The algorithm starts with all edges in the given graph G unmatched. It searches for an augmenting path. When such a path is found, the matching is augmented. Then a search is made for an augmenting path in the new matching. The search-augment process is repeated until no augmenting path exists. At this point a maximum matching has been found.

During a search, the graph is repeatedly transformed. Under certain conditions, a collection of vertices gets replaced by a single

vertex, called a "blossom" (a precise definition is given below). Also, under certain conditions, a blossom gets replaced by its constituent vertices. We call the graph currently being searched the working graph. We refer to vertices of the working graph as either vertices or blossoms.

Table I shows how a maximum matching on G1 is found. (The last three columns are for weighted matching). The algorithm makes six searches, S1 - S6. In S3, blossom b1 is formed, from five vertices. This is illustrated in Figure 3(a). Blossoms b2 and b3 form in search S5 (Figures 3(b) - (c)); they get expanded into their constituent vertices in search S6.

A search works by constructing a number of trees made up of alternating paths. An alternating tree is defined as a tree in which (i) the root is an unmatched vertex; (ii) any path from a vertex to the root is alternating. Let v be a vertex in the tree, and let u be the root. Then P(v,u) denotes the path from v to u. Vertex v is 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, v is 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 stops, and the matching is augmented.

Figure 4 shows the alternating trees grown in search S6. The search stops when in Figure 4(d), edge (6,12) is scanned. This edge

joins outer vertex 6 with the tree for unmatched vertex 12. It gives an augmenting path.

A search constructs alternating trees in four different steps: start, grow, blossom, and expand. These steps are shown in Figure 2. They are described below.

Start: A search begins by making every unmatched vertex the root of an alternating tree.

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

In Figure 4(c), edge (9,7) is scanned. A grow step is done, giving Figure 4(d).

Blossom: Suppose the search scans an edge (x,y), where x and y are outer vertices in the same tree. Then the search does a blossom step. Figure 5 illustrates this step schematically. The paths P(x,u) and P(y,u) join at some outer vertex j. In the blossom step, all vertices up to and including j in P(x,u) and P(y,u) are replaced by a single vertex b. Vertex b is called a blossom, and vertex j is its join. 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 three blossom steps for G1 are shown in Figure 3. For example, in Figure 3(b), edge (7,b1) is scanned. This edge comes from edge (7,4) in the original graph. (This is indicated by the "4" at the head of (7,b1).) Blossom b2 is formed, replacing vertices 7, 6, and b1.

The rationale for blossom steps is that any matching on the new working graph, W', gives a matching on the original working graph, W. Suppose the matched edge incident to blossom b corresponds to (v,v') in W. Here vertex $v \in b$, $v' \notin b$. Figure 5 shows in W, there is an alternating path P(v,j) from v to the join j, that starts with a matched edge. (If v is outer, P(v,j) is the beginning of P(v,u). If v is inner, P(v,j) is a path from v to x (or y), plus P(x,j) (or P(y,j)); for example, $P(x_1,j)$ consists of (x_1,x), (x,y), and P(y,j).) Suppose 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 (v,v') to be matched. It gives the desired matching on W.

Expand: Suppose a blossom b is made inner in a grow step of some search. (This search is not the one where blossom b is formed, since 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 b. Second, some constituent vertices are put in the tree to replace b. This prevents losing the part of the tree "hanging from" b.

Figure 4(a) - (b) shows the expansion of b3. The matching is not changed; the three constituent vertices of b3 are put in the tree. Figure 4(b) - (c) shows the expansion of b2. The matching is changed from Figure 3(b), to reflect the augment in search S5 (Table I). Also, vertex b1 is put in the tree.

Expand steps insure inner blossoms do not "hide" augmenting paths. For example, in search S6, the augmenting path (12,6,7,9,8,11)

is hidden in Figure 4(b), where it corresponds to (12,b2,b2,9,8,11). After b2 is expanded, the augmenting path is found.

The cardinality matching algorithm is summarized in Figure 2. A complete discussion of the underlying theory is in [3]. Now we describe an efficient implementation.

4. A Data Structure for Blossoms. This section presents a data structure that handles blossoms efficiently. Data is stored in five arrays, called BLOSSOM, MATE, LABEL, NEXT, and LAST. These arrays are described and illustrated.

The algorithm begins by numbering the vertices and edges of the given graph G. The vertices are numbered from 1 to V. Each array in the data structure has V entries. A vertex's number is used as an array index. Usually we identify a vertex and its number.

The edges of G are numbered from 1 to 2E. Each edge (x,y) gets 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 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.

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 the given graph G. Then i(b) is the number of vertex b.

(ii) Suppose b is formed in a blossom step, and has join j. Then i(b) = i(j).

For example, in Figure 3, i(b3) = i(b2) = 7.

These indices can be used to identify blossoms, since in a given working graph, distinct blossoms have distinct indices. This is done in the BLOSSOM array, which specifies which blossom contains a given vertex.

Definition 2: Let v be a vertex of G, contained in blossom b in the current working graph. Entry BLOSSOM[v] is i(b), the index of blossom b.

For example, in Figure 3(c), vertices 1-9 are in blossom b3, so their BLOSSOM value is 7.

Blossom indices are also used as array indices, in the MATE and LABEL arrays. These arrays are maintained as follows. When the algorithm begins, any vertex v is a blossom index. MATE and LABEL information for 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, which specifies the matching.

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

For example, consider vertex 5 in Figure 3. Since i(b1) = 5, in Figure 3(a),MATE[5]=0, and in Figure 3(b),MATE[5]=(4,6). This entry does not change in Figure 3(c),since vertex 5 is no longer a blossom index. Table II gives MATE entries for the vertices in blossom b3.

Now we describe the LABEL array, which specifies the structure of the alternating trees and blossoms.

Definition 4: Let v be a vertex of 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). So if P(b,u) = $(b,b_1,b_2,\ldots)$, edge (x,y) corresponds to (b_1,b_2). Also, $x \in b_1$. (If b is an unmatched outer vertex, P(b,b) has no edges, and LABEL[v]=0).

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

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

For example, consider Figure 3(b). Blossom b1 is outer, so LABEL[5] = (6,7). Vertex 6 is inner. Before the blossom step, LABEL[6] < 0; after, LABEL[6] = (4,7). Table II gives LABEL entries for vertices in blossom b3.

The following pseudo-Algol code shows how a matching is augmented. It illustrates the use of MATE and LABEL.

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

Now we describe the NEXT and LAST arrays. These arrays list the vertices in a blossom. They are based on the following method for listing vertices.

Definition 5: The vertex list for a blossom b, L(b), is defined recursively as follows:

(i) Suppose b is a vertex in G. Then L(b) is the list b.

(ii) Suppose b is a blossom formed from vertices x, $x_1,\ldots,x_{2r+1}$, y, $y_1,\ldots,y_{2s+1}$, j, as in Figure 5. Then L(b) is

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

(Here L(j)-i(j) denotes the list L(j) with i(j) deleted. Note i(b) = i(j).)

It is easy to see L(b) lists every vertex (of G) that is in b exactly once. For example, in Figure 3, L(b3) = 7,9,8,5,2,1,4,3,6.

NEXT and LAST derive from L as follows.

Definition 6: Let v be a vertex of G, contained in blossom b in the current working graph. Entry NEXT[v] is the vertex after v in L(b). (If v is last in L(b), NEXT[v]=0.)

Definition 7: Let v be a vertex of G; let b' be the last blossom with index v, i.e., i(b')=v. Entry LAST[v] is the last vertex in L(b').

Note that in Definitions 6-7, list L(b') is a sublist of L(b). Thus L(b') is v, NEXT[v],...,$NEXT^m$[v], where $NEXT^m$[v]=LAST[v]. This facilitates blossom expansion. Table II shows NEXT and LAST for blossom b3.

The following code shows how NEXT and LAST are updated when a new blossom is formed.

```
procedure build-L (x,y,j);
comment  Parameters x and y are vertices in the original graph, and
         j is a blossom index.  As in Figure 5, edge (x,y) completes
         a new blossom b, and j is its join.  Build-L builds list L(b)
         in NEXT and LAST;
begin
rear := j; comment rear is the current end of L(b).  Initially only
                  j is in L(b);
save := NEXT[j]; comment Save list L(j),for later insertion in L(b);
for side := BLOSSOM[x], BLOSSOM[y]do
comment Process the vertices in both paths;
     begin
     b1 := side; comment b1 steps through the outer vertices of the path;
     while b1≠j do
          begin
          b2 := BLOSSOM[head of edge MATE[b1]]; comment b2 steps through
                                                      the inner vertices
                                                      of the path;
          NEXT[rear] := b1; comment Add L(b1) to L(b);
          NEXT[LAST[b1]] := b2; comment Add L(b2);
          rear := LAST[b2];
          b1 := BLOSSOM[head of edge LABEL[b1]];
          end;
     end;
comment  Now add L(j) to L(b);
if save = 0 then LAST[j] := rear  else NEXT[rear] := save;
end;
```

5. Expanding Blossoms. This section describes a method for expanding blossoms, illustrating how the blossom arrays are used.

We first consider an example. A blossom b forms, as in Figure 5, in the search S. In subsequent searches, b is on one or more augmenting paths. Finally, in some search, b is expanded. Before expansion, b is an inner vertex. The 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 $x_1,y_1 \in b$, and $x_1',y_1' \notin b$.

The expand step changes MATE and LABEL entries for blossom b. We discuss these changes.

Before expanding, MATE contains the matching of search S on the constituent vertices of b. The expand step must change MATE entries for b, so vertex x_1 is not matched with a vertex of b. (This allows (x_1,x_1') to be matched.) This can be done by rematching the path $P(x_1,j)$. MATE must be changed so these edges are matched:

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

LABEL 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 b, every vertex is joined to x_1 by an alternating path. 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.

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 made 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 (although LABEL entries are still negative). In effect, x_1 has become the join of the blossom.

The third procedure updates LABEL, putting the alternating path between y_1 and x_1 into the tree. This path is computed from LABEL, 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 these procedures are in [7]. The first procedure is shown below.

```
procedure start-expand (b);
comment Parameter b is the index of the blossom to expand;
begin
comment Start-expand contains two main loops.  A two-pass organization
        is not required, but is used for clarity;
comment Pass 1:  Step through the constituent vertices of b (in Figure
        5, x,x1,...,x2r+1, y,y1,...,y2s+1,j) to find the last one (j);
b1 := NEXT[b];
comment b1 is the first constituent vertex of b;
b2 := NEXT[LAST[b1]];
comment b2 is alwasy one vertex ahead of b1;
e := eb := LABEL[b2];
comment In Figure 5,eb is the edge (x,y) that formed blossom b.  Variable
        e steps through the unmatched edges in b;
for side := 1, 2 do
comment Step through both "sides" of b;
     begin
     while LABEL[b2] = e do
     comment This test insures b1 and b2 are in the current side of b.
             Advance;
          begin
          e := edge opposite to LABEL[b1];
          b1 := NEXT[LAST[b2]];
          b2 := NEXT[LAST[b1]];
          end;
     comment Now b1 is not in the current side.  Try the other side;
```

```
        e := edge opposite to eb;
        end;
    first := b1;
    comment first is the first vertex in L(b)-i(b) that is in blossom j
            (see Figure 5).  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
    comment b1 steps through the constituent vertices of b;
        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;
```

The expand procedures are also used after the final search of the algorithm. At this point, a maximum matching has been found. However, the working graph may contain unexpanded blossoms. These blossoms are expanded so MATE is correct. This is illustrated by the last step in Table I,where blossom b1 is expanded.

6. Weighted Matching. This section describes the theory and algorithm developed by Edmonds for maximum weight matching. Table I (in particular, the last three columns) illustrates the algorithm on graph G1.

The theory is based on duality theory of linear programming [3] (A combinatoric interpretation can also be made [6]). Here we give an argument that is simple, although lacking in motivation.

Let G be a weighted graph, where edge (x,y) has weight w(x,y). Let M be a matching. Let s be a vertex weight function, assigning a non-negative number s(b) to each set b of an odd number of vertices of G. (Note, any blossom contains an odd number of vertices.) Let s(x) denote the weight s({x}) for vertex x. Suppose s and M satisfy the following

Optimality Conditions:

(1) For every edge (x,y) in G,

$$s(x) + s(y) + \sum_{x,y \in b} s(b) \geq w(x,y).$$

If $(x,y) \in M$, then equality holds in this constraint.

(2) If $s(x) > 0$, then vertex x is matched.

(3) If $s(b) > 0$, then b is maximally matched, i.e., b contains $\lfloor \frac{|b|}{2} \rfloor$ edges of M.

These conditions imply M has maximum weight. No matching N has greater weight, since

$$\sum_{(x,y)\in N} w(x,y) \leq \sum_{(x,y)\in N} s(x)+s(y) + \sum_{\substack{(x,y)\in N \\ x,y\in b}} s(b)$$

$$\leq \sum_{x} s(x) + \sum_{b} \lfloor \frac{|b|}{2} \rfloor s(b)$$

$$= \sum_{(x,y)\in N} w(x,y).$$

So to find a maximum weight matching, the algorithm constructs M and s satisfying (1) - (3), as follows.

The algorithm starts with no edges matched; for each vertex x in G, $s(x) = W/2$, where W is the maximum edge weight; for all other odd sets b, $s(b) = 0$. For example, Table I shows at the start of search S1, all vertex weights are 7.

The initial choice of s satisfies optimality conditions (1) and (3), but not (2). Each search of the algorithm maintains (1) and (3), and eliminates violations of (2). Eventually (2) is satisfied, and the matching has maximum weight.

A search works as follows: Call an edge (x,y) tight if equality holds in condition (1). A search puts only tight edges in alternating trees. If no augmenting path can be found, s is modified so more edges are tight; the search continues. When an augmenting path is found, the matching is augmented. Newly matched edges are tight, so condition (1) stays true. Since more vertices are matched,

there are fewer violations of (2).

For example,consider search S6. It begins (Figure 4(a)) by putting tight edges (11,8), (1,10) in the tree. No other tight edges can be put in the tree. So s is modified, and an expand step (Figure 4(b)) is done. Eventually an augmenting path is found, giving a maximum weight matching.

Now we describe how s is modified. Call a blossom b proper if it was formed in a blossom step (so b contains an odd number ≥ 3 of vertices of G). Note in any working graph, any vertex x is in some blossom; either the blossom is x itself, or it is proper.

The algorithm modifies s by computing a value δ (see below), and making the following

Weight Transformation:

(1) If x is a vertex in an outer blossom, $s(x) \leftarrow s(x)-\delta$.

(2) If x is a vertex in an inner blossom, $s(x) \leftarrow s(x)+\delta$.

(3) If b is a proper outer blossom, $s(b) \leftarrow s(b)+2\delta$.

(4) If b is a proper inner blossom, $s(b) \leftarrow s(b)-2\delta$.

For appropriate δ, the transformation preserves optimality constraints (1) and (3). For example, edge (1,10) is tight after the grow step of Figure 4(a). When weights are modified for the expand step of Figure 4(b), the changes in s(1) and s(10) balance, and edge (1,10) stays tight.

In general, the new weights s satisfy optimality conditions (1) and (3), provided δ satisfies the following

Feasibility Constraints:

(1) $\delta \leq \min\{s(x) | x \text{ is a vertex in an outer blossom}\}$.

(2) $\delta \leq \min\{s(b)/2 | b \text{ is a proper inner blossom}\}$.

(3) $\delta \leq \min\{s(x)+s(y)-w(x,y) |$ for edge (x,y), x is in an outer blossom and y is in a blossom that is not in a tree$\}$.

(4) $\delta \leq \min\{(s(x)+s(y)-w(x,y))/2 |$ for edge (x,y), x and y are in distinct outer blossoms$\}$.

Constraint (1) is needed because weight transformation (1) decreases vertex weights; the constraint guarantees each vertex has non-negative weight. Similarly, constraint (2) guarantees each blossom has non-negative weight. Constraints (3) and (4) guarantee all edges satisfy optimality condition (1).

The algorithm chooses the largest possible value δ permitted by constraints (1) - (4). This gives equality in some constraint. Depending on the constraint, a grow, blossom, expand or augment step can be done, or else the search can stop with a maximum matching. We show this below, discussing constraints in the order (2), (3), (4), (1).

Suppose equality holds in constraint (2). Some proper inner blossom b has a new weight $s(b) = 0$. Optimality constraint (3) no longer applies to b, and it need not be maximally matched. So the search can expand blossom b. In Figure 4(b), blossom b3 is expanded when s(b3) becomes 0.

Suppose equality holds in constraint (3). Some edge (x,y) has become tight, where (x,y) joins an outer blossom to a blossom not in

a tree. The search can do a grow step for (x,y). In search S5, a grow step for edge (7,4) is done when (7,4) becomes tight.

Suppose equality holds in constraint (4). Some edge (x,y) has become tight, where (x,y) joins two distinct outer blossoms. The search can do a blossom step, if these blossoms are in the same tree; otherwise, it can do an augment. In the last step in search S5, an augment is done when edge (1,10) becomes tight.

Finally we consider constraint (1). First note at any point in the algorithm, all unmatched vertices of G, u, have the same weight s(u). For when the algorithm starts, all vertex weights are the same; whenever s changes, the weight of an unmatched vertex u decreases by δ (transformation (1)). This reasoning also shows the value s(u) is the smallest of all vertex weights s(x).

Now suppose equality holds in constraint (1). All unmatched vertices u have new weight s(u) = 0. So optimality condition (2) holds. The search can stop, since a maximum matching has been found.

To summarize, the algorithm searches the tight edges to find an augmenting path. If the search cannot proceed, δ is computed from the feasibility constraints; the weight transformation is applied; then the search continues. Eventually a matching satisfying the optimality conditions is found,and the algorithm halts.

7. A Data Structure for Edges and Weights. This section describes a data structure that handles weighted edges efficiently. The arrays S, EDGE, and EDGEΔ are described and illustrated.

The S array derives from the vertex weight function s. It uses the variable Δ to keep track of weight transformations.

Definition 8: The variable Δ gives the total decrease in the weight of an unmatched vertex since the current search began.

In Table I, when b2 is expanded in search S6, $\Delta = 2$.

Definition 9: Let v be a vertex of G, contained in blossom b in the current working graph. Entry S[v] is a non-negative number defined as follows:

(i) Suppose b is not in a tree. Then S[v] = s(v).

(ii) Suppose b is an outer blossom. Then S[v] = s(v)+Δ.

(iii) Suppose b is a proper inner blossom. Then S[v] = s(v)+s(b)/2.

(iv) Otherwise (b is an inner blossom with one vertex), S[v] =∞.

Table III shows S at the end of search S6, when $\Delta = 2$.

S is organized so it keeps track of s, without changing in most weight transformations. We show this below.

When a search begins, S gives each vertex's weight. Suppose a blossom b is made outer. Let vertex vεb. Following Definition 9, the algorithm increases S[v] by Δ. Vertex v remains in an outer blossom for the rest of the search. So each weight transformation decreases s(v) by δ, and increases Δ by δ; S[v] does not change. At the end of the search b is no longer outer. The algorithm decreases

S[v] by Δ, giving s(v).

Proper inner blossoms are handled similarly. Suppose b is made inner. Let $v \in b$. The algorithm increases S[v] by s(b)/2. S[v] does not change until blossom b is expanded.* This occurs when s(b) = 0, and S[v] = s(v). This value of S[v] is correct until v is put back in a tree.

Thus we see S does not change in most weight transformations.

Values of the vertex weight function s on proper blossoms b are not stored explicitly. When needed, they can be computed from optimality constraint (1) (Section 6). For this, it is useful to define the slack in edge (x.y) as S[x] + S[y] - w(x,y). The following code shows how S is changed when a blossom b is made inner; it also illustrates how s(b) is computed.

```
procedure make-inner (b);
comment Parameter b is the index of a blossom that is being made inner;
begin
if LAST[b] ≠ b then comment b is proper;
    begin
    ε:=-(slack in edge MATE[NEXT[b]])/2; comment ε is s(b)/2;
    v:=b; comment v steps through all vertices in b;
    while v ≠ 0 do
        begin
        S[v]:=S[v]+ε;
```

* An exception is if blossom b is absorbed in an outer blossom before it is expanded (as in search S5). When this occurs, S[v] gets changed.

```
            v:=NEXT[v];
            end;
        end
    else comment b is a single vertex;
        S[b]:=∞;
    end;
```

Slack is also used to determine when edges become tight, and can be used in the search. For example, suppose (x,y) joins an outer blossom to a blossom not in a tree. It becomes tight when s(x) + s(y) - w(x,y) = 0, or equivalently, when its slack is Δ. If (x,y) joins two outer blossoms, it becomes tight when its slack is 2Δ. These facts are used in the definitions of EDGE and EDGEΔ. EDGE contains all edges that may be used in the remainder of the search; EDGEΔ indicates when the edges can be used.

Definition 10: Let v be a vertex of G, contained in blossom b in the current working graph. Entry EDGE[v] is a list of directed edges, defined as follows.

(i) Suppose b is not in a tree, or b is a proper inner blossom. Then EDGE[v] is the edge (x,v) with the smallest slack possible such that x is in an outer blossom. (If no such edge exists, EDGE[v] has no edges.)
(ii) Suppose b is outer, and v = i(b). Then EDGE[v] is a list of edges, $(x_1,y_1),\ldots,(x_n,y_n)$.* The list is sorted, $y_1<\ldots<y_n$. For each edge (x_i,y_i), $1 \leq i \leq n$, $x_i \in b$, $y_i \notin b$; x_i was put in an outer blossom after y_i;

* In most programming languages, a list of edge numbers cannot be stored in one word; EDGE[v] would be the head of a linked list of edges. For simplicity, we overlook this point.

and (x_i, y_i) has the smallest slack of all edges (x, y_i), where $x \in b$.
(iii) Otherwise, EDGE[v] has no edges.

Definition 11: Let v be a vertex of G, contained in blossom b in the current working graph. Entry EDGEΔ[v] is a non-negative number, defined as follows.

(i) Suppose b is outer, and v = i(b). Then EDGEΔ[v] is half the smallest slack in list EDGE[v].
(ii) Otherwise, EDGEΔ[v] is the slack of edge EDGE[v].
(If EDGE[v] has no edges, EDGEΔ[v] = ∞)

Table III illustrates these arrays. Note outer vertex 6 has only one edge in its list.

EDGE and EDGEΔ have the following property. Suppose at some point in the search, a grow, blossom, expand, or augment step is done. The flowchart of Figure 2 shows how an edge (x,y) is associated with the step. As in Figure 2, let $x \in b$, $y \in c$, with b,c blossoms, and b outer. For a grow step, c is not in a tree; for an expand step, c is inner; etc. Then before the step for (x,y), when b is made an outer blossom, some EDGE[v] contains (x,y); also, EDGEΔ[v] is the value of Δ when the step for (x,y) can be done. We show this below, by considering the different types of steps in turn.

First suppose a grow step is done for edge (x,y). Edge (x,y) is the first edge joining c to an outer blossom that becomes tight, i.e., the quantity $(s(x) + s(y) - w(x,y))$ is the smallest possible. When blossom b is made outer, y is in some blossom c'. (Note, if $c' \neq c$, then c' contains c, and c' is expanded before the grow step is done.)

Either c' is not in a tree, or c' is a proper inner blossom. In either case, the slack in edge (x,y) is smallest possible. Thus EDGE[y] = (x,y) (Definition 10(i)), and EDGEΔ[y] is the value of Δ when (x,y) can be used (Definition 11(ii)).

Next suppose a blossom or augment step is done for (x,y). Similar reasoning shows (x,y) is stored in EDGE[b] and EDGEΔ[b].

Finally, suppose the inner blossom c is expanded. This occurs when s(c) = 0. The edge (x,y) is not uniquely determined for an expand step; let (x,y) be a tree edge, where $y \in c$. Its slack is Δ + s(c)/2, the smallest possible slack for an edge joining c to an outer blossom. Thus EDGE[y] = (x,y). Further, when EDGEΔ[y] = Δ, then s(b) = 0, and (x,y) can be used (as in Figure 2) to expand blossom c.

These remarks show EDGE and EDGEΔ allow the algorithm to determine the sequence of steps in the search. This is illustrated by the following code for finding the next step.

```
procedure choose-edge;
comment Choose-edge finds the next step to be done.  It sets e to
        the edge associated with the step, and Δ to the value
        when e can be used;
begin
comment first find vertex v with smallest EDGEΔ[v];
v := 1;
for i:= 2 to V do
     if EDGEΔ[v] > EDGEΔ[i] then v := i;
Δ := EDGEΔ[v]; comment this is the next value of Δ in the search;
```

```
if Δ < ∞ then
     begin
     b := BLOSSOM[v];
     if b is outer then comment a blossom or augment step is next.
                              Find e in list EDGE[b];
          begin
          e := first edge in EDGE[b];
          while (slack in edge e) > 2Δ do
               e := next edge in EDGE[b];
          end;
     else comment b is not outer;
          e := EDGE[b]
else comment Δ = ∞;
     go to end-search; comment the search ends without augmenting,
                              since no more steps can be done;
end;
```

Arrays EDGE and EDGEΔ are easy to maintain. For example, suppose a blossom step forms a new outer blossom b, as in Figure 5. To build list EDGE[b], first lists for the previously inner blossoms, $x_1, x_3, \ldots, x_{2r+1}, y_1, y_3, \ldots, y_{2s+1}$, are formed; then the lists for all constituent vertices are merged, according to Definition 10(ii). The merge can be done efficiently since all lists are sorted, as specified in Definition 10(ii).

The following code shows how the list EDGE[b] is formed when a blossom b becomes outer in a grow or expand step. (This code is also used in a blossom step, for a previously inner blossom b.)

```
procedure scan (b);
comment b is the index of a blossom being made outer in a grow or
          expand step.  The edges incident to b are scanned and stored
          in EDGE and EDGEΔ;
begin
empty list EDGE[b];
EDGEΔ[b] := ∞;
comment start with no edges in b's list;
x := b;
comment x steps through all vertices in b;
while x ≠ 0 do
     begin
     for each edge (x,y) do
     comment Some edges (x,y) are merged into EDGE[b].  To do this
             efficiently, y takes on values in numerical order;
          begin
          c := BLOSSOM[y]; comment (x,y) joins blossoms b and c;
          ε' := slack in edge (x,y);
          if c is outer then
               begin comment (x,y) may possibly be put in EDGE[b];
               if c ≠ b then
                    begin
                    find the place for (x,y) on EDGE[b]; comment EDGE[b]
                              is sorted on y;
                    if no edge (x',y) is in EDGE[b] then
                         insert (x,y) in EDGE[b]
```

```
                else comment an edge (x' ,y) is in EDGE[b];
                if (slack in (x' ,y)) > ε then
                     replace (x' ,y) with (x,y) in EDGE[b];
                EDGEΔ[b] := min (ε/2, EDGEΔ[b]);
                end
           end
           else comment c is not outer;
           if (BLOSSOM [head of MATE[c]] is not outer)
                comment c is not in a tree;
                or (LAST[c] ≠ c) comment c is proper; then
           if EDGEΔ[y] > ε then
              begin
              EDGE[y] := (x,y);
              EDGEΔ[y] := ε;
              end;
     end;
x:= NEXT[x];
end;
```

Thus we see S, EDGE and EDGEΔ handle numeric weights efficiently.

8. Numerical Accuracy. Since the algorithm performs numerical divisions, roundoff errors may be possible. This section shows we can guarantee the arithmetic involves integers only, and so is exact. More precisely, the following is true.

Theorem: Suppose all edge weights w(x,y) are even integers. Then all numbers computed by the algorithm are integers.

Note that if some edge weights are odd, we can double all edge weights. The new matching problem has even weights, and is equivalent to the original problem.

Proof: The algorithm makes three divisions. We show each division gives an integer result.

The first division is when the initial vertex weights are set to W/2, where W is the maximum edge weight. Since W is even, the division is exact.

The second division is when EDGEΔ[b] is computed, following Definition 11(i), as $(\min \{ S[x] + S[y] - w(x,y) \mid (x,y)$ is in EDGE[b]$\})/2$ (see procedure scan). Definition 9 for S shows this division results in an integer plus $(s(x) + s(y))/2$. Thus, it suffices to show the parity of a vertex weight s(v) is the same for all vertices v in outer blossoms c. We do this below.

Let W be the current working graph. Let M be the matching on G corresponding to the matching on W. (Recall M is constructed by expanding each blossom and matching it properly). Let u be the unmatched vertex of G in the root of the alternating tree containing c.

Matching M can be modified so vertex u is matched and v is unmatched. To do this, rematch the alternating path from c to u in W, to get a matching

on W with c unmatched; then expand all blossoms. Let N be the new matching on G.

We compute the difference in weight between M and N. M consists of tight edges only; as in Section 6, its weight is

$$\sum_{(x,y)\in M} s(x) + s(y) + \sum_{\substack{(x,y)\in M \\ x,y\in b}} s(b)$$

$$= \sum_{x \text{ matched}} s(x) + \sum_{b \text{ a blossom}} \left\lfloor \frac{|b|}{2} \right\rfloor s(b)$$

N also consists of tight edges. N has the same matched vertices as M, except for u and v. In each blossom b, N and M contain the same number of edges. So the difference in weight between M and N is $(s(v) - s(u))$. The difference is even, because all edge weights are even. Thus $s(v)$ has the same parity as $s(u)$. Since all unmatched vertices have the same weight $s(u)$, all vertices v in outer blossoms have the same parity weight. Thus, the second division is exact.

The third division is when the S[v] is computed, following Definition 9, for v in an inner blossom, as $s(v) + s(b)/2$ (see procedure make-inner). The weight transformations of Section 6 show $s(b)$ is even for any blossom b. So this division is exact. QED.

9. Efficiency. This section discusses the efficiency of the maximum weight matching algorithm, from theoretical and practical points of view.

The execution time 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 the portions of the search that require time $O(V^2)$.

Let x be a vertex of G. The first time x is absorbed in an outer blossom, the edges incident to x are scanned (as in procedure scan). During a search, an edge (x,y) may be scanned twice, for x and for y. So the time spent scanning edges is $O(E) = O(V^2)$.

A search does at most V/2 grow steps, since a working graph contains at most V/2 outer blossoms. When a blossom is made inner, S and EDGEΔ are modified (as in procedure make-inner). These arrays are processed in time O(V). So the time to change inner blossom weights is $O(V^2)$.

A search does at most V/2 blossom steps, since a blossom reduces the number of vertices by at least two. The BLOSSOM, NEXT, LAST, and LABEL arrays are modified in time O(V). So the time to change blossom arrays is $O(V^2)$.

In a blossom step, EDGE is modified, when the EDGE lists for the constituent vertices are merged. The time to merge two lists is O(V). At most V merges are done, since each merge reduces the number of lists by 1. So the time for merging is $O(V^2)$.

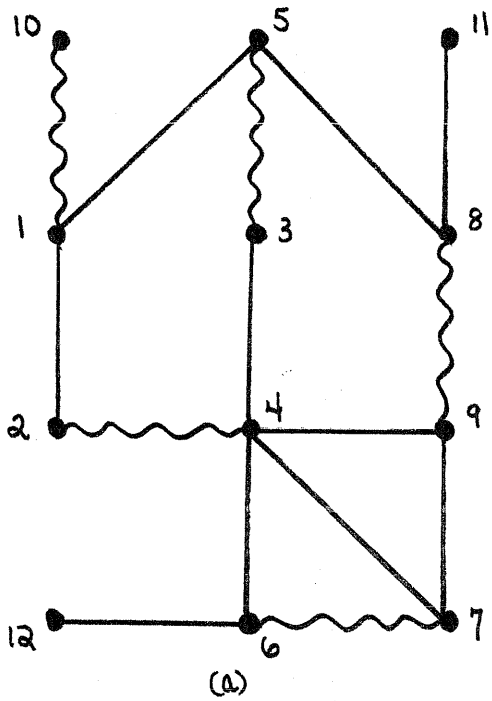
A search does at most V/2 expand steps, since only blossoms formed in previous searches can be expanded. The BLOSSOM, NEXT, LAST, MATE, and LABEL arrays are modified in time O(V). So the time to change blossom arrays is $O(V^2)$.

A search calls procedure choose-edge (Section 7) at most 3V/2 times to find the next step. This procedure scans through the EDGEΔ array, and may scan through one list EDGEΔ[b]. This requires time O(V). So the total time is $O(V^2)$.

Other portions of the search procedure require time $O(V)$. Thus a search requires time $O(V^2)$, and the algorithm runs in time $O(V^3)$.

The algorithm was programmed in ALGOL W, and tested on the IBM 360/165, using "random" graphs. Edges were assigned random integer weights between 1 and 100. Table IV shows results for graphs with 100 vertices. The times are approximately two orders of magnitude greater than corresponding times for an efficient cardinality matching algorithm [6]. The time could probably be reduced one order of magnitude by making obvious changes in the code.

10. Conclusion. The techniques discussed for matching can be extended to a large class of integer programming problems described by Edmonds [4]. One example is finding the minimum weight cover on a graph, where a cover is a set of edges meeting every vertex at least once. An $O(V^3)$ covering algorithm can be constructed by making simple changes to the matching algorithm.



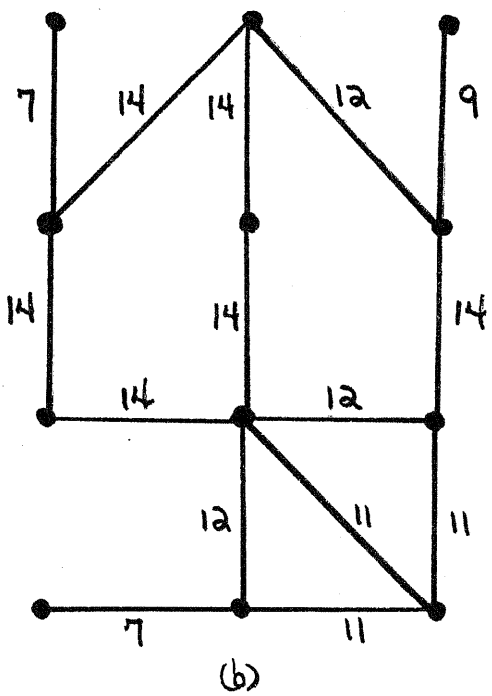


Figure 1. Graph G1

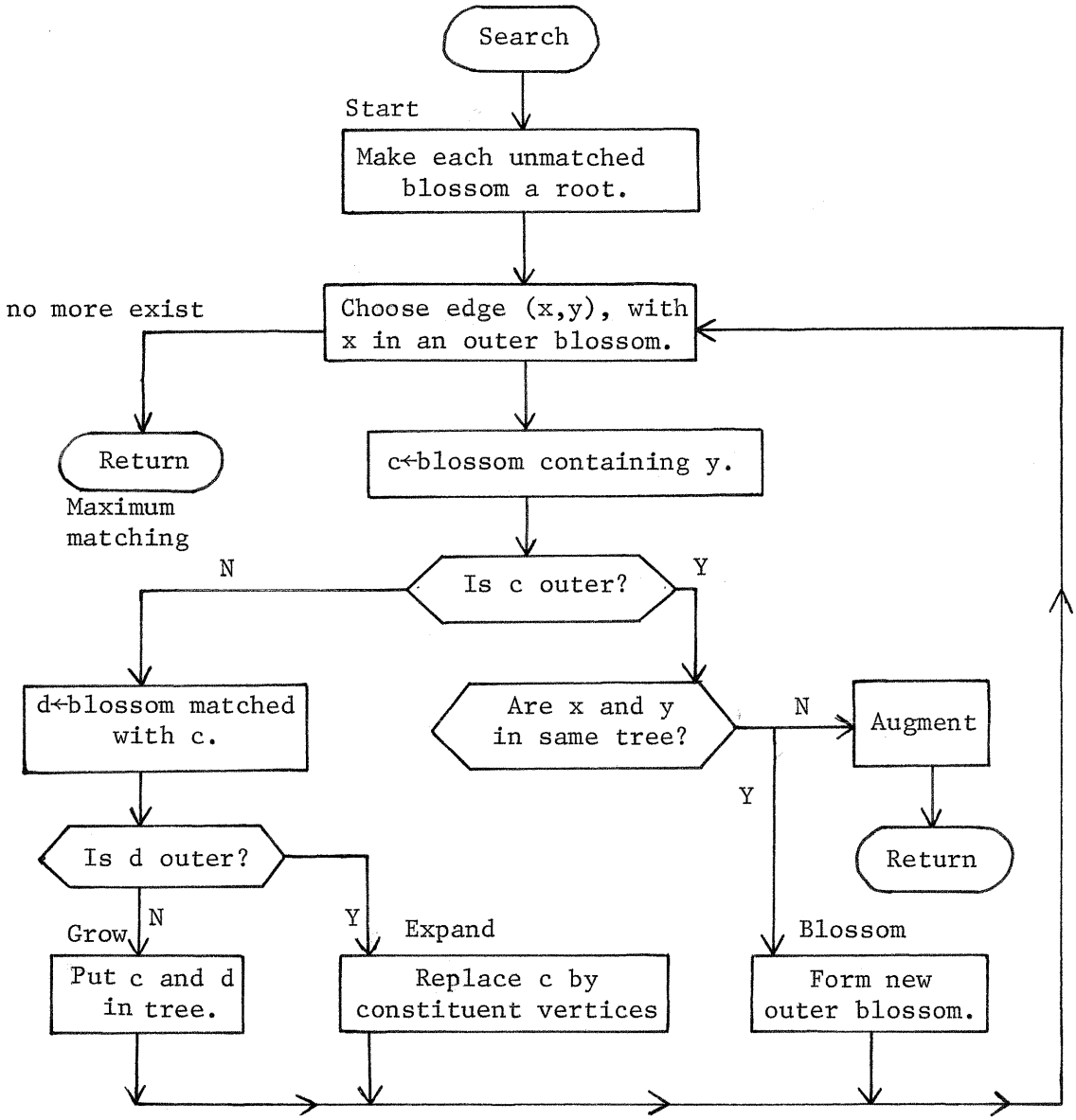


Figure 2. Cardinality matching flowchart

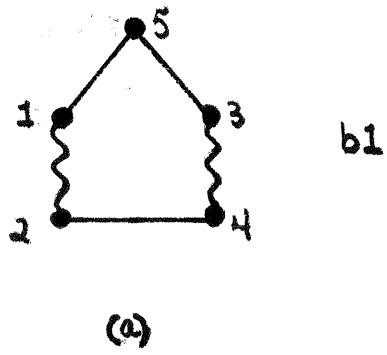


Figure 3. Blossom Formation.

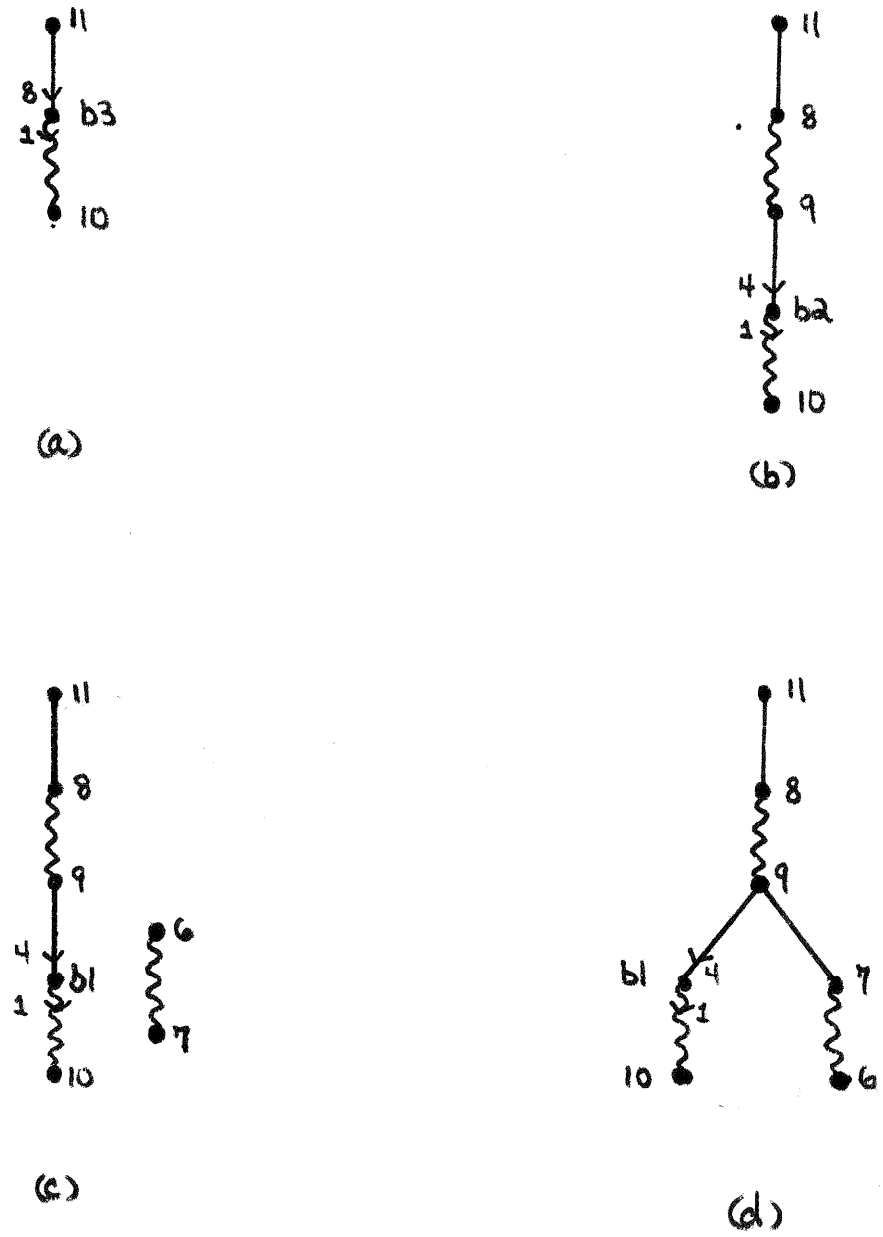


Figure 4. Search S6.

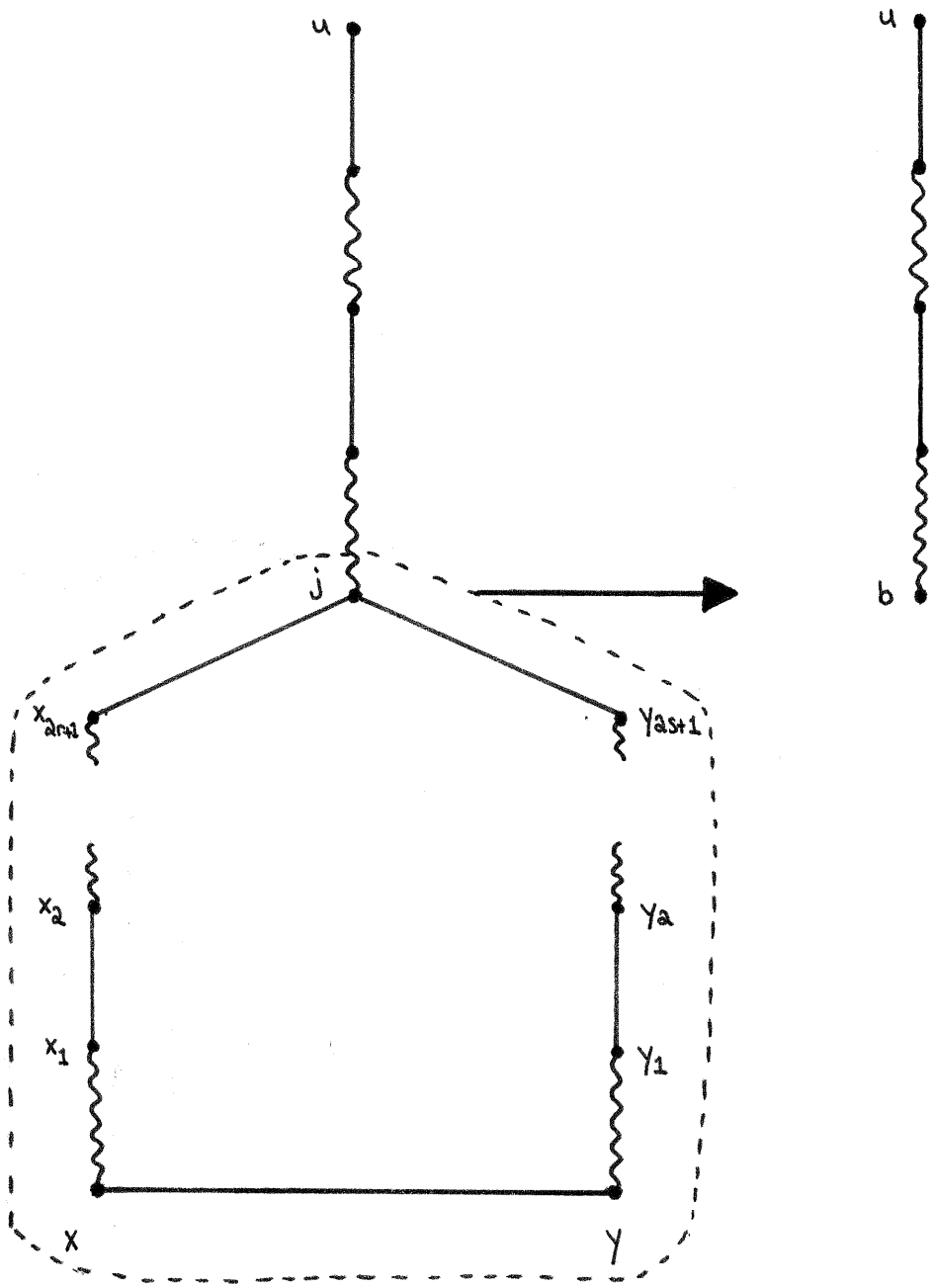


Figure 5. Blossom Step.

Search	Edge (x,y)	Edge (c,d)	Step	Action	δ	Vertex Weight	Blossom Weight
S1			Start	i is outer, $i \leq i \leq 12$		s(i)=7, $1 \leq i \leq 12$	
	(1,2)		Augment	Match (1,2)			
S2			Start	i is outer, $3 \leq i \leq 12$			
	(3,4)		Augment	Match (3,4)			
S3			Start	i is outer, $5 \leq i \leq 12$			
	(5,1)		Grow	Put 1,2 in tree			
	(5,3)		Grow	Put 3,4 in tree			
	(2,4)		Blossom	b1 replaces 1,2,3,4,5.			
	(8,9)		Augment	Match (8,9)			
S4			Start	i is outer, i=b1,6,7,10,11,12			
	(4,6)	(b1,6)	Augment	Match (b1,6)	1	s(i)=6,all $i \neq 8,9$	s(b1)=2
S5			Start	i is outer, i=7,10,11,12			
	(7,4)	(7,b1)	Grow	Put b1,6 in tree	1	s(i)=5,i=7,10,11,12	
	(7,6)		Blossom	b2 replaces b1,6,7			
	(4,9)	(b2,9)	Grow	Put 9,8 in tree	1	s(i)=4, i=7,10,11,12 s(i)=5, $1 \leq i \leq 6$	s(b2)=2
	(5,8)	(b2,8)	Blossom	b3 replaces 9,8,b2			
	(1,10)	(b3,10)	Augment	Match (10,b3)	1	s(i)=3, i=7,10,11,12 s(i)=4, $1 \leq i \leq 6$ s(i)=6, i=8,9	s(b3)=2

Search	Edge (x,y)	Edge (c,d)	Step	Action	S	Vertex Weight	Blossom Weight
S6			Start	i is outer, i = 11,12			
	(11,8)	(11,b3)	Grow	Put b3,10 in tree			
	(11,8)	(11,b3)	Expand	8,9,b2 replace b3 in tree	1	s(i)=2,i=10,11,12 s(i)=4,i=7 s(i)=5,1≤i≤6 s(i)=7,i=8,9	s(b3)=0
	(9,7)	(9,b2)	Expand	b1,6,7 replace b2 Match (6,7) Put b1 in tree	1	s(i)=1, i=10,11,12 s(i)=5,i=7 s(i)=6, 1≤i≤6,i=9 s(i)=8, i=8	s(b2)=0
	(9,7)		Grow	Put 7,6 in tree			
	(6,12)		Augment path	Match (6,12) (7,9), (8,11)			
Done			Expand	1,2,3,4,5 replace b1 Match (2,4), (3,5)			

Table I: Matching graph G1

v	1	2	3	4	5	6	7	8	9
MATE	(1,2)	(2,1)	(3,4)	(4,3)	(4,6)	(6,4)	0	(8,9)	(9,8)
LABEL	(2,4)	(1,5)	(4,2)	(3,5)	(6,7)	(4,7)	-	(9,4)	(8,5)
NEXT	4	1	6	3	2	0	9	5	8
LAST	1	2	3	4	3	6	6	8	9

Table II: Blossom arrays for b3

v	1	2	3	4	5	6	7	8	9	10	11	12
S	7	7	7	7	7	8	∞	∞	8	3	3	3
EDGE				(9,4)		(12,6)	(9,7)	(11,8)				
EDGEΔ				3		2	∞	∞				

Table III: Numeric weights at end of S6.

Edges	100	400	700	800	900	1000
Time (1/60 sec.)	56	146	287	319	344	360

Table IV: Time for graphs with 100 vertices

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

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

[4] 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.

[5] Gabow, H., 1972. "An efficient implementation of Edmonds' algorithm for maximum matching on graphs;" to appear; JACM.

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

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

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