STAN-CS-72-328 SEL-72-02

An Efficient Implement at ion of Edmonds' Maximum -Matching Algorithm

by

Harold Gabow

June 1972

Technical Report No. 31

This work was supported by the
National Science Foundat ion
Graduate Fellowship Program and
by the National Science Foundation
under grant GJ - 1180

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

STAN-CS-72-328 SEL-72-026

AN EFFICIENT IMPLEMENTATION OF EDMONDS' MAXIMUM MATCHING ALGORITHM

by

Harold Gabow

JUNE 1972

Technical Report No. 31

DIGITAL SYSTEMS LABORATORY

Dept. of Electrical Engineering Dept. of Computer Science

Stanford University

Stanford, California

This work was supported by the National Science Foundation Graduate Fellowship Program and by the National Science Foundation under grant GJ-1180.

An efficient implementation of Edmonds' maximum matching algorithm

by

Harold Gabow

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University

Abstract

A matching in a graph is a collection of edges, no two of which share a vertex. A maximum matching contains the greatest number of edges possible. This paper presents an efficient implementation of Edmonds' algorithm for finding maximum matchings. The computation time is proportional to V^3, where V-is the **number** of vertices; previous algorithms have computation time **proportional** to V^4. The implementation avoids Edmonds' blossom reduction by using pointers to encode the structure of alternating paths.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

The problem of finding maximum matchings on nonbipartite graphs has applications in integer programming and optimum scheduling. For example, Fujii, Kasami, and Ninomiya [1969] have devised an efficient algorithm for scheduling two processors. The slowest part of their algorithm is a subroutine for finding maximum matchings.

We present an algorithm for finding maximum matchings on graphs. If V is the number of vertices in a graph, the running time is proportional to V^3. The space required is roughly 3.5 V words in addition to the space needed for the graph and the matching.

The basic approach is a careful implementation of the ideas presented by Edmonds [1965]. His algorithm has running time proportionalto V^4 [Edmonds, 1965, and Fujii, Kasami, and Ninomiya, 1969-erratum]. We improve this by a factor of V. The speed-up is achieved by eleminating the process of blossom reduction. We use a system of pointers to store the relevant structure of alternating paths.

This approach is similar to the labelling techniques in the matching algorithms of Balinski [1967] and Witzgall and Zahn [1965]. We can implement Balinski's algorithm in time V^3 by maintaining a stack for vertex selection. However the generality which has made Edmonds' method so successful is lost in this implementation.

After summarizing some well-known ideas in Section 2, we state the algorithm in Section 3. A proof of correctness is given in the next section. Section 5 discusses time and space bounds and applications of the algorithm. The Appendix contain a listing of an ALGOL W program for the algorithm.

2. Some 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. A graph G_1 is shown in Fig. 1 (a). In this section G denotes an arbitrary graph.

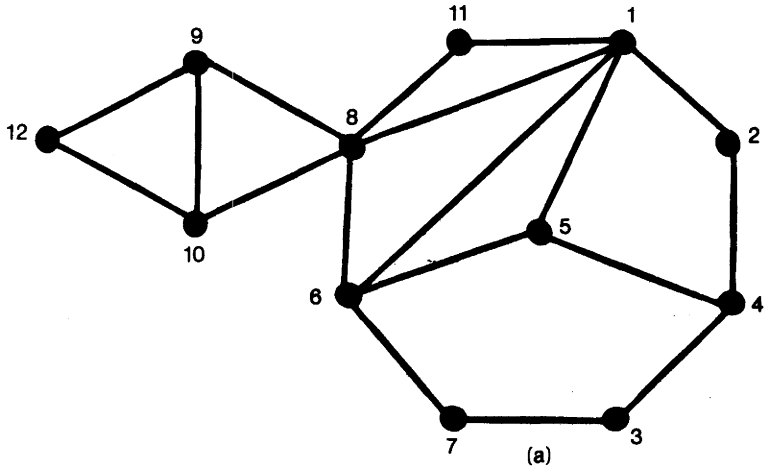
The two vertices of an edge are said to be adjacent. An adjacency list for a vertex v is an ordered list of the vertices adjacent to v. The adjacency lists in Fig. 1 (b) define the graph G_1.

A matching in G is a collection of edges, no two of which share a vertex. Figure 1 (c) shows a matching in G_1. Matched edges are drawn with wavy lines. In this section M denotes a matching. The pair (G,M) is a matched graph. M is a maximum matching in G if no matching in G contains more edges than M.

A walk [Harary, 1969] is a list of vertices $(v_1, v_2, \ldots, v_n)$ such that for $1 \le i < n$, $v_i v_{i+1}$ is an edge. A walk is simple if no vertex occurs more than once in the list. A path is a simple walk. A cycle is a walk $(v_1, v_2, \ldots, v_n)$ such that $n > 3$, $(v_1, v_2, \ldots, v_{n-1})$ is simple, and $v_n = v_1$.

Let $P = (v_1, v_2, \ldots, v_n)$ and $Q = (w_1, w_2, \ldots, w_n)$ be paths. The reverse path of P, denoted rev P, is $(v_n, v_{n-1}, \ldots, v_1)$. The concatenation of P and Q, denoted P*Q, is $(v_1, v_2, \ldots, v_n, w_1, w_2, \ldots, w_m)$. b e a path it is necessary that $v_n w_1$ be an edge and that $v_i \neq w_j$ for $1 \le i \le n$, $1 \le j \le m$.

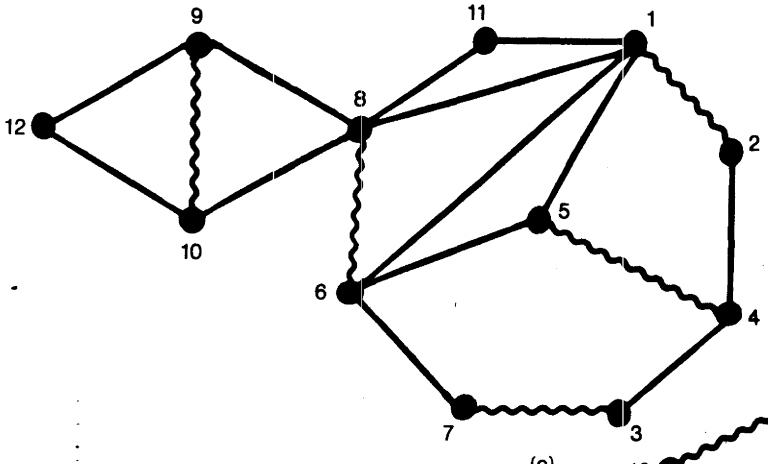
An alternating walk in a matched graph (G,M) is a walk $(v_1, v_2, \ldots, v_n)$ such that exactly one of every two edges $v_{i-1}v_i$ and $v_i v_{i+1}$, $1 < i < n$, is matched. An alternating path is a path that is an alternating walk. An exposed vertex is a vertex that is not in any edge of M. An augmenting path is an alternating path whose ends v_1 and v_n are exposed vertices.



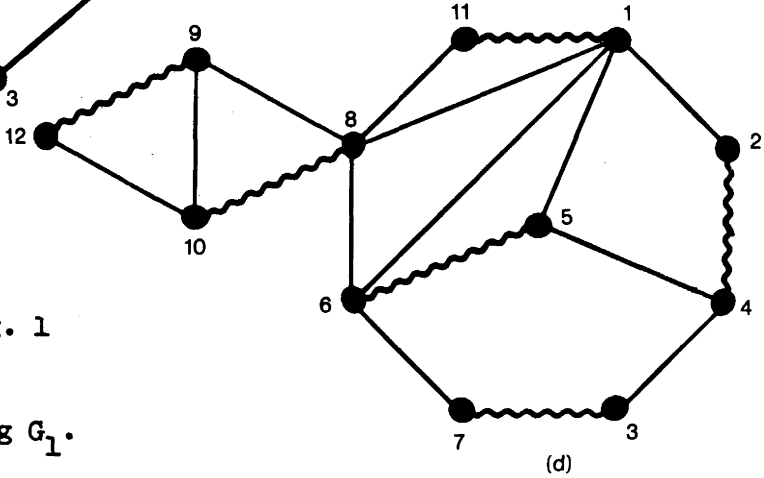
(a)

1:	(2,5,6,8,11)	7:	(3,6)
2:	(4,1)	8:	(9,10,1,6,11)
3:	(4,7)	9:	(10,12,8)
4:	(5,2,3)	10:	(9,12,8)
5:	(6,1,4)	11:	(1,8)
6:	(8,7,5,1)	12:	(9,10)

(b)



(c)



(d)

Fig. 1

(a) The graph G_1.

(b) Adjacency lists defining G_1.

(c) A matching in G_1.

(d) G_1 after augmenting along (12,9,10,8,6,5,4,2,1,11).

If $(v_1, v_2, \ldots, v_n)$ is an augmenting path in (G,M), a larger matching M' is obtained by replacing the matched edges $v_{2i}v_{2i+1}$, $1 \le i < n$, with the unmatched edges $v_{2i-1}v_{2i}$, $1 \le i \le n$. The construction of M' from M is called an augmentation. In Fig. 1 (c), (12, 9, 10, 8, 6, 5, 4, 2, 1, 11) is an augmenting path. Performing an augmentation along this path gives the matched graph with no exposed vertices shown in Fig. 1 (d).

Augmenting paths are important for the following reason.

Lemma 1: A matched graph (G,M) has an augmenting path if and only if M is not maximum.

Proof: See [Berge , 1957] or [Edmonds, 1965].

As a consequence, a maximum matching can be obtained by repeatedly searching for augmenting paths and performing augmentations. The algorithms presented in [Balinski, 1967], [Berge, 1957], [Witzgall and Zahn, 1965], and the algorithm described in the next section are organized in this manner.

3. Statement of the Algorithm

This section presents an efficient algorithm for finding maximum matchings on graphs. First the overall strategy is described. Then the data structures used by the algorithm are discussed and illustrated, and the strategy is elaborated. Next the algorithm is presented in full detail. An example of how it works on a particular graph is given. Finally an application of Edmonds' algorithm to the same graph is discussed, and the two algorithms are compared.

The algorithm is called MATCH. The input to MATCH is a collection of adjacency lists defining a graph. The output is a maximum matching for the graph, stored in an array MATE. MATE contains an entry for each

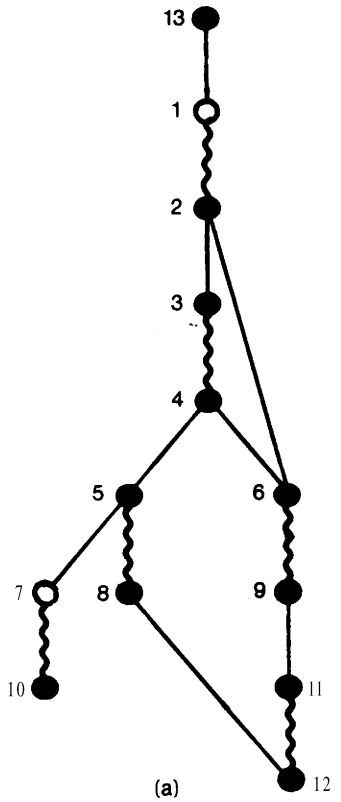
vertex. If u and v are vertices, edge uv is matched if and only if **MATE (u) =** v and **MATE** (v) = **u.**

MATCH begins with the empty matching, that is, all vertices are exposed. It searches for an augmenting path. If such a path is found, the matching is augmented. The new matching contains 1 more edge than the previous one. Next, **MATCH** searches for an augmenting path for the new matching. This process is iterated until no augmenting path is found. At this point MATCH halts with a maximum matching.

MATCH searches for an augmenting path in the following way. First an exposed vertex e is chosen. MATCH scans edges to find alternating paths to e. A vertex **v** is said to be linked when MATCH finds an alternating path that starts with a matched edge and goes **from v** to e. Let such a path be **P(v,e)** = (v, v_1,...,**e),** so vv_1 is a matched edge. MATCH sets an entry in an array LINK for every linked vertex v. The path **P(v,e)** can be computed from **LINK** (v). If an edge joining a linked vertex v to an exposed vertex f ≠ e is ever scanned, MATCH finds an augmenting path (f) * P(v,e). If no such edge exists and no more vertices can be linked, there is no augmenting path.

Figure 2 illustrates the results of such a search. A matched graph is shown in Fig. 2(a). Vertex 13 is exposed. Figure 2 (b) shows the values MATCH stores when it searches for an augmenting path to 13. Figures 2(c)-(e) show several paths **P(v,e)** defined by these values. The following paragraphs explain how LINK and the associated arrays define these paths.

The LINK entry for a linked vertex is interpreted in one of three ways, depending on the link type. The three link types are degenerate, pointer, and pair. The table in Fig. 2(b) indicates 11 vertices are

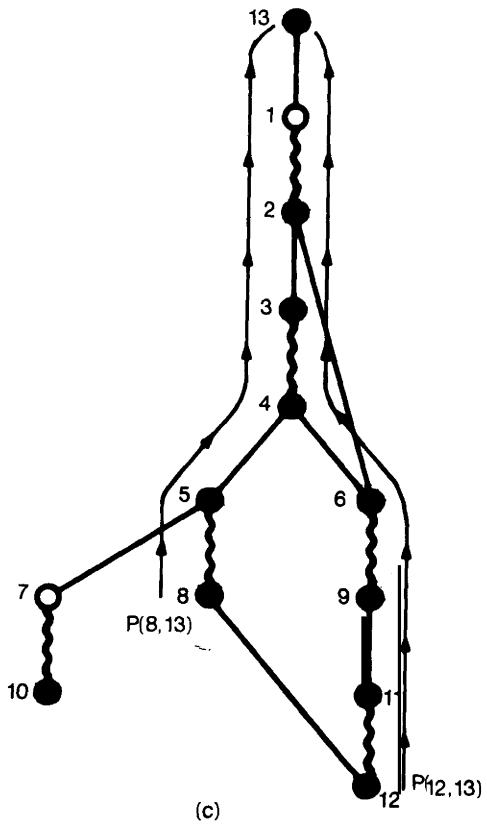


(a)

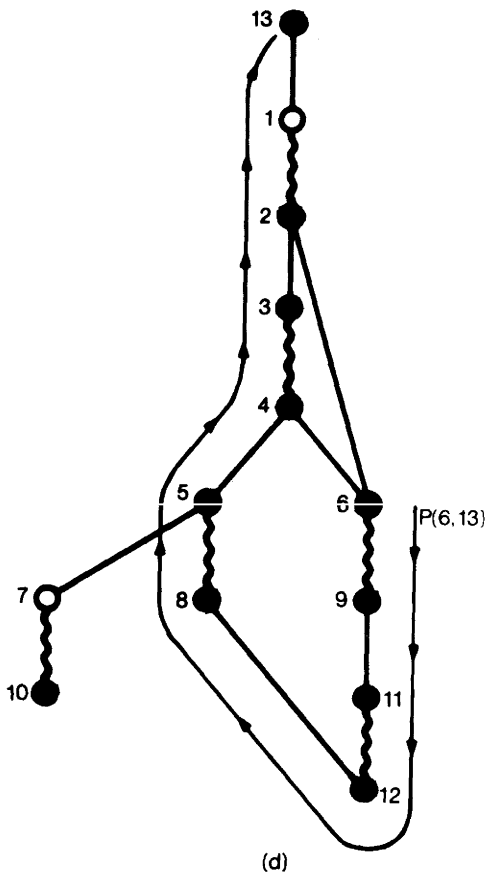
vertex	mate	link type	link
1	2	unlinked	-
2	1	pointer	13
3	4	pair	2
4	3	pointer	2
5	8	pair	1
6	9	pair	1
7	10	unlinked	-
8	5	pointer	4
9	6	pointer	4
10	7	pointer	5
11	12	pair	1
12	11	pointer	9
13	-	degenerate	-

pair link	base1	base2	top
1	8	12	1
2	2	6	1

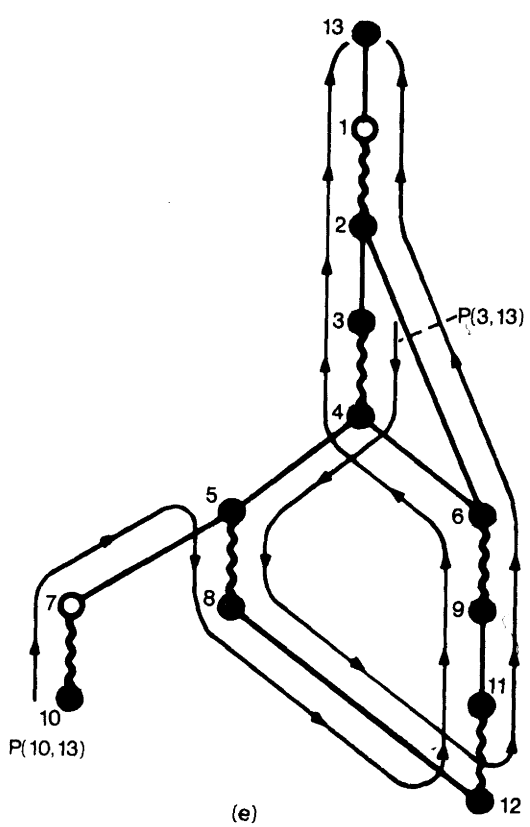
(b)



(c)



(d)



(e)

Fig. 2

A search from vertex 13

(a) A matched graph.

(b) Values stored during search.

Some paths defined by these values :

(c) P(8,13), P(12,13) .

(d) P(6,13).

(e) P(10,13), P(3,13).

linked in one of these ways. The remaining 2 vertices, vertex 1 and vertex 7, are unlinked. This means there is no alternating path starting with a matched edge that goes from 1 or 7 to 13. Note that in Fig. 2(c)-(e), the unlinked vertices are drawn hollow. This convention is used in this paper in all illustrations of matched graphs with links.

Now we describe the three link types.

Degenerate - In the search for an augmenting path to an exposed vertex e, MATCH assigns a degenerate link to e. This defines a degenerate alternating path, P(e,e) = (e). Note that if e is adjacent to an exposed vertex f, (f) * P(e,e) is an augmenting path.

Figure 2(b) -indicates that vertex 13, and no other vertex, has a degenerate link.

Pointer - If vertex v has a pointer link, LINK (v) is the number of another linked vertex. So a path P(LINK(v),e) is defined. The path P(v,e) is defined as (v, MATE (v))* P(LINK (v),e).

Using this definition and the values given in Fig. 2(b), we compute P(8, 13):

P(8,13) = (8,MATE (8)) * P(LINK (8),13) = (8,5) * P(4,13).

P(4,13) = (4,MATE (4)) * P(LINK(4), 13) = (4,3) * P(2,13).

P(2,13) = (2,MATE (2)) * P(LINK (2),13) = (2,1) * P(13,13).

= (2,1,13).

P(8,13) = (8,5,4,3,2,1,13).

Note vertices 8,4 and 2 all have pointer links, so the computation is valid. The path P(8,13) is illustrated in Fig. 2(c). Also shown is P(12,13), which is defined in a similar way by pointer links.

Pair - For vertex v to have a pair link, MATE (v) must have a pointer link. This is illustrated by the values given in Fig. 2(b).

If vertex v has a pair link, LINK(v) is an index into the parallel arrays BASE1 and BASE2. The pair of-values BASE1 (LINK (v)), BASE2 (LINK (v)) specifies vertices that define P(v,e).

As an example, consider vertex 6. The path P(6,13) is shown in Fig. 2 (d). Note that (BASE1 (LINK(~)), BASE2 (LINK (6)) = (8,12). This pair defines P(6,13) as follows: Vertices 8 and 12 are both linked. Hence there are alternating paths P(8,13) and P(12,13) (see Fig. 2(c)). Vertex 6 is in P(12,13). Let P(12,6) denote the portion of P(12,13) from 12 to 6. Thus P(12,13) = (12,11,9,6). Then P(6,13) is defined as the path rev P(12,6) * P(8,13). We can compute P(6,13) as follows:

$$P(6,13) = \text{rev}\,(12,11,9,6) * P(8,13)$$
$$= (6,9,11,12) * (8,5,4,3,2,1,13)$$
$$= (6,9,11,12,8,5,4,3,2,1,13).$$

This is the path illustrated in Fig. 2(d).

In the same way, P(3,13) can be computed. The pair link of vertex 3 specifies the vertex pair (2,6). Since vertex 3 is in P(6,13), the path P(3,13) is defined as rev P(6,3) * P(2,13). This path is shown in Fig. 2(e). The figure also shows the path P(10,13), which can be computed using the rules for pointer and pair links.

There is one other array shown in Fig. 2(b), TOP. This array has an entry for each pair link. An entry in TOP contains the number of an unlinked vertex. MATCH uses TOP to compute the unlinked vertices in paths P(v,e). For instance, if vertex v has a pair link, then TOP (LINK (v)) is the first unlinked vertex in P(v,e). Thus in Fig. 2, the first unlinked vertex in P(3,13) is 1 = TOP (2) = TOP (LINK(3)).

It is possible that P(v,e) does not contain an unlinked vertex. In this case, if' v has a pair link, TOP (LINK (v)) is set to the dummy vertex 0.

TOP is maintained because it speeds up the computation. Using TOP, **MATCH** finds the first unlinked vertex in P(v,e) with a table look-up* Without TOP, this operation would involve computing vertices in P(v,e) until an unlinked vertex is reached. Thus TOP enables MATCH to do in constant time what might **otherwise** require time proportional to the number of vertices.

Now we can give a more detailed description of how the algorithm searches for an augmenting path. A search begins by choosing an **exposed** vertex e, for which no search has previously been made. Vertex e is given a degenerate link. All other vertices are initially unlinked. **MATCH** repeatedly scans edges that emanate from linked vertices. Let x be a linked vertex, and let xy be an edge emanating from x. When **MATCH** scans xy, it processes the edge in one of four ways, depending on vertex y:

(i) If y is an exposed vertex distinct from e, MATCH augments the matching along the path: (y) * **P(x,e)**. The LINK array is used to compute **P(x,e)**, as described above. This process is illustrated schematically in Fig. j(a)-(b). After the augmentation, MATCH starts a new search.

(ii) If" y is matched with a vertex v = MATE(y) and both vertices are unlinked, v is given a pointer link, **LINK(v)←x**. This process is illustrated schematically in Fig. j(c)-(d). After linking v, **MATCH** continues the search from e.

(iii) If y is a linked vertex, the pair link **(x,y)** is assigned to certain unlinked vertices. The process is illustrated schematically

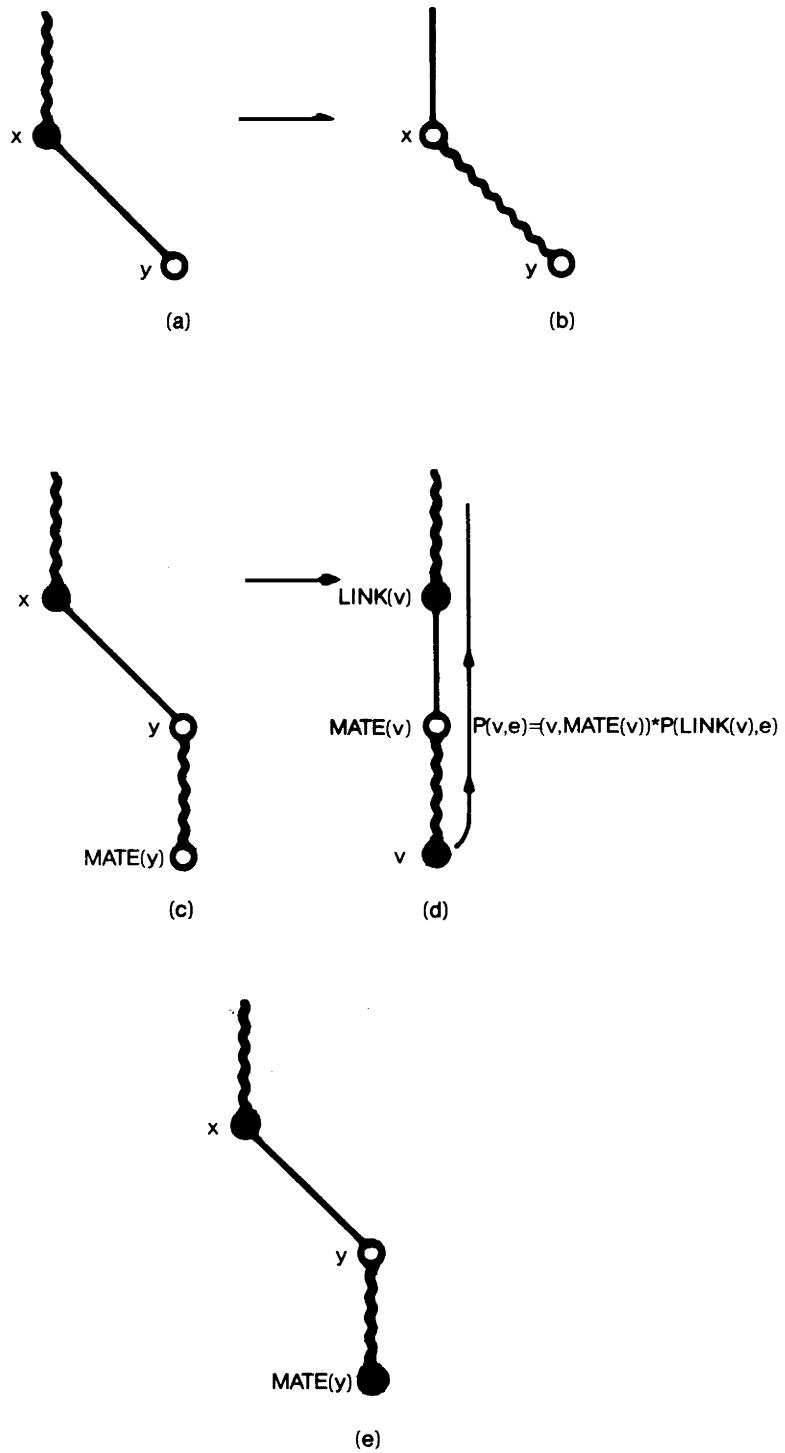


Fig. 3

MATCH scans edge xy.

(a)-(b) y exposed:augment.

(c)-(d) y, MATE(g) unlinked: assign pointer link to v = MATE(y).

(e) y unlinked, MATE(y) linked: no new links.

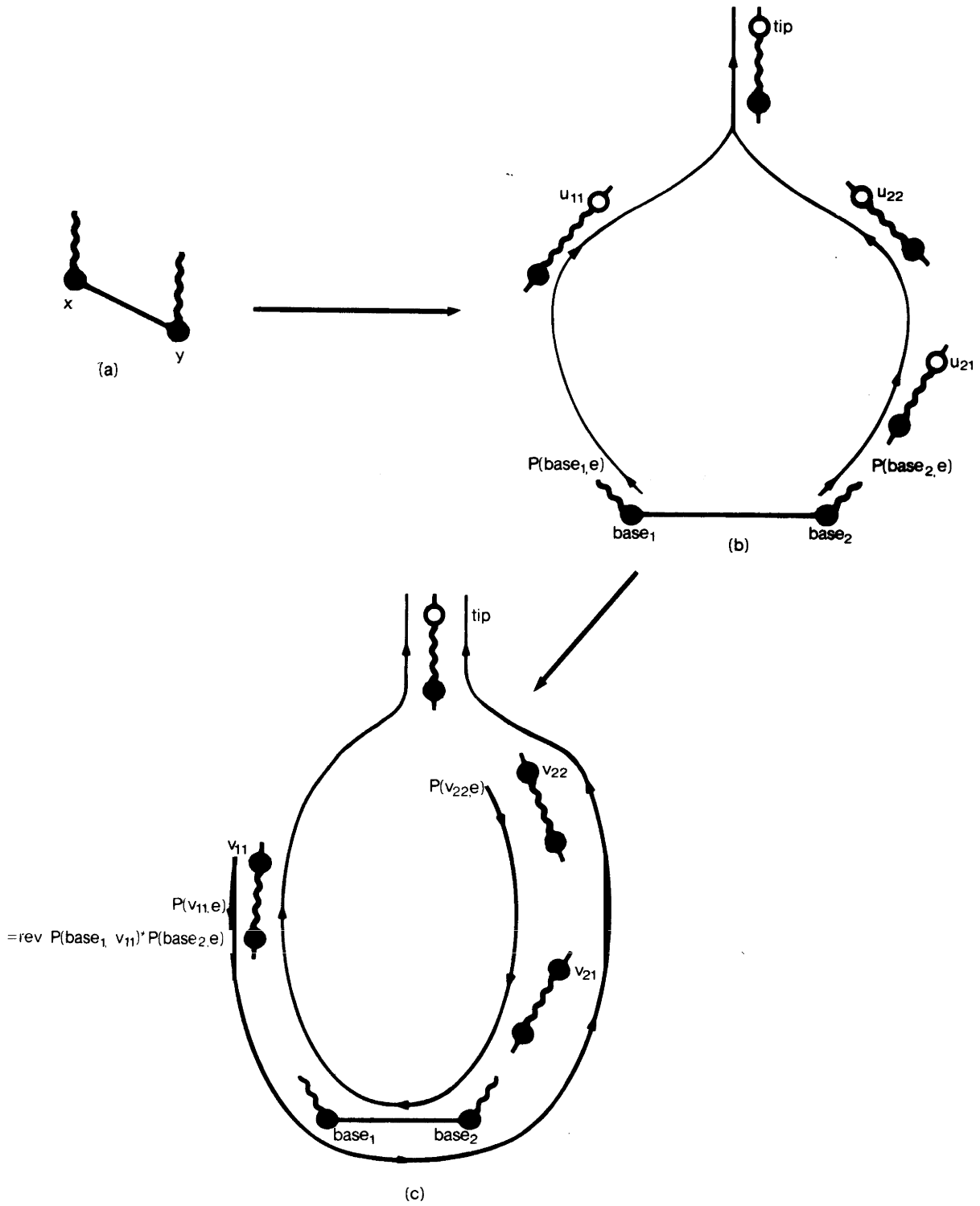


Fig. 4

MATCH scans edge xy

(a) y linked: call PAIR **LINK (y,x).**

(b) u_1 **and** u_2 step through unlinked vertices to find tip.

(c) v steps through unlinked vertices preceding **tip**, assigning pair links.

in Fig. 4(a)-(c) for (x,y) = (base1, base2). First a vertex tip is computed (Fig. 4(b)). Tip is the first unlinked vertex that is in both P(base$_1$, e) and P(base$_2$,e). TOP is used to compute tip efficiently. Next the link (base$_1$,base$_2$) is assigned (Fig. 4(c)). It is assigned to the unlinked vertices that precede tip in P(base$_1$,e) or in P(base$_2$,e) After assigning these pair links, MATCH continues the search from e.

(iv) If y is not in any of the classifications (i) - (iii), MATCH takes no further action for edge xy. (see Fig. 3(e)). The search from e is continued.

The search from e ends either when MATCH augments the matching or when MATCH runs out of edges to scan. In the former case, e is matched with a vertex during the augmentation; in the final matching e will be matched, although not necessarily with the same mate. In the latter case, e is exposed when the search ends; in the final matching e will still be exposed.

Now we present MATCH in full detail. First specifications for the data and the storage areas are given. Then the algorithm is stated.

The vertices of the input graph are numbered from 1 to V. MATCH also uses a dummy vertex 0 for boundary conditions.

The graph is stored as a collection of adjacency lists. (An adjacency matrix could be used instead, with no loss of speed). The order of the vertices in the adjacency list of v gives the order in which the edges emanating from v are scanned.

The output of the algorithm is in MATE. MATE specifies a matching this way: If u, v $\neq$ 0 are vertices, MATE (u) = 0 if and only if u is exposed; edge uv is matched if and only if MATE (u) = v and MATE (v) = u.

Intermediate matchings developed by the algorithm are stored in MATE in the same way.

There are two bits for each vertex specifying the link type. One bit specifies whether or not a vertex is linked. If it is linked, the second bit indicates the link type, pointer or pair. (The degenerate link type need not be specified.) In the statement of the algorithm below, these bits are referenced implicitly in tests such as, "If the vertex is linked, then. ..". (For example; see step M 4.)

The LINK array has an entry for each vertex. If a vertex v is linked in the current search (as indicated by the linked/unlinked bit described above), LINK (v) defines P(v,e). If v is not linked in the current search, MATCH does not use LINK (v).

In the table of Fig. 2(b), pair links have one level of indirection: the linking information is stored in BASE1 and BASE2, and a LINK entry, addresses this information. This is also how the ALGOL implementation of MATCH works. In the remainder of Section 3, and in Section 4, we are less formal. Ignoring the indirection, we write LINK (v) = (b_1,b_2), instead of b_1 = BASE1 (LINK(v)), b_2 = BASE2 (LINK(v)). This is done only for convenience.

The TOP array has an entry for each vertex pair (b_1,b_2) that has been assigned as a pair link in the current search. It is easy to see there are at most $\left\lfloor \frac{V-1}{2} \right\rfloor$ entries in TOP: In any search, 1 vertex has a degenerate link. Of the remaining V-1 vertices, half may have pointer links and half may have pair links. So at most $\left\lfloor \frac{V-1}{2} \right\rfloor$ vertices have pair links. Thus there are at most $\left\lfloor \frac{V-1}{2} \right\rfloor$ distinct vertex pairs (b_1,b_2) having entries in TOP.

We adopt a convention for addressing the entries in TOP, similar to

the one used for LINK. If v has a pair link addressing the pair (b_1,b_2) we write TOP (b_1,b_2) instead of TOP (LINK(v)).

Entries in the TOP array are made and modified by the subroutine PAIR LINK. If (b_1,b_2) is a pair link, TOP (b_1b_2) has the following properties: TOP (b_1,b_2) is the first unlinked vertex in $P(b_1,e)$; it is also the first unlinked vertex in $P(b_2,e)$. If v has the pair link (b_1,b_2), $TOP(b_1,b_2)$ is the first unlinked vertex in P(v,e); it is also the first unlinked vertex in P(MATE(v),e). If TOP (b_1,b_2) is the dummy vertex 0, there is no unlinked vertex in any of these paths.

The algorithm is presented below. A 'high level" language similar to the one developed by Knuth [1968] is used.

The algorithm consists of four routines. MATCH is the main driver; it initiates and coordinates searching for augmentations. PAIR LINK assigns pair links to vertices, using FIRST FREE to find unlinked vertices. REMATCH performs augmentations by rematching edges.

MATCH constructs a maximum matching for a graph, s a search for an augmenting path to each exposed vertex. It scans edges of the graph, deciding to assign new links or to augment the matching.

M0. [Initialize.] Read the graph into an adjacency structure, numbering the vertices 1 to V. Create a dummy vertex 0. For $0 \le i \le V$ set MATE(i) ← 0; alternatively, start with an arbitrary matching in MATE. Mark 0 as unlinked, but set LINK(0)&.

M1. [Start a new search]. Choose an exposed vertex e that has not been previously examined in M1. Mark it as linked. If no such e exists, halt; MATE contains a maximum matching.

M2. [Scan a new edge.] Choose a linked vertex x and an edge emanating from it, xy. Thisvertex-edge pair must not have been

scanned previously in M2 in this search. If no such pair exists, erase all links and go to M1 (e is not on an augmenting path, so a new search is begun).

M3. [Augment the matching.] If y is exposed, set MATE (y)←x, call REMATCH (y,x), then erase all links and go to M1 (REMATCH completes the augmentation along (y) * P(x,e). See Fig. 3 (a)-(b)),

M4. [Assign pair links.] If y is linked, call PAIR LINK (y,x) and then go to M2 (PAIR LINK assigns pair link (y,x) to unlinked vertices in P(y,e) and P(x,e). See Fig. 4).

M5. [Assign a pointer link]. Set v←MATE (y). If v is unlinked, mark v as having a pointer link, set LINK (v)←x, and go to M2 (See Fig. 3(c)-(d)).

M6. [Get a new edge.] Go to M2 (y is unlinked and MATE(y) is linked, so this edge adds nothing. See Fig. 3(e)).

FIRST FREE (v) is a subroutine of PAIR LINK. The parameter v is a linked vertex. FIRST FREE (v) returns the value of the first unlinked vertex in P(v,e); if none such exists it returns the dummy vertex 0.

F1. [Return MATE.7 If MATE(v) is unlinked, return MATE(v).

F2. [Return TOPJ If v has a pair link, set (b_1,b_2)←LINK(v) and return TOP(b_1,b_2).

F3. [Return TOP] (MATE(v) must have a pair link.) Set (b_1,b_2)← LINK(MATE(v)) and return TOP(b_1,b_2).

PAIR LINK ($base_1$,$base_2$) assigns the pair link ($base_1$,$base_2$) to unlinked vertices. The parameters $base_1$ and $base_2$ are linked vertices joined by an edge. PAIR LINK sets tip to the first

unlinked vertex in both $P(base_1,e)$ and $P(base_2,e)$. Then it links all unlinked vertices preceding tip in $P(base_1,e)$ and in $P(base_2,e)$. See Fig. 4(b)-(c).

PL0. [Initialize.] Set $u_i \leftarrow$ FIRST FREE($base_i$) for i=1,2. If $u_1=u_2$, return (no unlinked vertices can be linked). Otherwise flag u_i, i=1,2.

PL1. [Loop.] Do PL2 for i alternating between 1 and 2. Each time i is set to 1 remove any flag from the dummy vertex 0.

PL2. [Find vertices to link.] Set $u_i \leftarrow$ FIRST FREE(LINK(MATE(u_i))) (u_i is set to the next unlinked vertex in $P(base_i,e)$). If u_i is flagged, set tip$\leftarrow u_i$ and go to PL3. Otherwise flag u_i, reset i according to PL1, and go to PL2.

PL3. [Link vertices in $P(_{i},e)$.] (Tip is now set so all unlinked vertices between $base_i$ and tip can be assigned pair links. See Fig. 4(b).) Set v$\leftarrow$FIRST FREE($base_1$) and do PL4. Then set v$\leftarrow$FIRST FREE ($base_2$) and do PL4. Go to PL5.

PL4. [Link v.] If v$\neq$tip, mark v as having a pair link, set LINK(v)$\leftarrow$($base_1$,$base_2$), unflag v, set v$\leftarrow$FIRST FREE(LINK(MATE(v))) and go to PL4. (See Fig. 4(c).) Otherwise continue as specified in PL3.

PL5. [Set TOP] Set $u_1 \leftarrow$ TOP($base_1$,$base_2$)$\leftarrow$tip (Tip is the first unlinked vertex in $P(base_1,e)$).

PL6. [Remove flags.] Unflag u_1. Set $u_1 \leftarrow$ FIRST FREE(LINK(MATE(u_1))). If u_1 is flagged go to PL6.

PL7. [Update TOP.] For each pair link (b_1,b_2) that has been assigned in the current search from e, if TOP (b_1,b_2) is linked set TOP $(b_1,b_2)\leftarrow$ tip. has become the first unlinked vertex in $P(base_{\_1},e)$).

PL8. [Return.: Return.

REMATCH (f,v) rematches edges along an augmenting path. The argument f is a vertex which has become exposed; v is a linked vertex which will be rematched to f. REMATCH is a recursive routine.

R1. [Match f and v.] Save w←MATE(v). Set MATE(v)←f.

R2. [Rematch a path.] If MATE(w)=v and v has a pointer link, set MATE(w)←LINK(v), call REMATCH(w,LINK(v)) recursively, and then return.

R3. [Rematch two paths.] Ii' MATE(w)=v and v has a pair link, set (b_1,b_2)←LINK(v), call REMATCH(b_1,b_2) recursively, call REMATCH(b_2,b_1) recursively, and then return.

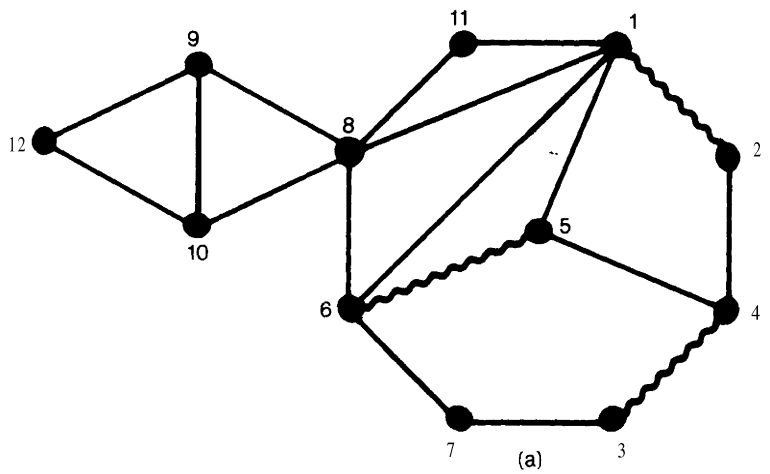
R4. [Return.] (MATE(w) ≠ v so a path has been completely rematched.) Return.

We illustrate this algorithm by showing how it works on the graph G_1 of Fig. 1(a). The input to MATCH is the collection of adjacency lists in Fig. 1(b). MATCH constructs the matching shown in Fig. 1(d).

Initially all vertices in G_1 are exposed. MATCH searches for an augmenting path to vertex 1. The first edge scanned, 12, forms such a path. An augmentation is done by placing 12 in the matching. MATCH sets MATE(1)←2, MATE(2)←3.

In a similar manner, edges 34 and 56 are matched. The matched graph at this point is shown in Fig. 5(a).

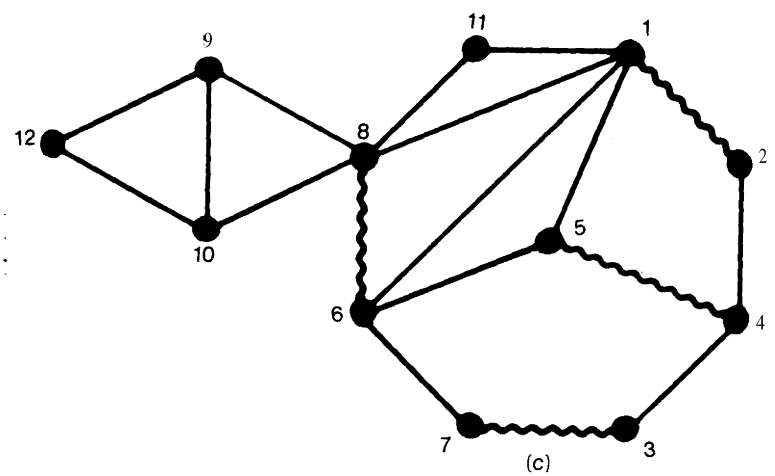
MATCH starts the next search at exposed vertex 7. The links assigned in this search are shown in Fig. 5(b). First MATCH scans edge 73 and assigns a pointer link to vertex 4. Next, MATCH chooses



(a)

vertex	link
4	7
6	4
7	dgn.

(b)



(c)

Fig. 5

(a) G_1 after 3 edges have been matched.

(b) Links assigned in search from 7.

(c) G_1 after augmenting along (8,6,5,4,3,7).

arbitrarily to scan an edge from vertex 4. This edge, 45, links vertex 6. Choosing arbitrarily again, MATCH scans edge 68. This completes an augmenting path, (8) * P(6,7). The matching which results from the augmentation is shown in Fig. 5(c).

The matching in Fig. 1(c) results when MATCH searches from vertex 9 and matches edge 9-10.

The last search is from vertex 11. Figures 6(a)-(f) show the intermediate states of the search. Each state is illustrated by a graph and tables. The graph shows the edges of G_1 that have been processed. The tables show the entries that have been made in LINK and in TOP. The graph also indicates paths P(v,11) for newly linked vertices v.

Figure 6(a) shows the state of the search after four pointer links have been assigned. When MATCH scans edge 34, pair links are assigned to vertices 5 and 7. The result is shown in Fig. 6(b).

Now we give a detailed account of how vertices 1 and 8 are linked, and Fig. 6(c) is obtained. MATCH scans edge 24. Since vertices 2 and 4 are linked, PAIR LINK (4,2) is called to assign the link (4,2).

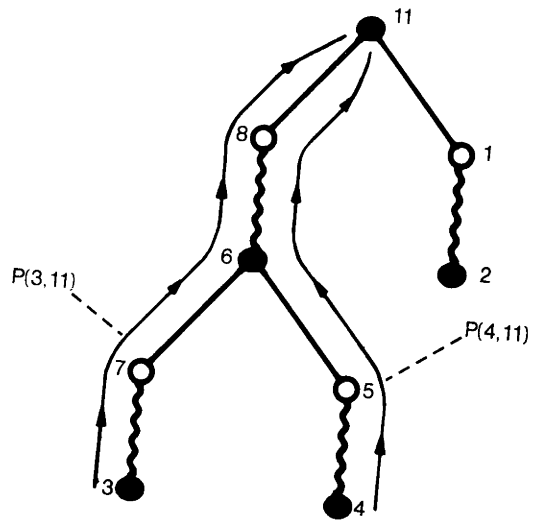
PAIR LINK first computes tip in steps PL0-PL2. Tip is found to be 0, as follows:

1. In step PL0, the first unlinked vertex in P(4,11) is computed to be vertex 8. This computation is done by the invocation FIRST FREE (4). Vertex 8 is flagged.

2. Similarly vertex 1, the first unlinked vertex in P(2,11), is computed and flagged in step PL0.

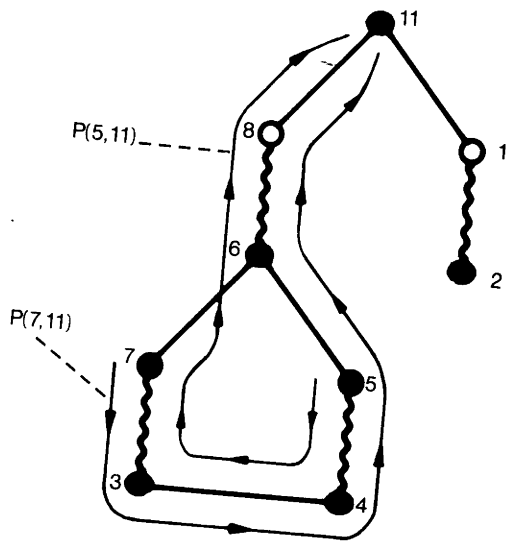
3. In step PL2, the next unlinked vertex in P(4,11) is computed to be 0. Vertex 0 is flagged.

4. In step PL2, the next unlinked vertex in P(2,11) is computed



(a)

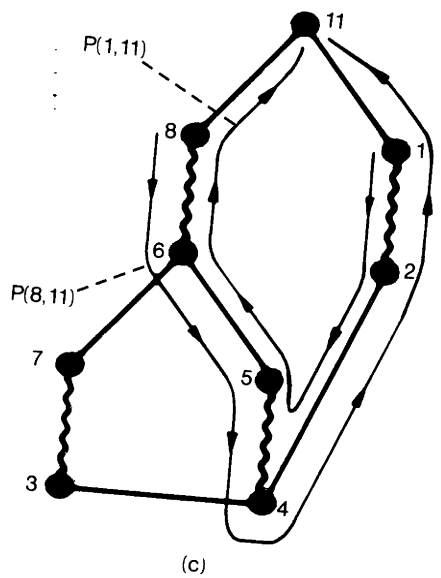
vertex	link
2	11
3	6
4	6
6	11
11	dgn.



(b)

vertex	link
2	11
3	6
4	6
5	(4,3)
6	11
7	(4,3)
11	dgn.

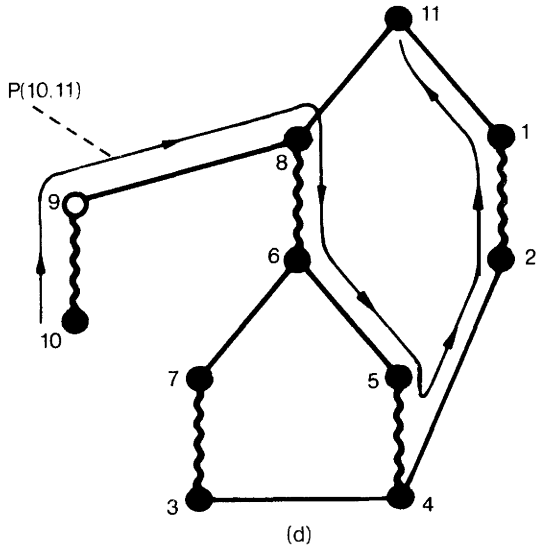
pair link	top
(4,3)	8



(c)

vertex	link
1	(4,2)
2	11
3	6
4	6
5	(4,3)
6	11
7	(4,3)
8	(4,2)
11	dgn.

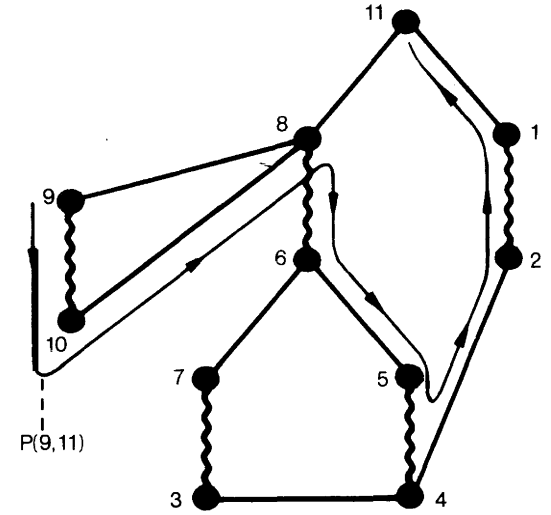
pair link	top
(4,3)	0
(4,2)	0



(d)

vertex	link
1	(4,2)
2	11
3	6
4	6
5	(4,3)
6	11
7	(4,3)
8	(4,2)
10	8
11	dgn.

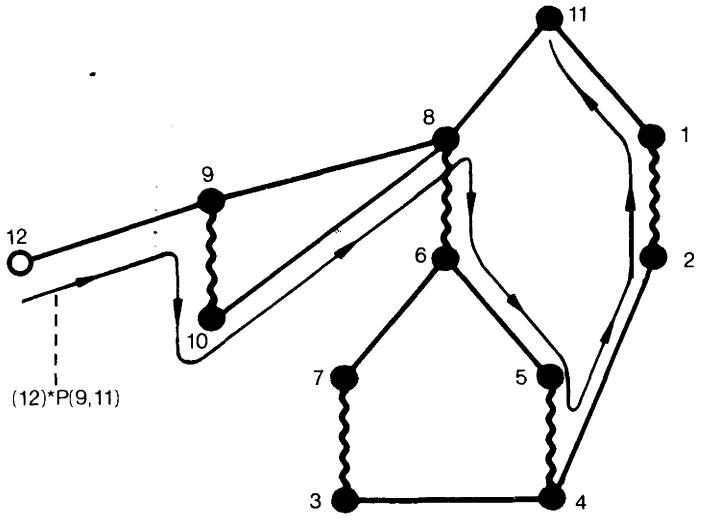
pair link	top
(4,3)	0
(4,2)	0



(e)

vertex	link
1	(4,2)
2	11
3	6
4	6
5	(4,3)
6	11
7	(4,3)
8	(4,2)
9	(10,8)
10	8
11	dgn.

pair link	top
(4,3)	0
(4,2)	0
(10,8)	0



(f)

Fig. 6
The search from vertex 11.
(a) Vertices 2,6,3,4 get pointer links.
(b) Edge 34 links vertices 5,7.
(c) Edge 24 links vertices 1,8.
(d) Vertex 10 gets a pointer link.
(e) Edge 8-10 links vertex 9.
(f) Edge 9-12 completes augmenting path (12) * P(9,11).

to be 0. Since 0 is already flagged, tip is set to 0.

In steps PL3-PL4, PAIR LINK assigns the link (4,2) to vertices 1 and 8. The flags on these vertices are also removed. The value tip = 0 is used in this process.

In steps PL5-PL6, PAIR LINK removes the flag remaining on tip = 0. Now all flags have been removed.

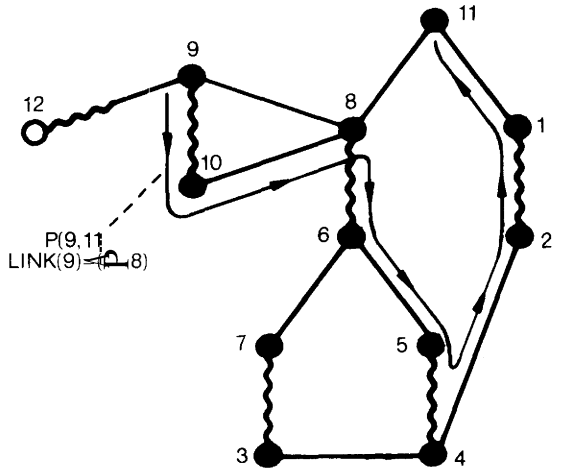
PAIR LINK sets TOP(4,2) to 0 in step PL5. This indicates there are no unlinked vertices in P(4,11) or P(2,11).

PAIR LINK resets TOP(4,3) in step PL7. Vertex 6, the previous value of TOP (4,3), is now linked. Since there are no longer any unlinked vertices in P(3,11) -or P(4,11), TOP(4,3) is reset to 0.

Finally PAIR LINK returns, in step PL8. Now MATCH continues scanning edges. Figures 6(d) and 6(e) show how vertices 10 and 9 are linked. Figure 6(f) shows how MATCH finds the augmenting path (12) * P(9,11) = (12,9,10, 8,6,5,4,2,1,11).

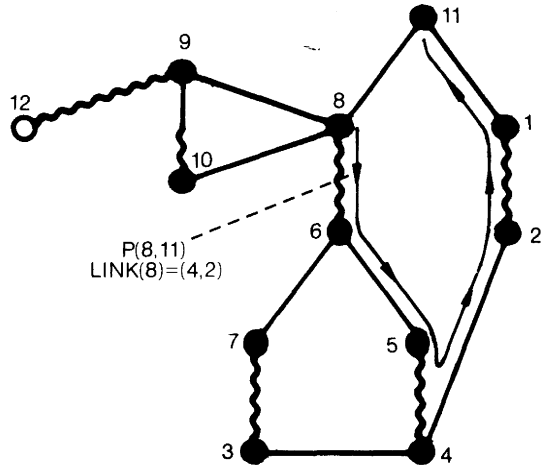
Subroutine REMATCH performs the augmentation. Figures 7 (a)-(h) show the intermediate states of the augmentation, Each state is illustrated by a graph and a stack. The stack is the stack of recursive calls to REMATCH. The graph shows a setting of MATE. As usual, vertices u and v are joined by a wavy line if and only if MATE(u) = v and MATE(v) = u. Half-wavy lines also appear in the graphs, such as edge 68 in Fig. 7(e). If uv is an edge that is wavy at u and straight at v, then MATE(u) = v but MATE(v) $\neq$ u. Thus in Fig. 7(e), MATE(6) = 8, MATE(8) = 10.

Figure 7(a) shows the matching when MATCH calls subroutine REMATCH. In step M3, MATCH sets MATE(12) to 9, as indicated by the half-wavy line between 12 and 9. Then REMATCH(12,9) is called, as shown in the stack. The path P(9,11) is shown in this-figure to clarify the operation of REMATCH(12,9)



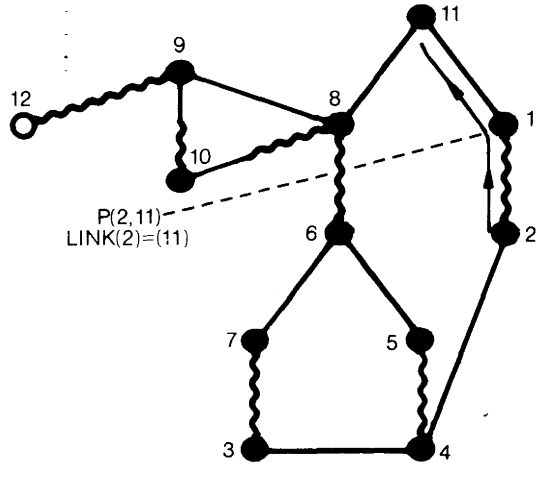
(a)

REMATCH (12,9)



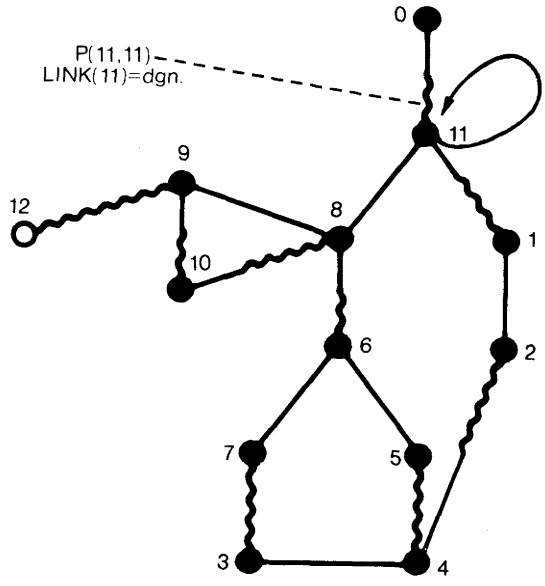
(b)

REMATCH (10,8)
REMATCH (8,10)



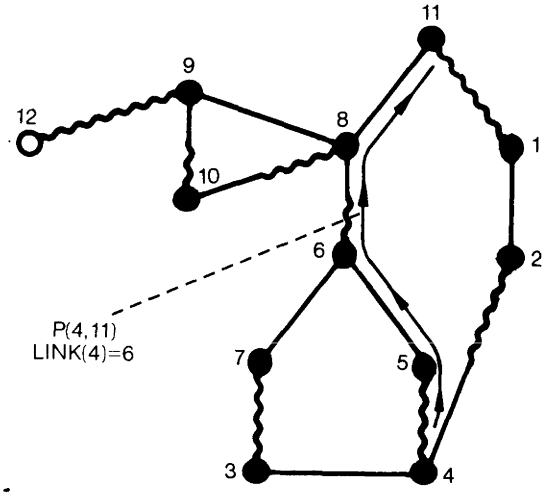
(c)

REMATCH (4,2)
REMATCH (2,4)
REMATCH (8,10)



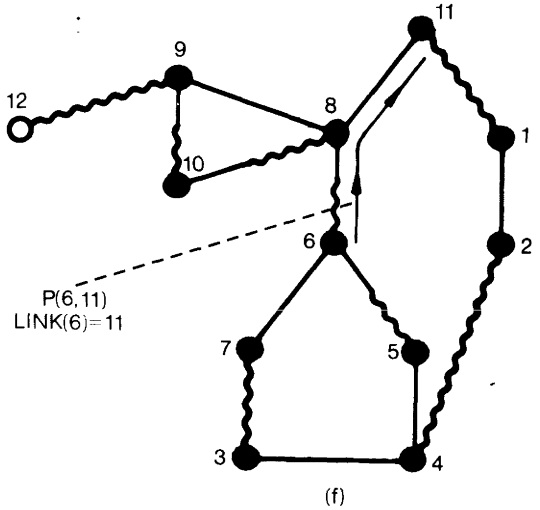
(d)

REMATCH (1,11)
REMATCH (2,4)
REMATCH (8,10)



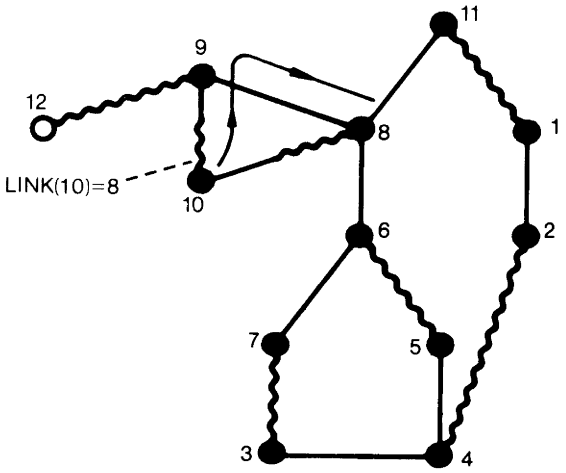
(e)

REMATCH (2,4)
REMATCH (8,10)



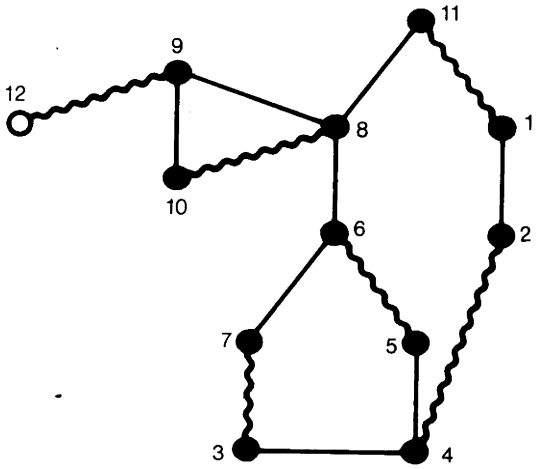
(f)

REMATCH (5,6)
REMATCH (8,10)



(g)

REMATCH(8,10)



(h)

Fig. 7

REMATCH augments along (12) * P(9,11)

(a)-(g) The invocation of REMATCH at the top of the stack is being entered. The setting of MATE is shown in the graph.

(h) The augmented matching.

Figure 7(b) shows the results of' REMATCH(12,9). Vertex 9 is completely matched with vertex 12. Also two recursive calls are in the stack. Note that P(9,11) is defined as the concatenation of two paths, rev P(10,9) and P(8,11). The two calls on REMATCH process P(9,11) by processing these two paths.

The invocation REMATCH(10,8) processes P(8,11) in a similar manner, since vertex 8 has a pair link. The results are shown in Figure 7(c).

Figure 7(d) shows the results of REMATCH(4,2). Vertices 2 and 1 have new mates. A new recursive call is in the stack. Note that P(2,11) is defined as the concatenation of (2,1) and P(11,11). The recursive call REMATCH(1,11) completes the processing of P(2,11) by processing P(11,11).

Figures 7(e)-(g) illustrate the other invocations of REMATCH, REMATCH finally returns with the matching shown in Fig. 7(h).

At this point there are no exposed vertices. MATCH halts in step M1, having constructed a maximum matching. Note this matching is identical to the matching in Fig. 1(d).

For comparison we briefly describe how Edmonds' algorithm finds the same matching in G_1. The algorithm develops the matching shown in Fig. 1(c) in a manner similar to MATCH. We discuss the search for an augmenting path to vertex 11. This search is illustrated in Fig. 8. The six graphs in Fig. 8(a)-(f) correspond to those in Fig. 6(a)-(f) for MATCH.

Edmonds conducts a search by growing a planted tree. Such a tree has an exposed vertex for a root. Its edges are alternately unmatched and matched. The planted tree in Fig. 8(a) is grown. It is easy to see the structure of planted trees corresponds to that of pointer links.

When edge 34 is scanned in Fig. 8(a) it completes a cycle (6,7,3,4,5,6,).

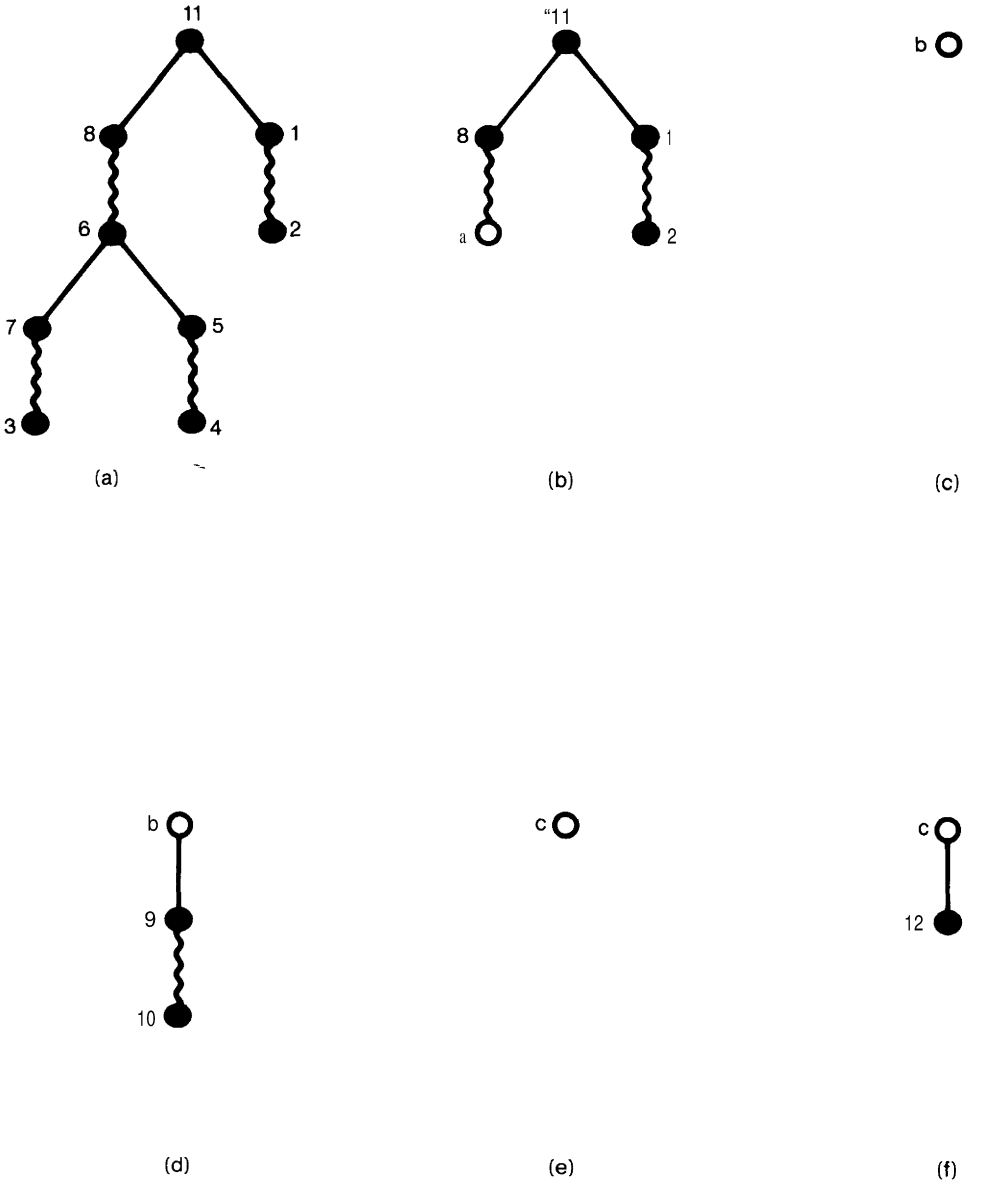
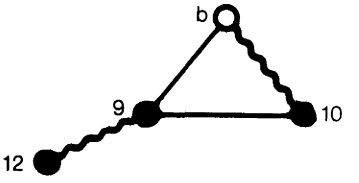


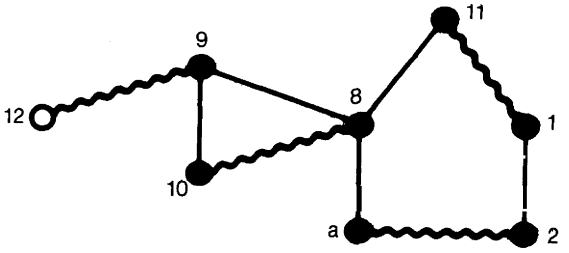
Fig. 8
The search from vertex 11 in Edmonds algorithm

(a) A planted tree. ,
(b) Blossom step for 34 yields a pseudovertex a ={6,7,3,4,5}.
(c) Blossom step for 2a yields a pseudovertex b = {11,8,a,2,1}.
(d) A planted tree in the reduced graph.
(e) Blossom step for b10 yields a pseudovertex c = {10,b,9}.
(f) Augmenting path (12 ,c) in the reduced graph.

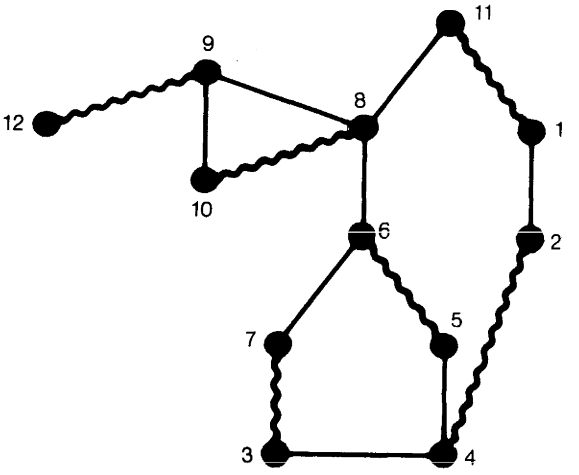
(g)



(h)



(i)



(j)

Fig. 8 (cont'd)

(g) Augmentation in reduced graph.
(h) Pseudovertex c is expanded.
(i) Pseudovertex b is expanded.
(j) Pseudovertex a is expanded.

Edmonds defines a blossom as an odd number of vertices joined by a cycle that is maximally matched. Vertices 6,7,3,4, and 5 form a blossom. The subgraph of G_1 consisting of these vertices and the edges between them are shrunk into a single vertex, a, called a pseudovertex. This results in a reduced graph G_1. The planted tree "in G_1 is shown in Fig. 8(b). The pseudovertex a is drawn hollow.

Now the problem is to find a maximum matching in the reduced graph. Suppose this has been done, as shown in Fig. 8(i). The pseudovertex a can be expanded into the original cycle (6,7,3,4,5,6,). The matching for these vertices can be chosen from the edges of the cycle, as shown in Fig. 8(j). In general, this process can be carried out because one vertex of a blossom is matched by an edge leading into the pseudovertex. The even number of vertices that remain can be matched among themselves.

The intermediate steps that construct the maximum matching in G_1' are similar. They are illustrated in Fig. 8(b)-(j). Two more blossoms are shrunk (Fig. 8(c),(e)) and then expanded (Fig. 8(h), (j)). The end result, shown in Fig. 8(j), is identical to the matching constructed by MATCH.

The shrinking and expansion operations in Edmonds' algorithm are time consuming. To construct a reduced graph for each blossom requires $O(V^2)$ steps per blossom. The result is a V^4 algorithm. MATCH avoids shrinking by recording the pertinent structure of blossoms in LINK and TOP. The factor of V speed-up results from this.

4. Proof of Correctness

We show MATCH operates in a valid and complete fashion. By valid we mean MATCH finds valid augmenting paths and correctly rematches edges along these paths. By complete we mean MATCH finds an augmenting path if one exists.

The first five lemmas establish validity and the last two lemmas establish completeness. More precisely, Lemmas 2-3 prove M4 and M5 set links so that P(v,e) is an alternating path; Lemma 6 proves M3 rematches edges along P(v,e). Lemma 7 proves each search M2-M6 is complete; Lemma 8 proves M1 initiates enough searches.

We begin by focusing on the loop M2-M4-M5-M6. This loop scans edges and assigns pointer and pair links. It terminate8 when an augmenting path is found, or when all edges have been scanned.

Lemma 2: During the loop M2-M4-M5-M6, two matched vertices v and MATE(v) are always in one of these three states:

0. v and MATE(v) are unlinked.
1. v has a pointer link and MATE(v) is unlinked.
2. v has a pointer link and MATE(v) has a pair link.

-The only possible transition from state 0 is to state 1. The only possible transition from state 1 is to state 2. Once assigned, a pointer or pair link is never changed.

These states, and the transitions between them are illustrated in Fig. j(c)-(d) and Fig. b(b)-(c).

Before proceeding, we introduce a convenient notation. Define U to be

the set of unlinked vertices in state 1. That is,

$$U = \{u \mid \text{MATE}(u) \text{ has a pointer link and } u \text{ is unlinked}\}.$$

Proof: The argument is by induction. We check that each time step M2 is reached, the classification of the Lemma holds. Also, we check that another property holds:

(1) Let x be a linked vertex. FIRST FREE(x) retumr the number of a vertex in U.

Property (1) is needed to check the classification.

Step M2 is reached after executing step M1, M4, M5, or M6. We check the two inductive assertions in each of these four cases.

Case 1: Step M1. is executed.

Step M2 is reached for the first time after M1. At this point all matched vertices are unlinked. Hence all vertices v, MATE(v) are in state 0, and the classification holds. Property (1) is vacuously true.

Case 2: Step MS is executed.

No new vertices are linked in this step. So the inductive assertions still hold when M2 is reached.

Case 3: M5 is executed.

This step assigns a pointer link to a vertex v. Both v and MATE(v) are unlinked on entry to M5. So this is a transition from state 0 to state 1.

Property (1) holds for linked vertices $x \neq v$, by induction. Property (1) also holds for vertex v: FIRST FREE(v) returns the value MATE(v) in step F1, and MATE(v) $\in$ U.

Cape 4: Step M4 is executed.

Step M4 calls PAIR LINK. In steps PL3-PL4, this subroutine links

vertices computed by FIRST FREE. So by (1), step M4 links vertices In U. These vertices make a transition from state 1 to state 2. So the classification still holds.

Now we check that property (1) holds after step M4. We consider three cases, depending on vertex x.

First suppose vertex x is in state 1. Then FIRST FREE(x) still returns the value MATE(x) ϵU.

Next, suppose FIRST FREE(x) = tip. In step PL2, tip is set to a value returned by FIRST FREE. By induction, tip ϵU. Hence FIRST FREE(x) EU.

The remaining possibility is that vertex x is in state 2 and FIRST FREE(x) $\neq$ tip. Note in this case, both x and MATE'(x) are linked vertices on entry to M4. For if x or MATE(x) is linked in PAIR LINK, FIRST FREE(x) = tip (see steps PL3-PL5,F2-F3).

Let u be the value of FIRST FREE(x) on entry to M4. By induction, uϵU on entry to M4. Below we show that after M4 is executed, FIRST FREE(x) = u and uϵU. Together these statements imply property (1) for x.

The invocation FIRST FREE(x) returns a value TOP (b_1, b_2), in step F2 or F3. So TOP$(b_1,b_2) \neq$ tip. This implies TOP (b_1,b_2) was not changed in PAIR LINK, step PL7. So the value of FIRST FREE(x) on entry to M4 is TOP(b_1,b_2). Thus u = TOP(b_1,b_2) = FIRST FREE(x).

Next note vertex u was not linked in PAIR LINK. For if u were linked, TOP(b_1,b_2) would have been changed to tip in PL7 u s uϵU after M4 is executed.

Thus property (1) holds for all linked vertices after M4.

The Lemma now follows by induction,

QED

Lemma 2 enables us to ignore such possibilities as a linked vertex being assigned a new link, or becoming unlinked. In particular, we can define a partial order $\lhd$ on the set of linked vertices, as follows:

$v \lhd w$ if and only if w is linked after v.

For example, in Fig. 4, $11 \lhd 3$, $11 \lhd 4$, $3 \lhd 5$, $3 \lhd 7$, $5 \lhd 1$, $7 \lhd 8$. For the purposes of $\lhd$, we consider vertices linked in the same invocation of PAIR LINK as being linked simultaneously. So neither $5 \lhd 7$ or $7 \lhd 5$ is true.

We also make several definitions relating to the lists (paths) **P(v,e)**. The precise rules that define these lists are given below.

0. In any search, the exposed vertex e is linked by the degenerate alternating path $\tilde{P}(e,e) = (e)$.

1. If v has a pointer link, LINK(v) contains the number of another linked vertex, and $P(v,e) = (v, MATE(v)) * P(LINK(v), e)$.

2. If v has a pair link, LINK(v) contains the numbers of two linked vertices b_1, b_2. Vertex v is in $P(b_i, e)$, for $i = 1$ or $i = 2$ (but not both). For this value of i, $P(v,e) = \underline{rev}\ P(b_i, v) * P(b_{3-i}, e)$.

These definitions are illustrated schematically in Fig. 3(d) and Fig. 4(c). In the latter, vertex v_{11} has the pair link $(\underline{base}_1, \underline{base}_2)$.

We also use a list notation, writing

$$P(v,e) = (v_0, v_1, v_2, \ldots, v_{2n}).$$

Here $v_0 = v$, $v_{2n} = e$. The last subscript is even because P(v,e) starts with a matched edge, ends with an unmatched edge, and is alternating. For convenience, define v_{2n+1} to be 0, the dummy vertex which is unlinked. This allows us to treat boundary conditions in a uniform manner.

Finally, we define a useful function:

If v is a linked vertex, free(v) is the first unlinked vertex in P(v,e).

For example, in Fig. 6(a), free(3)=7; in Fig. 6(b), free(3) =8; in Fig. 6(c), free(3) = 0. The third equality is due to the convention that 0, an unlinked vertex, is the last vertex in any path P(v,e). In general, if P(v,e) contains no "real" unlinked vertices, free(v) = 0.

In the proof of Lemma 3, we show FIRST FREE(v) computes free(v), for linked vertices v.

The first goal is to prove P(v,e) is an alternating path beginning with a matched edge. This is done in Lemma 5. We begin by showing that P(v,e) is well-defined and has several useful properties.

Lemma 3: In the loop M2-M4-M5-M6, each time step M2 is reached, the following Properties hold for every linked vertex v.

(1) P(v,e) is a well-defined list of vertices.

(2) v_{2i} is linked and v_{2i+1} = MATE(v_{2i}), for all i in $0 \leq i \leq n$.

(3) If v_{2i+1} is unlinked for some i in $0<i<n$, then P(v,e) = $P(v,v_{2i-1}) * P(v_{2i},e)$.

(4) If v has a pair link (b_1,b_2), then TOP(b_1,b_2) = free(v) = free (MATE(v)).

These properties are illustrated in Fig. 6(b) for the linked vertex v = 7. As shown, P(v,e) = P(7,11) = (7,3,4,5,6,8,11). Clearly properties (1) and (2) hold. The path decomposition of property (3) holds for i = 2, v_5 = 8, and P(v,e) = P(7,11) = P(7,5) * P(6,11) = P(v,v_4) * P(v_4,e). The setting of TOP to an unlinked vertex described in (4) holds for vertex 7 with pair link (b_1,b_2) = (4,3) and TOP (4,3)=8.

Property (3) may seem overly restrictive. It seems natural to claim

the decomposition $P(v,e) = P(v,v_{2i-1}) * P(v_{2i},e)$ holds for all i in $0<i<n$. Huwever this more general statement is false. This is illustrated in Fig. 6(c). Taking v = 8, P(v,e) =(8,6,5,4,2,1,11). For i = 1, $P(v,v_1) * P(v_2,e)$ =(8,6) * P(5,11) = ,(8 6,5 4 3 7 6,8,11) ≠ P(v,e).

Proof: The argument is by induction. We check that the Lemma is true each time step M2 is reached.

Step M2 is reached after executing step M1, M4, M5, or M6. It is easy to check the Lemma after M1, M5, and M6. This is done in Cases 1-3, below. The main part of the proof is checking the Lemma after step M4, which assigns pair links. This is done in Case 4.

Case 1: Step M1 is executed.

After M1, the only linked vertex is e. Vertex e has a degenerate link that defines P(e,e) = e. Properties (1)-(4) are easy to check:

Property (1) P(e,e) is clearly well-defined.

Property (2) For i = 0, Vertex v_0 = e is linked. Also v_1 = 0 = MATE(e).

Property (3)-(4) These properties are vacuously true.

In the remaining cases we proceed inductively. We assume that on entry to step M4, M5, or M6, Properties (1)-(4) hold for all linked vertices. We show that after the step is executed, the Properties still hold for all linked vertices.

Case 2: Step M6 is executed.

This step changes nothing. So the Properties still hold.

Case 3: M5 is executed.

Step M5 assigns a pointer link to a vertex v. We must check Properties

(1)-(4) hold after M5 for linked vertices x, $x \otimes v$, and also for v.

If $x \otimes v$, Properties (1)-(4) hold for x on entry to M5. Step M5 does nothing to modify these Properties, so they are still valid on exit.

For vertex v, the list P(v,e) is defined as (v, MATE(v))* P(LINK(v),e). Note $\mathbf{LINK(v)} \otimes v$, as illustrated in Fig. j(c)-(d). Now we verify (1)-(4) for v.

Property(1)

The list P(LINK(v),e) is well-defined, by induction. So P(v,e) is the concatenation of two well-defined lists, and hence is well-defined.

Property(2)

Property (2) holds for vertices in P(LINK(v),e), by induction. Hence Property (2) holds for v_{2i} and v_{2i+1}, $1 \le i \le n$.

For i = 0, the definition of P(v,e) shows $v_0 = v$, $v_1 = MATE(v)$.

Property(j)

Suppose $v_{2i} + 1$ is unlinked for some i in $2 \le i < n$. The following equalities show Property(j) holds in this case.

$$\begin{aligned} P(v,e) &= (v,MATE(v)) * P(LINK(v),e) && \text{def'n} \\ &= (v,v_1) * P(LINK(v), v_{2i-1}) * P(v_{2i},e) && \text{induction} \\ &= P(v,v_{2i-1}) * P(v_{2i},e) && \text{def'n} \end{aligned}$$

For i = 1, $P(v,e) = (v,v_1) * P(v_2,e)$, by definition. This is independent of whether v_3 is linked or unlinked.

Property(4)

This Property is vacuously true, since v has a pointer link.

Case 4: Step M4 is executed.

This case is the main portion of the proof. The argument is lengthy,

and divides into two parts. Part A analyzes the operation of PAIR LINK, the subroutine called in M4. The analysis depends on the inductive assumption of Properties (1)-(4). Part B uses the results of the analysis to verify that Properties (1)-(4) hold on exit from M4.

Part A: Analysis of PAIR LINK

The conclusions of this analysis form a description of how PAIR LINK and its subroutine FIRST FREE operate. The description is given below, as Properties (5)-(13). Then each of these 8 Properties is proved in turn.

Description of PAIR LINK

(5) Let $\bar{x}$ be a vertex that is linked on entry to M4. Then FIRST FREE(x) returns the value free(x).

(6) In step PL0 of PAIR LINK, u_i is initialized to the first unlinked vertex in ($base_i$,e), for i=1,2.

(7) In the loop PL1-PL2, step PL1 varies i according to the sequence i = 1,2,1,2,.... Step PI2 sets u_i to the next unlinked vertex in P($base_i$,e). If step PL2 is entered with u_i set to the dummy vertex 0, PI2 resets u_i to 0.

(8) The loop PL0-PI2 terminates when u_1 assumes a value that has been assumed by u_2, or vice versa. Tip is set to this common value.

(9) Tip is an unlinked vertex that is in P($base_1$,e) and in P(base2, e). No unlinked vertex that precedes tip in P($base_1$,e) is also in P($base_2$,e). No unlinked vertex that precedes tip in P($base_2$,e) is also in P($base_1$,e).

(10) In the loop PL3-PL4, variable v assumes the values of all unlinked vertices that precede tip in P($base_1$,e) or in P($base_2$,e)

These vertices, including tip, are assigned pair links ($base_1$,$base_2$).

(11) In the loop PL5-PL6, variable u_1 assumes the values of all the unlinked vertices that are flagged in PL0-PI2 but not linked in PL3-PL4. These vertices, including tip, are made unflagged.

(12) In step PL5, an entry for the new pair link ($base_1$,$base_2$) is added to TOP and initialized to tip. v is any vertex that receives the pair link ($base_1$,$base_2$) in PL3-PL4, then free(v) = free (MATE(v)) =TOP ($base_1$, $base_2$).

(13) In step PL7, some entries in TOP are reset to tip, so the following is true: If x has a pair link (b_1,b_2), then free(x) = free (MATE(x)) = $TOP(b_1,b_2)$.

Now we prove the Properties of the description.

Property (5)

If FIRST FREE returns in step F1, MATE(x) is unlinked. Property (2) implies P(x,e) = (x,MATE(x),...). Hence MATE(x) = free, (x). Thus FIRST FREE returns free (x).

If FIRST FREE returns in step F2, x has a pair link (b_1,b_2). Property (4) implies $TOP(b_1,b_2)$ = free(x). Thus FIRST FREE returns free (x).

If FIRST FREE returns in step F3, both x and MATE(x) are linked, and x has a pointer link. The classification of Lemma 2 implies MATE(x) has a pair link (b_1,b_2). Property (4) implies $TOP(b_1,b_2)$ = free (MATE(MATE(x))) = free (x). Thus FIRST FREE returns free (x)

QED for (5)

Property(G)

First we introduce a notational convenience: Variables u, base

stand for u_1, base1 or u_2, $base_2$.

The assignment

u ← FIRST **FREE**(base)

initializes u to free(base), by Property (5). Thus u starts out with the value of the first unlinked vertex in P (base,e). Note u is the dummy vertex 0 if there are no "real" unlinked vertices in P (base,e).

Step **PL0** returns if $u_1 = u_2$. In this case we define tip to be this common value. Note that Properties (7)-(9) are satisfied by this definition.

QED for (6)

Property (7)

It is clear that i varies between 1 and 2. We analyze the assignment in step **PL2**,

u ← FIRST **FREE**(LINK(MATE(u))),

assuming PI2 is entered with u set to an unlinked vertex in P (base,e).

First suppose u = 0. From step M0 it is clear that MATE(0) = 0, LINK(o) = 0. So **PL2** executes the assignment, u ← FIRST FREE(0). FIRST **FREE(0)** returns 0 in step **F1**. Thus **PL2** resets u to the dummy vertex 0.

Now the main case is treated, u ≠ 0 on entry to **PL2**. We show step PI2 computes the first unlinked vertex beyond u in P (base,e) and assigns this value to u.

First note that Property (3) can be applied with v = base and $v_{2i+1} = u$. Property (3) is valid for v = base, by induction. u is unlinked vertex u has an odd subscript 2_{j+1} in P(base,e), by Property (2). Since u ≠ 0, j < n. So if j > 0, Property (3) holds.

Property (3) can be written in the following way:

(14) P(base,e) = P(base,u') * P(MATE(u),e)

Here u' us defined as $(base)_{2j-1}$, the vertex that precedes u by two in P(base,e). Also MATE(u) = $(base)_{2j}$, by Property (2).

We have proved (14) for j>0. If- j = 0, u = $(base)_1$ and MATE(u) = base. Since u' = $(base)_{-1}$ is undefined, we interpret P(base,u') as the empty list. Then (14) holds for j = 0. So (14) is valid for any unlinked vertex u≠0 in P(base,e).

By Lemma 2, MATE(u) has a pointer link. The definition of pointer link implies this further decomposition:

(15) P(base,e) = P(base, u') * (MATE(u),u) * P(LINK(MATE(u)),e).

So the unlinked vertex that follows u in P(base,e) is free (LINK (MATE(u))). The assignment of PL2 computes this value, by Property (5). Thus PL2 sets u to the next unlinked vertex in P(base,e)

QED for (7)

Property (8)

We begin by proving tnis preliminary result :

(16) An unlinked vertex u occurs at most once in a list P(base,e).

The proof is by contradiction. Suppose u occurs more than once in P(base,e). First we show LINK(MATE(u)) ⊲ MATE(u). Then we use the supposition to derive a contradiction.

As noted in the proof of Property(T), MATE(u) has a pointer link. Thus, as illustrated in Fig. j(c)-(d), LINK(MATE(u)) ⊲ MATE(u).
Now consider the decomposition (15), applied to the first occurrence of u in P(base,e). The second occurrence of u is in P(LINK(MATE(u)),e). So by

Property (2), MATE(u) occurs with an even subscript in P(LINK(MATE(u)),e). Property (2) also implies that at the time LINK(MATE(u)) was assigned a link, the vertices with even subscripts in P(LINK(MATE(u)),e) were all linked vertices. Thus LINK(MATE(u))≠MATE(u). This is the desired contradiction.

QED for(16)

Now we prove Property (8). The loop PL0-PL2 terminates when u_1 assumes the value of a vertex that has already been flagged. Tip is set to this vertex. We show below that at some point, u_{3-i} took on the value tip. For convenience, we take i =1, and argue in terms of $u_i = u_1$ and $u_{3-i} = u_2$.

Case 1: Tip ≠ 0.

Tip was flagged in step PL0 or PL2 if u_1 or u_2 was assigned the value tip. If the assignment was made to u_1, then u_1 assumed the value tip twice in the loop PL0-PL2. Then Properties (6) and (7) imply tip occurs twice in P($base_1$,e). But this contradicts (16). We conclude that u_2 previously took on the value tip.

Case2: Tip = 0.

Variable u_1 may assume the value 0 more than once in loop PL0-PL2. Indeed, by Property (7), once u_1 assumes the value 0, it is always reset to 0 in PL2. However if $u_2 \neq 0$, the flag on 0 is removed before PL2 is executed again for u_1. So for tip to be 0, we must have $u_1 = u_2 = 0$.

QED for (8)

Note that Property (8) implies both u_1 and u_2 assume the value tip in PL0-PL2. Hence tip is in P($base_1$, e).

Property (9)

This Property is illustrated in Fig. 4(b). Tip is shown as the first

unlinked vertex that is common to both P(base$_1$,e) and P(base$_2$,e). As noted above, Property (8) implies tip occurs in P(base$_1$,e) and in P(base$_2$,e). We show below that if t is an unlinked vertex that precedes tip in P(base$_1$,e), t is not in P(base$_2$,e). This suffices to establish Property(g) since the argument for t in P(base$_2$,e) is similar.

First note the decomposition (14) holds for u = tip:

(17) P(base,e) = P(base, tip') * P(MATE(tip),e).

This was proved for tip ≠ 0 in the discussion of Property (7). If tip = 0, define tip'= e and take P(MATE(tip),e) = P(0,e) to be the null list. Then the decomposition holds for all values of tip.

So P(base$_2$, e) decomposes into two parts. We show that t does not belong to either-part.

Suppose t occurs in P(base$_2$,tip'). Thus u_1 and u_2 assume the value t before they assume the value tip. This cannot be, since it contradicts Property (8).

Suppose t occurs in P(MATE(tip),e). Consider the decompostion(7) for base = base$_1$. Vertex t occurs in P(base$_1$,tip'), by hypothesis, and in P(MATE(tip),e), by supposition. Thus t occurs twice in P(base$_1$,e). This cannot be, since it contradicts (16).

Thus t does not belong to P(base$_1$,e).

QED for (9)

Property (10)

In step PL3, variable v is initialized by the assignment

v← FIRST FREE (base).

This is the same as the initialization in step PL0.

In step PL4, variable v is reset by the assignment

v← FIRST FREE (LINK(MATE(v))).

This is the same as the resetting in step PL2.

So it is easy to see that v assumes the values of all unlinked vertices preceding tip in P(base,e), and these vertices are linked. This is illustrated in Fig. 4(c).

QED for (10)

Property (11)

In the loop PL0-PL2, a vertex is flagged when its number is assigned to u_1 or u_2. The loop terminates when u_i assumes the value tip, which was previously assumed by u_{3-i}. Again, take i = 1, for convenience. So the vertices that are flagged in PL0-PL2 are these: the vertices that precede tip in P($base_1$,e); the vertices that precede tip in P($base_2$,e); tip and the first k unlinked vertices following tip in P($base_2$,e), for some k. The vertices in the last set correspond to the k values assigned to u_2 after tip.

The vertices in the first two sets are made unflagged and linked in the loop PL3-PL4.

Now we show that the loop PL5-PL6 processes the vertices in the third set. Begin by considering the decomposition (17) for base = $base_2$. The decomposition shows the vertices in the third set are the first (k + 1) unlinked vertices in P(MATE(tip),e).

In step PL5, u_1 is initialized by the assignment u_1←tip. Thus u_1 is set to the first unlinked vertex in P(MATE(tip),e).

In step PL6, u_1 is reset by the assignment u_1← FIRST FREE(LINK(MATE(u_1))). Thus u_1 takes on values of consecutive unlinked vertices in P(MATE(tip),e).

So u_1 takes on the values of the vertices in the third set. These

vertices are unflagged. When u_1 assumes the value of an unflagged vertex, all (k + 1) vertices of the third set have been processed, so the loop halts.

(Note again the special case, --when 0 is the last of the (k + 1) vertices. When u_1 assumes the value 0 for the first time, the flag is removed from 0. Then in step PL6, u_1 is reset to 0. Now u_1 has no flag, so the loop terminates.)

QED for (11)

Property (12)

We begin by proving that free(v), the first unlinked vertex in P(v,e), is tip. Then we prove a similar equality for free (MATE(v)).

First note that free(base) = tip by Property (10), every vertex preceding tip in P(base,e) is linked after steps PL3-PL4.

Now consider a vertex v that has the link ($base_1$, $base_2$). For convenience, suppose v is in P($base_1$,e). Figure 4(c) illustrates this situation. By definition, P(v,e) = rev P($base_1$,v) * P($base_2$,e). e list P($base_1$,v) contains no unlinked vertices, since free($base_1$) = tip and v precedes tip. So the first unlinked vertex in P(v,e) is the the first unlinked vertex in P($base_2$,e). Thus free(v) = free ($base_2$) = tip, as claimed.

Next consider a vertex MATE(v), where v has the link ($base_1$, $base_2$). We rewrite the decomposition (14):

$$P(base_1, e = P(base_1, v') * P(MATE(v), e).$$

Vertex tip occurs after v in P($base_1$,e), whence tip occurs in P(MATE(v),e). So free($base_1$) = tip = free (MATE(v)),as claimed.

. QED for (12)

Property (13)

Suppose x has a pair link (b_1, b_2). The case (b_1,b_2)= $(\underline{base}_1,\underline{base}_2)$ is treated in Property (12). m e $x \textcircled{<} v$.

Note that on entry to PAIR LINK, free(x) = free(MATE(x)) = $TOP(b_1,b_2)$, by Property (4). Let u be this common value.

If u is not linked in PL3-PL4, then free (x) and free (MATE(x)) do not change. Also TOP (b_1,b_2) is not modified in PL7. So the three values remain equal, and Property (13) holds.

Suppose u is linked in PL3-PL4. A decomposition similar to (14) holds:

$$P(x,e) = P(x,u') * P(MATE(u),e).$$

The vertices in P(x,u') precede u, so none of them are unlinked. So the first unlinked vertex in P(x,e) is the first unlinked vertex in P(MATE(u),e). Thus free (x) = free(MATE(u))= tip, by Property (12).

The proof that free(MATE(x))= tip in this case is analogous.

QED for (13)

B. Proof of Properties (1)-(4)

Now that PAIR LINK has been analyzed, it is easy to check that Properties (1)-(4) hold for all linked vertices after step M4.

If no vertices are linked in PAIR LINK, step PL0 returns. Nothing is changed in step M4. So Properties (1)-(4) still hold after I&.

Now suppose one or more vertices are linked in PAIR LINK. Let v be such a vertex. We check Properties (1)-(4) for v and for all vertices $x \textcircled{<} v$, below.

Vertex v has the pair link $(\underline{base}_1, \underline{base}_2)$. For definiteness, choose v in $P(\underline{base}_1,e)$. Thus P(v,e) = rev $P(\underline{base}_1,v)$ * $P(\underline{base}_2,e)$.

This is illustrated by vertex v_{11} in Fig. 4(c).

Property (1)

Property (1) holds for vertices $x \leq v$ on entry to M4, by induction. Since PAIR LINK does not reset any entries in LINK or MATE, the lists P(x,e) do not change. Hence Property (1) still holds for vertices x on exit from M4.

In particular, the lists P(base$_1$,e) and P(base$_2$,e) are well-defined.. Also, P(base$_1$,v) is well-defined, since Property (10) shows v occurs in P(base$_1$, e). Thus P(v,e) = rev P(base$_1$,v) * P(base$_2$,e) is well-defined. So (1) holds for v.

QED for(1)

Property (2)

Property (2) holds for vertices $x \leq v$, since the only possible change in the list P(x,e) is that some unlinked vertices become linked.

Now we check that the vertices with even subscripts in P(v,e), v_{2i}, are linked. Writing P(v,e) = rev P(base$_1$, v) * P(base$_2$,e), we check the two portions of P(v,e) separately.

All vertices in P(base$_1$,v) are linked. This is a consequence of Property (10). So the vertices v_{2i} in rev P(base$_1$,v) are certainly linked;

Now we check the vertices v_{2i} in P(base$_2$,e). On entry to M4, the even-subscripted vertices in P(base$_1$, e) are linked, by Property (2). Thus vertex v has an odd subscript in P(base$_1$,e). So in P(v,e), base$_1$ has an odd subscript, and base$_2$ has an even subscript. Thus the vertices v_{2i} in P(base$_2$,e) are the vertices with even subscripts in P(base$_2$,e). So Property (2) for base$_2$ shows these vertices v_{2i} are linked.

It remains only to check that $v_{2i+1} = MATE(v_{2i})$. This is illustrated in Fig. 4(c). The proof follows easily from the properties just established.

Q,ED for (S)

Property (3)

Property (3) holds for vertices $x \leq v$, since the only possible change in the list P(x,e) is that some odd-subscripted vertices become linked.

Now we check Property (3) for v. Write P(v,e) = rev $\mathbf{P}(\underline{base}_1, v)$ * $\mathbf{P}(\underline{base}_2, e)$. Let v_{2i+1} be an unlinked vertex in this list. As noted above, all vertices in $P(\underline{base}_1, y)$ are linked. So v_{2i+1} has an odd subscript, 2j+1, in $P(\underline{base}_2, e)$. So for j > 0, the following equality holds:

$$
\begin{aligned}
P(v,e) &= \text{rev } \mathbf{P}(\underline{base}_1, v) * P(\underline{base}_2, e) \\
&= \text{rev } \mathbf{P}(\underline{base}_1, v) * \mathbf{P}(\underline{base}_2, v_{2i-1}) * \mathbf{P}(v_{2i}, e) && \text{Property (3)} \\
&= \mathbf{P}(v, v_{2i-1}) * \mathbf{P}(v_{2i}, e) && \text{def 'n}
\end{aligned}
$$

So Property (3) holds for v in this case.

For j = 0, the definition of P(v,e) gives Property (3).

QED for (3)

Property (4)

This Property was proved in the analysis of PAIR LINK, as Properties (12) and (13).

QED for (4)

Now the inductive hypotheses have been verified for all cases. The Lemma follows, by induction.

QED

It is easy to conclude from Lemma 3 that P(v,e) is an alternating walk beginning with a matched edge. First a simple induction shows P(v,e) is a walk. The argument is illustrated in Fig. 3(d) and Fig. 4(c). Then Property (2) of Lemma 3 shows P(v,e) is alternating, with the first edge matched.

The proof that P(v,e) is simple is more involved. It depends on another relationship between linked and unlinked vertices, proved in Lemma 4. First we give a definition extending free to a function of two variables:

If v and w are linked vertices and w ϵ P(v,e), then free (v,w) is the first unlinked vertex beyond w in P(v,e).

For example, in Fig. 2(e), free (10,6) = 1; free (10,13) = 0; free (10,10) = 7. In general, free (v,v) = free (v).

Strictly speaking, free (v,w) is not well-defined. We have not shown P(v,e) is simple, so w may occur more than once. We agree to always choose the first occurrence of w.

Lemma 4: Suppose v and w are linked vertices and w ϵ P(v,e). Then free (w) = free (v,w).

Figure 2(e) illustrates the Lemma. Taking v = 10 and w = 3, free (3) = 1 = free (10,3). This figure also disproves two modifications of the Lemma that one might conjecture. First, free (3) = 1 $\neq$ 7 = free (10), so the conjecture free (w) = free (v) is false. Second, one might hope that P (w,e) is a sub-path of P(v,e). This is not the case in Fig. 2(e).

The proof' is by induction. We show the Lemma is true each time a link is assigned.

Suppose v is assigned a pointer link, so P(v,e) = (v,MATE(v))* P(LINK(v),e). Let w be a linked vertex in P(LINK(v),e). So free(v,w) = free(LINK(v),w). By induction, free(w) = free(LINK(v),w). Using these equalities, we see the Lemma holds after a pointer link is assigned.

To check the Lemma after pair links are assigned, we consider four cases. These depend on whether v and w are linked during the current execution of PAIR LINK or were previously linked.

Case 1: v and w were previously linked.

Suppose prior to the execution of PAIR LINK, u = free(w) = free(v,w). If u is unlinked after PAIR LINK, this equality still holds. Otherwise, decomposition (15) derived in Lemma 3 holds for v end w:

$$P(v,e) = P(v,u') * P(MATE(u),e)$$

$$P(w,e) = P(w,u') * P(MATE(u),e)$$

If t is the first unlinked vertex in P(MATE(u),e), t = free(w) = free(v,w).

Case 2: v was previously linked.

Vertex w is linked by PAIR LINK, so MATE(w) was previously linked. Furthermore, MATE(w) ϵ P(v,e) by (2) of Lemma 3. So by Case 1, free(MATE(w)) = free(v,MATE(w)). Property (4) of Lemma 3 shows free(w) = free(MATE(w)). Also free(v,w) = free(v,MATE(w)), since MATE(w) and w are consecutive vertices in P(v,e). Combining equalities we get free(w) = free(v,w).

Case 3: w was previously linked.

Vertex v is linked by PAIR LINK. Let P(v,e) = rev P($base_1$,v) * P($base_2$,e).

If w ϵ P($base_1$,v), Case 1 shows free(w) = free($base_1$,w). Since free($base_1$,w) = tip = free(v,w), the desired equality holds.

If w ϵ P($base_2$, e), Case 1 shows free(w) = free($base_2$,w).

$P(\underline{base}_2,e)$ is included in $P(v,e)$, $\underline{free}(v,w) = \underline{free}(\underline{base}_2,w)$, and the desired equality holds.

Case 4: v and w were previously unlinked.

It is clear from Fig. 4(c) that $\underline{tip} = \underline{free}(w) = \underline{free}(v,w)$.

By induction the Lemma holds each time a link is assigned.

QED

Now we can complete the proof that $P(v,e)$ is en alternating path.

Lemma 5: If v is a linked vertex, $P(v,e)$ is simple.

Proof: We assert the Lemma is true each time a link is assigned.

Suppose v is assigned a pointer link, so $P(v,e) = (v,MATE(v)) * P(LINK(v),e)$. The walk $P(LINK(v),e)$ is simple, by Induction. It does not contain v or MATE(v), since both vertices were previously unlinked. Hence $P(v,e)$ is simple.

Suppose v is assigned a pair link. Let $P(v,e) = \underline{rev}\ P(\underline{base}_1,v) * P(\underline{base}_2,e)$. Both $P(\underline{base}_1,e)$ and $P(\underline{base}_2,e)$ are simple, by induction. So $P(\underline{base}_1,v)$ is also simple. It suffices to show $P(\underline{base}_1,v)$ is disjoint from $P(\underline{base}_2,e)$.

Consider the graph before the pair link $(\underline{base}_1, \underline{base}_2)$ is assigned, as illustrated in Fig. 4(b). Suppose $w \in P(\underline{base}_1,e) \cap P(\underline{base}_2,e)$. We show $w \notin P(\underline{base}_1,v)$. We can choose w to be linked, since MATE(w) is also in the intersection, and w or MATE(w) is linked: Lemma 4 implies $\underline{free}(\underline{base}_1,w) = \underline{free}(w) = \underline{free}(\underline{base}_2,w)$. Referring back to Fig. 4(b), either $\underline{free}(w)$ is $\underline{tip}$ or free(w) lies beyond tip. Since v is assigned a link (has $\underline{base}_2$), v does not lie beyond w. Equivalently, $w \notin P(\underline{base}_1,v)$.

Thus $P(\underline{base}_1,v)$ and $P(\underline{base}_2,e)$ are disjoint, and $P(v,e)$ is simple.

By induction the Lemma holds each time a link is assigned.

QED

Note our results do not show that, as one might guess from Fig. 4(b), MATE(tip) is the first vertex common to P(base$_1$,e) and P(base$_2$, e). For example, consider Fig. 7. Suppose an edge joining 5 and 12 is scanned next. PAIR LINK is called. It sets tip to vertex 1, the first unlinked vertex common to P(5,13) and P(12,13). These two paths join and diverge several times before vertex 1. MATE(1) = 2 is certainly not the first common vertex. In general, although P(base$_1$,e) and P(base$_2$,e) may join and diverge arbitrarily before joining at tip, the argument in Lemma 5 shows only linked vertices occur between the intersection and MATE(tip).

We conclude from Lemma 5 that in step M2, when MATCH scans an edge xy leading to an exposed vertex y, (y) * P(x,e) is an augmenting path. Now we analyze step M3 and REMATCH to see how the matching is augmented.

Figure 9(a) shows (y) * P(x,e) when REMATCH (y,x) is called in M3. The hollow vertices x_{2i+1} may or may not be linked. The convention for half-wavy edges, introduced in Fig. 7, is used. Thus MATE(y) = x but MATE(x) $\neq$ y.

Figure 9(b) shows (y) *P(x,e) when REMATCH (y,x) returns. The path has been rematched and the augmentation is complete.

Lemma 6 shows REMATCH accomplishes the transformation shown in Fig. 8(a)-(b). First we make some definitions. If z is a vertex, let M(z) be the value of MATE(z) when the search begins in M1. Define a set Z that grows and shrinks as REMATCH resets MATE, by

$$Z = \{M(z) \mid MATE(MATE(z)) \neq z\}.$$

A vertex in Z is at the straight end of a half-matched edge, as illustrated

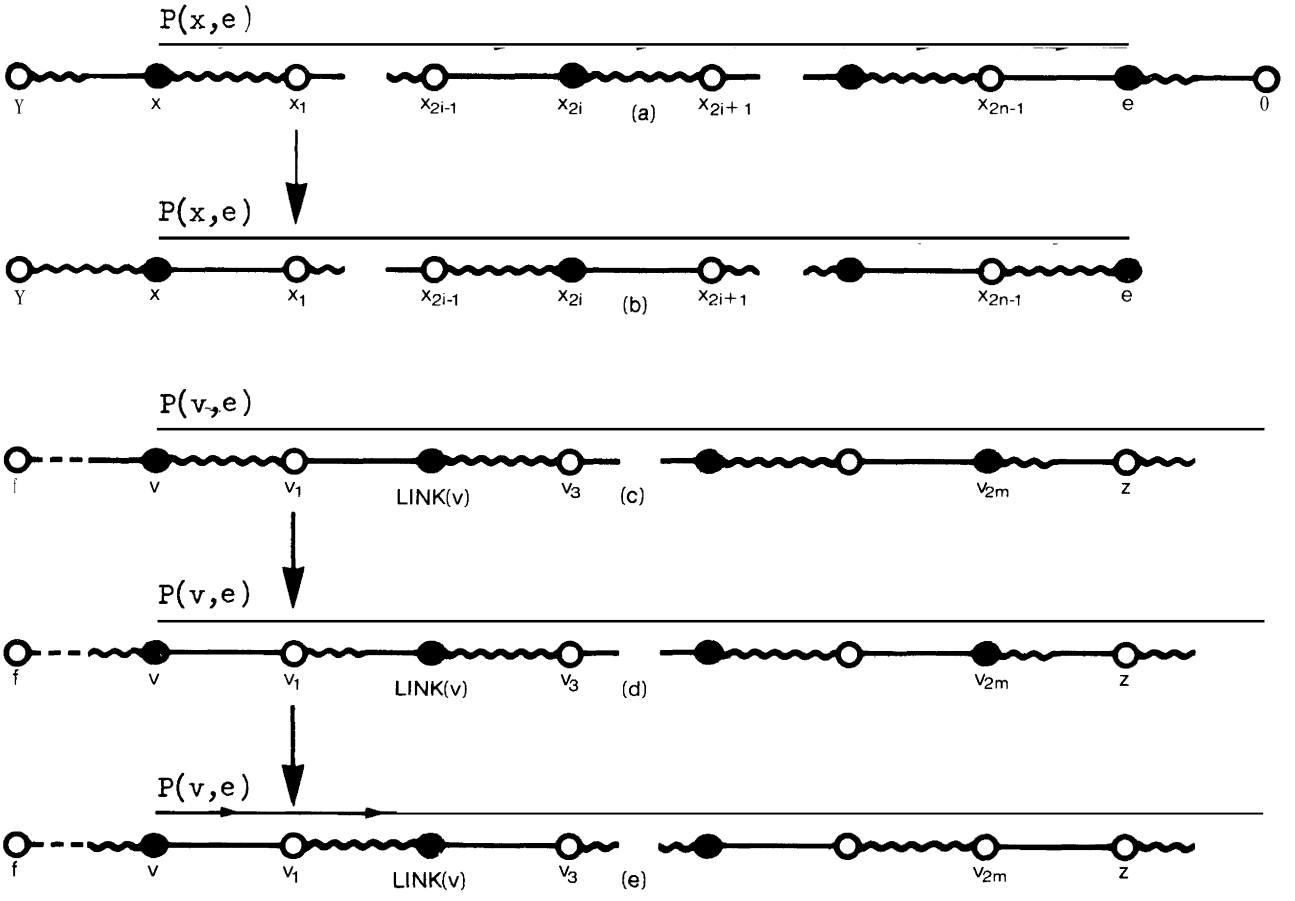


Fig. 9

Rematching an augmenting path
The augmenting path (y) * P(x,e).

(a) On entry to REMATCH (y,x).
(b) On exit.
The path (f) * P(v,z): v has a pointer link.
(c) On entry to REMATCH (f,v).
(d) On entry to REMATCH (v_1, LINK(v)).
(e) On exit.

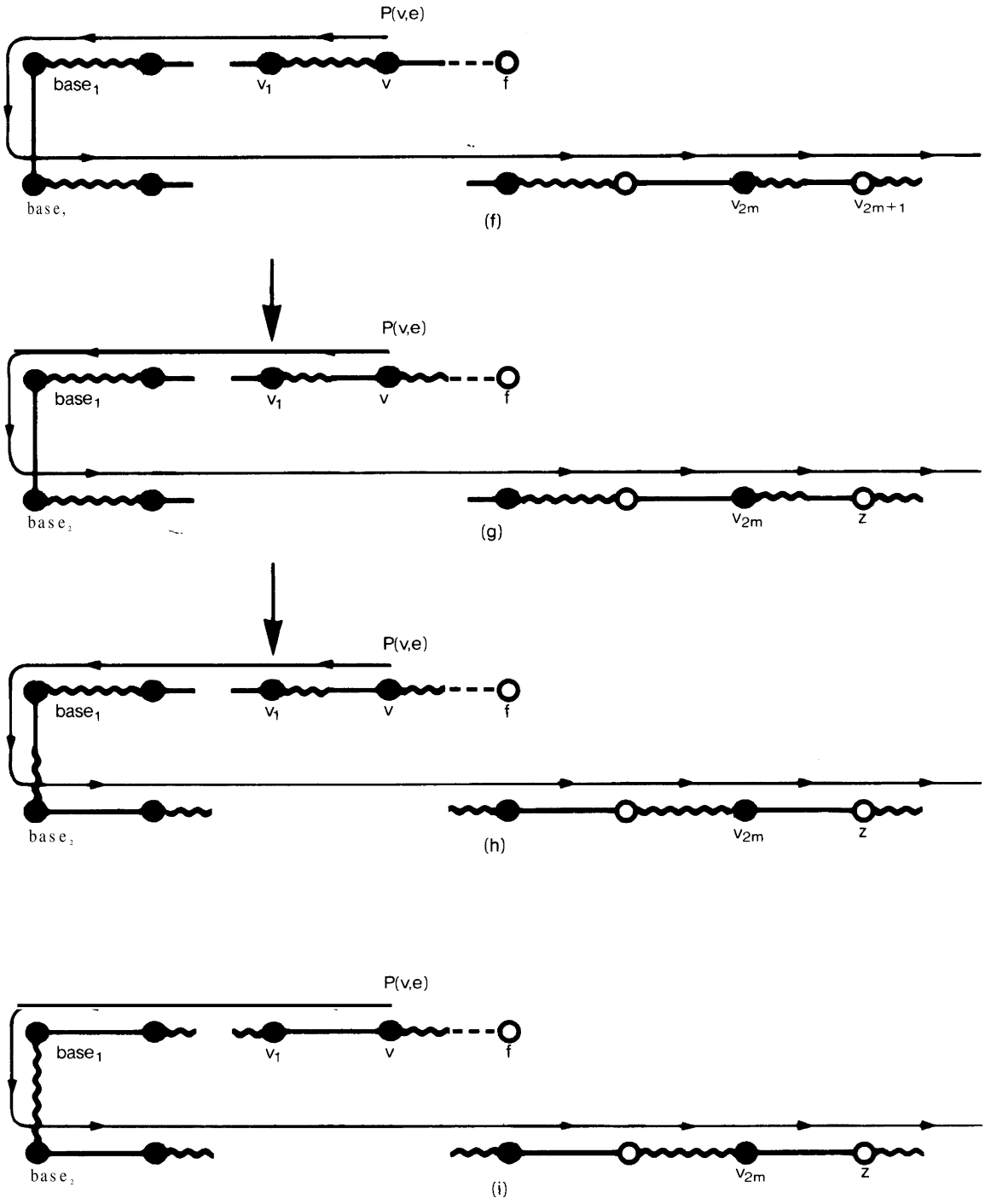


Fig. 9 (cont'd)

The path(f) * P(v,z): v has a pair link.

(f) On entry to REMATCH(f,v).

(g) On entry to REMATCH ($\underline{base}_1$,$\underline{base}_2$).

(h) On entry to REMATCH ($\underline{base}_2$,$\underline{base}_1$).

(i) On exit from REMATCH (f,v).

by x and 0 in Fig. g(a) and z = v_{2m+1} in Fig. 9(c).

Lemma 6: Suppose REMATCH(f,v) is called with v a linked vertex, vf an edge, f $\notin$ P(v,e). Set z to the first vertex in P(v,e) that is in Z, and set m so z = v_{2m+1}. Suppose these conditions hold:

(a) z in unlinked or v $\circledcirc$ z.

(b) MATE(v_i) = M(v_i)$_1$ for 0 $\leq$ i $\leq$ 2m.

Then REMATCH(f,v) returns with MATE reset in the following way:

(c) MATE(v_{2i-1}) = v_{2i}, MATE(v_{2i}) = v_{2i-1}, for 1 $\leq$ i $\leq$ m.

(d) MATE(v) = f.

In Fig. 9(a), (y) * P(x,e) satisfies conditions (a) and (b) with z = 0, m = n. Figure 9(b) illustrates conditions (c) and (d). Clearly (c) and (d) imply REMATCH works correctly.

Note vertex z of the Lemma exists. This is true because 0 $\in$ P(v,e) $\cap$ Z, since 0 = v_{2n+1} = M(e).

Proof: The proof is by induction on the linked vertices v ordered by@.

If m = 0, MATE(MATE(v)) $\neq$ v. In R1, MATE(v) is set so (d) holds. Then REMATCH returns in R4. Since condition (c) is vacuous, the Lemma is true in this case.

Suppose m > 0 and v has a pointer link. Figure 9(c) shows the path (f) * P(v,z) when REMATCH is entered. (Edge vf is shown half-dotted, meaning MATE(f) may or may not be se-i; to v.) Condition (b) shows P(v,z) is still well-defined by MATE and LINK.

Figure 9(d) shows the path after MATE(v) and MATE(v_1) are reset In R1 and R2. We see that for the recursive call REMATCH(v_1,LINK(v)), vertex z stays the same and m decreases by 1. Condition(a) holds because z is unlinked or LINK(v) $\circledcirc$ v $\circledcirc$ z, and condition(b) still holds. So by induction, REMATCH(v_1,LINK(v)) returns with edges rematched as in Fig. 9(e). So

conditions (c)-(d) are valid when REMATCH (f,v) returns.

Next, suppose m > 0 and v has a pair link. Figure 9(f) shows (f) * P(v,z) on entry to REMATCH. Note z ϵ P($\underline{base}_2$,e). This is true because Fig. 4(b)-(c) and condition (a) together imply z does not precede tip in P($\underline{base}_1$,e) or P($\underline{base}_2$,e). Figure 9(g) shows the path after R1. Note at this point, v ϵ P($\underline{base}_1$,e) $\cap$ Z and z ϵ P($\underline{base}_2$,e) $\cap$ Z.

For the recursive call REMATCH($\underline{base}_1$,$\underline{base}_2$), z is reset to v. Condition (a) holds because z is unlinked or $\underline{base}_2 \circledS v \circledS z$, condition (b) still holds. So by induction, REMATCH($\underline{base}_1$,$\underline{base}_2$) returns as shown in Fig. 9(h).

For the recursive call REMATCH($\underline{base}_2$,$\underline{base}_1$), z is reset to v. Condition (a) holds because $\underline{base}_1 \circledS v$, and condition (b) is still true. So by induction REMATCH ($\underline{base}_1$) returns as shown in Fig. 9(i). So conditions (c)-(d) are valid when REMATCH(f,v) returns.

The Lemma now follows by induction.

QED

We have shown MATCH finds valid augmenting paths and correctly rematches edges along these paths. The last two lemmas show MATCH finds all possible augmenting paths. First the search M2-M6 is proved complete.

Lemma If a vertex v is joined to e by an alternating path ($v,v_1,\ldots,v_{2n}$ = e) beginning with a matched edge vv_1, either v is eventually linked or the search M2-M6 finds an augmenting path.

Note this result shows that if an augmenting path to e exists, M2-M6 finds an augmenting path. For suppose ($f,v_0,v_1,\ldots,v_{2n}$ = e) is an augmenting path. By the Lemma, either v_0 is linked or M2-M6 finds' an augmenting path.

In the former case, M2-M6 finds (f) * $P(v_0,e)$ or some other augmenting path.

Proof: Suppose M2-M6 terminates at M2 without finding an augmenting path. Suppose v_{2i} is linked, for $1 \leq i \leq n$, and v is unlinked, as shown in Fig. 10. We derive a contradiction below. This proves the Lemma .

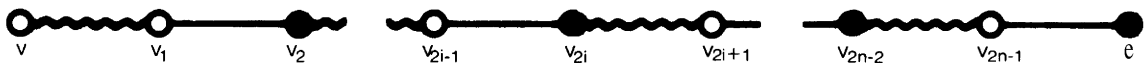


Fig. 10

We begin-by showing that for all i in $1 \leq i \leq n$, vertex $v_{2i\ 1}$ is linked and free(v_{2i-1}) = v. The proof is by induction.

First let i = 1. Note vertex v_1 is linked. For suppose the contrary. At some point in the search, in step M2, edge v_2v_1 is scanned from the linked vertex v_2. Then step M5 is executed and MATE(v_1) = v is linked. But this contradicts the original assumption that v is unlinked. We conclude v_1 is linked.

So $P(v_1,e)$ exists, and equals $(v_1,MATE(v_1) = v,...)$. Vertex v is the first unlinked vertex in this path. So the inductive assertion holds for i = 1.

Next suppose the assertion is true for some i and v_{2i-1}, where i<n. We prove the assertion for i + 1 and v_{2i+1}. At some point in the search, in step M2, edge $v_{2i\ 1}\ v_{2i}$ is scanned with both vertices v_{2i-1} and v_{2i} linked. Then step M4 is executed, and PAIR LINK (v_{2i-1}, v_{2i}) is called. This guarantees that during the rest of the search, free(v_{2i-1}) = free(v_{2i}). (See Fig. 4(c)). So v = free(v_{2i}). But $P(v_{2i},e) = (v_{2i},MATE(v_{2i}),...)$. Thus $MATE(v_{2i}) = v_{2i+1}$ is linked.

Furthermore, Property (4) of Lemma 3 implies free(v_{2i+1}) = free(MATE (v_{2i+1})) = v. So the inductive assertion holds for i + 1.

By induction, the assertion holds for all i in $1 \leq i \leq n$. In particular, v_{2n-1} is linked and free(v_{2n-1}) = v.

So at some point in the search, in step M2, edge $v_{2n-1}e$ is scanned with both vertices v_{2n-1} and e linked. Then PAIR LINK(v_{2n-1},e) is called. This invocation links v = free(v_{2n-1}).

But this contradicts the original assumption. So that assumption is false, and the Lemma is true.

QED

Now we show the algorithm halts with a maximum matching. It is clear from our discussion that MATCH always halts. Let M be the final matching in MATE.

Lemma 8: If e is an exposed vertex of M, there is no augmenting path to e.

Proof: (Witzgall and Zahn [1969]). In M1, a search for an augmenting path to e is started. Call this search S(e). S(e) ends in M2 without doing an augmentation M3. Let D be the set of edges emanating from linked vertices which are scanned in M2 during S(e). We first show no edge of D is rematched in an augmentation done after S(e).

Suppose the contrary. Let Q(f,g) be the first augmenting path MATCH finds after S(e) that includes an edge in D. Let this edge be w', with v linked to e. Choose p maximal so $v_{2p}v_{2p+1}$ is a matched edge in P(v,e) ∩ Q. As shown in Fig. 11, Q(f,g) = (f,w_0,w_1,...,w_{2q} = v_{2p}, w_{2q+1} = v_{2p+1},...,w_{2n-1},g). All vertices are shown solid, regardless of links. Note the case $w_{2q} = v_{2p+1}$, $w_{2q+1} = v_{2p}$ is possible. It is treated by a similar argument.

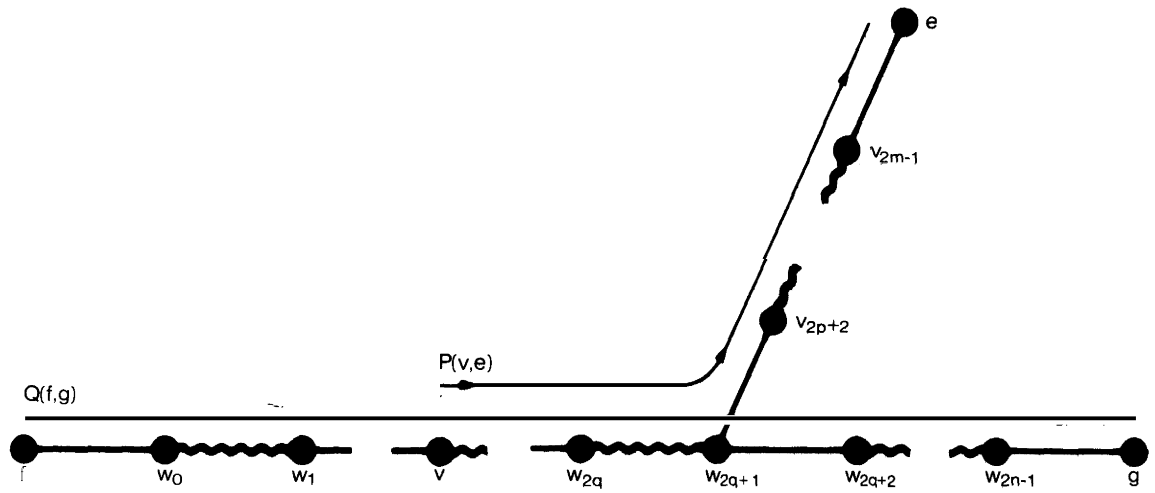


Fig. 11
The paths Q(f,g) and P(v,e).

The alternating walk $(w_0, w_1, \ldots, w_{2q-2}, w_{2q-1}, v_{2p}, v_{2p+1}, \ldots, v_{2m-1}, e)$ is simple, by the choice of p. So Lemma 7 shows w_0 is linked in S(e). But then the augmenting path (f) * $P(w_0, e)$ is discovered in S(e), contradicting the assumption e is exposed.

So no edge of D was rematched after S(e). If the search M2-M6 starting from e is repeated after MATCH halts, exactly the same edges D will be scanned. No augmenting path will be found. By Lemma 7, there is not augmenting path to e in the matching M.

QED

5. Efficiency and Applications

MATCH requires at most $O(V^3)$ time units when executed on a random access computer. For the search M2-M6 is done at most V times. We show below that each of the steps M2-M6 uses $O(V^2)$ time units per search.

Step M2 scans an edge emanating from a linked vertex. M2 may be executed twice for every edge of the graph. This requires $O(V^2)$ time units.

Step M3 calls REMATCH to augment the matching. M3 is executed at most once in a search. It requires time proportional to the length of P(v,e), or O(V) time units.

Step M4 calls PAIR LINK to assign pair links. M4 is executed for edges joining two linked vertices. So M4 may be executed $O(V^2)$ times. In all but $\left\lfloor \frac{V-1}{2} \right\rfloor$ executions, no links are assigned. PAIR LINK returns in step PL0, in constant time. In at most $\left\lfloor \frac{V-1}{2} \right\rfloor$ executions, PAIR LINK links vertices, requiring O(V) time units (in step PL7). So the total time used in M4 is $O(V^2)$.

Step M5 assigns a pointer link. M5 may be executed $\left\lfloor \frac{V-1}{2} \right\rfloor$ times. This requires O(V) time units.

Step M6 does no processing for an edge, but just transfers control. M6 may be executed O(d) times. This requires $O(V^2)$ time units.

So MATCH requires a total of $O(V^3)$ time units.

The space needed by MATCH can be seen from the listing in the Appendix. The adjacency lists of the graph require V + 4E words, where E is the number of edges. The matching, stored in MATE, uses V words. For the search M3-M6, 2.5 V words plus 2 V bits are used: 1.5 V words in the table (BASE,TOP) describing pair links, and V words (LINK) plus 2 V bits (FREE,PTR) for link information for vertices. Step M2 is implemented in a breadth-first manner, requiring a queue (LINKQUEUE) of V words.

This amounts to 2 V + 4 E words for the graph and matching, and 3.5 V words plus 2 V bits for MATCH itself.

Note procedure REMATCH is recursive, so it uses a run-time stack. It is easy to see only 1 word (LINK(L)) per recursive call need be saved. Thus at most 0.5 V words are needed for the stack. The stack may share the storage allocated to LINKQUEUE, since these two data areas exist at different times.

MATCH can be used to speed up the scheduler devised by Fujii, Kasami, and Ninomiya [1969]. They solved this problem: Compute an optimum schedule for N tasks to be executed by 2 processors, assuming the tasks have equal length and arbitrary precedence constraints. The approach is to construct a compatibility graph, showing which tasks may be executed simultaneously; find a maximum matching on the compatibility graph; sequence the matched task pairs and the unmatched tasks according to precedence constraints. This algorithm was thought to require time proportionalto N^4 [Fujii, Kasami, and Ninomiya, 1969-erratum]. But the first and last steps may be executed in time N^3, and we have shown the matching can be done in time N^3. So the scheduler is an N^3 algorithm.

MATCH can be generalized to find maximum matchings on weighted graphs. In a weighted graph, each edge has a weight which is a real number. The problem is to find a matching with maximum weight. Matching on ordinary graphs is the special case of this problem where all edges have the same weight. An algorithm has been developed which takes time proportional to V^3 log v. This and other generalizations are currently being investigated and programmed.

6. Acknowledgement

The author wishes to thank Professor Harold Stone for introducing him to the problem of maximum matching,- for many stimulating conversations, and for reviewing the manuscript with great energy and perspicacity.

7. Appendix

This section contains a listing of an ALGOL W program for the maximum matching algorithm.

Global Storage Declarations

```
BEGIN INTEGER V,E; STRING( 10) NAME;
COMMENT   V      IST H E NUMBER OF VERTICES IN THE GRAPH.
          E      I  STHE NUMBER OF EDGES IN THE GRAPH.
          NAME   I STHENAMEOF THE GRAPH;
INTFIELDSIZE:=3;
READ (NAME,V,E);
COMMENT   PROCESS EACH GRAPH U N T I L END-OF-DATA CARD IS READ;
WHILE V>0 DO
BEGIN
INTEGER ARRAY NEIGHBOR(V+1::V+2*E);
INTEGER ARRAY NEXT(1::V+2*E);
L IGICAL ARRAY FREE,PTR (0::V);
INTEGER ARRAY LINK,MATE (0::V);
INTEGER ARRAY BASE (1::   (V-1) DIV 2,1::2);
INTEGER ARRAY TOP( 1 : :(V-1) DIV 2 ) ;
INTEGER ARRAY L INKQUEUE(1 : :V);
INTEGER HEAD,TAIL,PAIRNUM,LINKVTX,PLACE,NBHR,H;
INTEGER TIP,F,J;
INTEGER ARRAY FREEVTX(1::2);

COMMENT   NEIGHBOR    CONTAINS THE ADJACENCY LISTS OF TIt GRAPH.

          NEXT(X)     I F XIS A VERTEX, THE ADJACENCY LIST OF X IS
                      ( NE IGHBOR (NEXT ( X) ), NEIGHBOR(NEXT(NEXT(X))),...).
                      THE LAST VERTFX IN THE LIST IS NEIGHBOR(Y),
                      WHERE NEXT(Y)I  SO.

          FREE(X)     I STRUE IF VERTEX X IS UNLINKED.

          PTR(X)      I SFALSE I F VERTEX X H A S A PAIR LINK.

          LINK(X)     I fVERTEX X HAS A POINTER LINK, LINK(X)I S
                      THE POINTER.
                      I F VERTEX X H A S A PAIR LINK, LINK(X) I S THE
                      NUMBER OF TI:E PAIR L I N K . IT IS USED A S A N
                      INDEX INTO BASE AND TGP.

          MATE(X)     I F VERTEX X I S O N A MATCHED EDGE, MATE(X) I S
                      THE VERTEX MATCHED TO X.
                      IF VFRTFX X I S EXPOSED, MATE(X) IS 0.

          BASE(N,I)   I FN I S THE NUMBER OF A P A I R LINK, BASE(N,1)
                      A N D BASE(N,2) ARE THE ADJACENT L INKED V E R T I C E S
                      W H I C H FORM THE P A I R.

          TOP(N)      IF N IS THE NUMBER OF A PAIR LINK, AND X IS A
                      L I N K E D V E R T E X WITH LINK N, THEN TOP(N) IS T H E
                      F I R S T UNLINKED V E R T E X IN P(X, EXPOSED VTX), THE
                      A L T F P N A T I N G PATH FROM X TO THE EXPOSED V E R T E X.
```

LINKQUEUE	CONTAINS THE QUEUE OF LINKED VERTICES TO BE EXAMINED.
HEAD	POINTS TO THE FIRST ENTRY IN THE QUEUE.
TAIL	POINTS TO THE LAST ENTRY IN THE QUEUE.
PAIRNUM	STORES THE NEXT PAIR LINK NUMBER TO BE ASSIGNED;

Routines for Reading and Printing a Graph

```
PROCEDURE READGRAPH;
COMMENT THIS PROCEDURE READS THE GRAPH AND CONVERTS IT TO ADJACENCY
        LISTS IN NEIGHBOR AND NEXT;
  BEGIN INTEGER V1,V2;
  FOR I:=1 UNTIL V DO NEXT(I):=0;
  FOR I:= V+2*E STEP -2 UNTIL V+2 DO
     BEGIN
     READON(V1,V2);
     NEIGHBOR(I):=V2;
     NEXT(I):=NEXT(V1);
     NEXT(V1):=I;
     NEIGHBOR(I-1):=V1;
     NEXT(I-1):=NEXT(V2);
     NEXT(V2):=I-1;
     END;
  END READGRAPH;

PROCEDURE WRITEGRAPH;
COMMENT THIS PROCEDURE WRITES THE ADJACENCY LISTS OF THE GRAPH;
  BEGIN
  WRITE(" "); WRITE(" ");
  WRITE("***** ",NAME,"*****");
  WRITE("V=",V,"E=",E);
  WRITE("ADJACENCY LISTS");
  FOR I:=1 UNTIL V DO
     BEGIN
     WRITE(I,":");
     J:=NEXT(I);
     WHILE J>0 DO
       BEGIN
       WRITEON(NEIGHBOR(J));
       J:=NEXT(J);
       END
     END;
  END WRITEGRAPH;
```

Routines for Searching for Augmentations

```
PROCEDURE SEARCH(INTEGER VALUE EXPOSEDVTX);
COMMENT    THIS PROCEDURE SEARCHES FOR AN AUGMENTING PATH TO EXPOSEDVTX,
           AN EXPOSED VERTEX. IT SCANS EDGES OF THE GRAPH, DECIDING WHEN
           TO ASSIGN LINKS AND PERFORM AN AUGMENTATION;

  BEGIN
  WRITE("SEARCH FOR EXPOSED VTX", EXPOSEDVTX);
  COMMENT    INITIALIZE. LINK EXPOSEDVTX, AND MAKE ALL OTHER VERTICES
             UNLINKED;
  FOR I:=0 UNTIL V DO FREE(I):=PTR(I):=TRUE;
  FREE(EXPOSEDVTX):=FALSE;
  LINKQUEUE(1):=EXPOSEDVTX;
  PAIRNUM:=HEAD:=TAIL:=1;
  COMMENT    THIS LOOP SETS LINKVTX TO A LINKED VERTEX FROM LINKQUEUE
             AND EXAMINES THE EDGES EMANATING FROM LINKVTX;
  WHILE HEAD<=TAIL DO
     BEGIN
     LINKVTX:=LINKQUEUE(HEAD);
     HEAD:=HEAD+1;
     PLACE:=NEXT(LINKVTX);
     WHILE PLACE¬=0 DO

       BEGIN
       COMMENT  SET NBHR TO THE NEXT VERTEX ADJACENT TO LINKVTX;
       NBHR:=NEIGHBOR(PLACE);
       PLACE:=NEXT(PLACE);
       COMMENT  IF NBHR IS LINKED, ASSIGN PAIR LINKS;
       IF ¬FREE(NBHR) THEN PAIRLINK(LINKVTX,NBHR)
       ELSE IF MATE(NBHR)=0 THEN
          BEGIN
          COMMENT  IF NPHR IS EXPOSED AUGMENT THE MATCHING;
          MATE(NBHR):=LINKVTX;
          WRITE("AUGMENT"),
          REMATCH(NBHR,LINKVTX);
          GOTO DONE
          END
     . COMMENT  IF NBHR AND MATE(NBHR) ARE UNLINKED, ASSIGN A
                POINTER LINK;
       ELSE IF FREE(MATE(NBHR)) THEN MAKELINK(LINKVTX,MATE(NBHR));
       END WHILEPLACE;
    END WHILEHEAD;
 DONE:
 END SEARCH;
```

Routine for Assigning Pair Links

```
PROCEDURE PAIRLINK (INTEGER VALUE BASE1,BASE2);
COMMENT  THIS PROCEDURE ASSIGNS PAIR LINKS TO UNLINKED VERTICES IN
         P(BASE1,EXPOSEDVTX) AND P(BASE2,EXPOSEDVTX). BASE1 AND BASE2
         ARE ADJACENT LINKED VERTICES.
         THESE VARIABLES ARE USED IN PAIRLINK:
         FREEVTX(I)    IF I IS 1 OR 2, FREEVTX(I) STEPS THROUGH THE
                       UNLINKED VERTICES IN P(BASE1,EXPOSEDVTX).

         TIP           IS SET TO THE FIRST UNLINKED VERTEX THAT IS IN
                       BOTH P(BASE1,EXPOSEDVTX) AND P(BASE2,EXPOSEDVTX);
  BEGIN

  INTEGER PROCEDURE FIRSTFREE (INTEGER VALUE L);
  COMMENT  THIS PROCEDURE RETURNS THE VALUE OF THE FIRST UNLINKED
           VERTEX IN P(L,EXPOSEDVTX);
   BEGIN
   COMMENT  STORE THE VALUE IN THE GLOBAL VARIABLE F AND
            RETURN F;
   F:= IF FREE(MATE(L)) THEN MATE(L)
       ELSE
       TOP(LINK(IF PTR(L) THEN MATE(L) ELSE L));
  F
   END;

  FREEVTX(1):=FIRSTFREE(BASE1);
  COMMENT IF THE FOLLOWING TEST FAILS THE PROCEDURE EXITS,
          SINCE NO LINKS MAY BE ASSIGNED;
  IF FREEVTX(1)¬=FIRSTFREE(BASE2) THEN
    BEGIN
    PTR(FREEVTX(1)):=FALSE;
    FREEVTX(2):=F;
    J:=2;

    COMMENTTHISLOOPFLAGSUNLINKEDVERTICESALTERNATELY IN
            P(BASE1,EXPOSEDVTX) AND P(BASE2, EXPOSEDVTX ), UNTIL THE
            FIRST COMMON UNLINKED VERTEXI  SFOUND. A VERTEXI  S
            FLAGGEDBYSETTINGITSPTRVALUE TOFALSE;
    WHILE PTR (F) DO
      BEGIN
      PTR(F):=FALSE;
      J:=3-J;
      COMMENTI F  T H E END OF P(BASEJ,EXPOSEDVTX) HAS BEEN REACHED,
               DON'T GO ANY FURTHER;
      I FFREEVTX ( J)=0 THEN J:=3-J;
      FREEVTX(3) :=FIRSTFREE(LINK(MATE(FREEVTX(J))));
      E-ND;
```

```
    COMMENT  MAKE ENTRIES IN BASE AND TOP;
    TOP(PAIRNUM):=TIP:=F;
    BASE(PAIRNUM,1):=BASE1;
    BASE(PAIRNUM,2):=BASE2;
    WRITE("PAIR: (",BASE1,BASE2,") TIP IS",TIP,"      ");

    COMMENT  RESET PTR TO TRUE FOR VERTICES ABOVE TIP;
    PTR(F):=TRUE;
    WHILE ¬PTR(FIRSTFREE(LINK(MATE(F))) ) DO
      PTR(F):=TRUE;

    COMMENT  LINK ALL UNLINKED VERTICES WHICH PRECEDE TIP IN
             P(BASE1,EXPOSEDVTX) AND P(BASE2,EXPOSEDVTX);
    FOR I:=1,2 DO
      IF FIRSTFREE(BASE(PAIRNUM,I))¬=TIP THEN
        BEGIN
        MAKELINK(PAIRNUM,F);
        WHILE FIRSTFREE(LINK(MATE(F)))¬=TIP DO
          MAKELINK(PAIRNUM,F);
        END;

    COMMENT  RESET  ENTRIES I  N TOP ARRAY WHICH HAVE JUST BEEN
             LINKED;
    FOR I  :=1 U N T I L PAIRNUM-1 D  C
       IF ¬FREE(TOP(I)) THEN TOP(I):=T I P ;

    COMMENT  R U M P PAIRNUM FOR THE N E X T PAIR L I N K;
    PAIRNUM  :=  PAIRNUM+1;
    END;
END PAIRLINK;
```

Routine for Assigning Links

```
PROCEDURE MAKELINK(INTEGER V A L U E L,FREEVTX);
COMMENT  THIS PROCEDURE A S S I G N S  4  L I N K  L  T O  A VERTEX FREEVTX;
     BEGIN
     FREE(FREEVTX):=F A L S E;
     L INK(FREEVTX) :=  L;
     COMMENT  PLACE F K E F V T X  A T  T H E  F N D OF THE QUEUE OF LINKED
              VERTICES;
     TAIL:=TAIL+1;
     LINKQUEUE(TAIL):=FREEVTX;
     I F PTR(FREEVTX) T H E N WRITE("PTR:");
     WRITEON(FREEVTX,L,"    ");
     END MAKELINK;
```

Routine for Rematching

```
PROCEDURE REMATCH(INTEGER VALUE F,L);
COMMENT THIS PROCEDURE MATCHES L TO F AND CONTINUES REMATCHING
        ALONG P(L,EXPOSEDVTX) BY CALLING ITSELF RECURSIVELY;
  BEGIN
  WRITEON(" MATCH",F,L);
  H:=MATE(L);
  MATE(L):=F;
  COMMENT  IF THE FOLLOWING TEST FAILS, THE REMATCHING ALONG
           P(L,EXPOSEDVTX) IS COMPLETE;
  IF MATE(H)=L THEN
    IF PTR(L) THEN
      BEGIN
      COMMENT IF L HAS A POINTER LINK, REMATCH ALONG P(L,EXPOSEDVTX);
      MATE(H):=LINK(L);
      REMATCH(H,LINK(L));
      END
    ELSE
    COMMENT  IF L HAS 4 PAIRLINK, REMATCH ALONG P(BASE1,EXPOSEDVTX)
             AND P(BASE2,EXPOSEDVTX);
    FOR I:=1,2 DO
    REMATCH(BASE(LINK(L),I),BASE(LINK(L),3-I));
  END REMATCH;
```

Driver Routine

```
  COMMENT THIS IS THE MAIN PROGRAM;

  COMMENT  READ INPUT GRAPH AND STORE IT IN ADJACENCY LISTS;
  READGRAPH;
  COMMENT  WRITE OUT THE ADJACENCY LISTS;
  WRITEGRAPH;
  WRITE("START MAT",TIME(1));
  COMMENT  INI        TIALIZE;
  FOR I:=0 UNTIL V DO MATE(I):=0;
  LINK(0):=0;
  COMMENT  SEARCH FOR AUGMENTING PATHS TO EACH EXPOSED VERTEX:
  FOR I:=1 UNTIL V DO IF MATE(I)=0 THEN SEARCH(I);
  WRITE("END MAT",TIME(1));
  COMMENT  WRITE OUT THE MATCHING;
  WRITE("MAXIMAL MATCHING:");
  FOR I:=1 UNTIL V DO WRITECN("  ",I,MATE(I));
  COMMENT  BEGIN THE NEXT GRAPH;
  READ(NAME,V,E);
  END
  END.
```

References

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, Chapel Hill, North Carolina, pp. 585-602, 1967.

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

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

Fujii, M., Kasami, T., and Ninomiya, K., 1969. "Optimal sequencing of two equivalent processors," SIAM Journal of Applied Mathematics, vol. 17, pp. 784-789, 1969, and erratum, Vol. 20, p. 141, 1971.

Harary, F., 1969. Graph Theory, Addison-Wesley, Reading, Mass., 1969.

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

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