アルゴリズム 21－5
（1991. 5. 29）

無向多重グラフのすべての3-辺成分を求める線形時間アルゴリズム

田岡智志　渡辺敏正　翁長健治

広島大学工学部
724 東広島市西条町鏡山1丁目4-1

本稿の目的は与えられた多重辺無向グラフG=(V,E)のすべての3−辺成分を求める$O(|V|+|E|)$時間アルゴリズムを提案することである．グラフの3−辺成分とは その中の任意の2点間に辺素なパスが3本以上存在するような極大な点集合である．提案アルゴリズムを使えば，Gが3-辺連結であるか否かを$O(|V|+|E|)$時間で判定でき，またGのすべてのカットペアを$O(|V|^2+|E|)$時間で求めることができる．

アルゴリズムは深さ優先探索(DFS)に基づく．一般性を失うことなくGは2-辺連結と仮定できる．1つのDFS木Tを固定するとき，カットペアは2つのタイプに分けられる: T上の枝と後退枝から成るtype 1ペア; T上の2本の枝から成るtype 2ペアである．すべてのtype 1ペアは$O(|V|+|E|)$時間で容易に求めることができる．重要な点はすべてのtype 2ペアを含む枝集合KE(T)が$O(|V|+|E|)$時間で求められることである．type 1ペアかKE(T)のいずれかに含まれる枝をGからすべて除去することによりGのすべての3-辺成分が求められる．

Computing All 3-Edge-Components of an Undirected Multigraph in Linear Time

Satoshi Taoka, Toshimasa, Watanabe and Kenji Onaga

Faculty of Engineering, Hiroshima University,
4-1, Kagamiyama 1 chome, Higashi-Hiroshima, 724 Japan.

The subject of the paper is to propose an $O(|V|+|E|)$ algorithm for finding all 3-edge-components of a given multigraph G=(V,E). This algorithm can be used in detecting whether G is 3-edge-connected or not in $O(|V|+|E|)$ time, or can be modified into an $O(|V|^2+|E|)$ algorithm for determining all cutpairs of G. An 3-edge-component of G is defined as a maximal set of vertices such that G has at least three edge-disjoint paths between everypair of vertices in the set. The algorithm is based on the depth-first search (DFS) technique. For any fixed DFS-tree T of G, cutpairs of G are partitioned into two types: a type 1 pair consists of an edge of T and a back edge; a type 2 pair consists of two edges of T. All type 1 pairs can easily be determined in $O(|V|+|E|)$ time. The point is that an edge set KE(T) in which any type 2 pair is included can be found in $O(|V|+|E|)$ time. All 3-edge-components of G appear as connected components if we delete from G all edges contained in type 1 pairs or in this edge set KE(T).

1. Introduction

The subject of the paper is to propose an $O(|V|+|E|)$ algorithm for finding all 3-edge-components of a given multigraph G=(V,E). This algorithm can be used in detecting whether G is 3-edge-connected or not in $O(|V|+|E|)$ time, or can be modified into an $O(|V|^2+|E|)$ algorithm for determining all cutpairs of G. This algorithm and the result in [11,12] make an $O(|V|+|E|)$ algorithm for the 3-edge-connectivity augmentation problem. An *3-edge-component* of G is defined as a maximal set of vertices such that G has at least three edge-disjoint paths between every pair of vertices in the set. A *cutpair* is a pair of edges whose deletion from G result in a graph with more components than G and such that deleting only one edge of the pair does not have such a property.

Generally, for $m \geq 1$, an m-edge-components of a multigraph G (an m-vertex-component of a simple graph G, respectively) is defined as a maximal set of vertices such that, for any pair of vertices in the set, G has at least m edge-disjoint (m internally-disjoint) paths between them [8,9,10]. It is known that if $m \leq 2$ then all m-edge-components and m-vertex-components can be found in $O(|V|+|E|)$ time by using DFS (see [1,2]). [5] proposed an $O(|V|+|E|)$ algorithm for dividing a graph G into triconnected components, and it is useful in computing all 3-vertex-components. [9] showed an $O(|V|(|V|+|E|))$ algorithm for finding all 3-edge-components. It is known that finding all m-edge-components (m-vertex-components, respectively) can be done by repeating a maximum flow algorithm $O(|V|)$ times [6] ($O(|V|^2)$ times; see [2]). Recently [7] has shown that, for any given $k \geq 1$, all m-edge-components with $m \leq k$ can be computed in $O(|E|+k^2|V|^2)$ time.

2. Preliminaries

Some definitions are explained with examples. Technical terms not specified here can be identified in [1,4,7,]. An undirected multigraph G=(V,E) consists of a vertex set V and an edge set E; they may be written as V(G) or E(G). Since any 3-edge-component is a subset of a 2-edge-component, we can assume without loss of generality that G is 2-edge-connected throughout the paper unless otherwise stated. If G is directed then the edge set is denoted as A(G). A directed edge from u to v is denoted by <u,v>; an undirected edge is written as (u,v). A directed path from u to v of G is called a <u,v>-path and is denoted as P_G<u,v>; P_G(u,v) is used for an undirected path. The subscript G is often omitted unless any confusion arises. For $P \subseteq V(G)$ and $Q \subseteq E(G)$, the graph defined by their deletion is denoted by $G-(P \cup Q)$. If $P \cup Q=\{s\}$ then we denote as G-s. An *m-component* means an m-edge-component unless otherwise stated. The *edge-connectivity* of G is denoted by ec(G). A *bridge* is an edge e whose deletion increases the number of components.

Suppose that $ec(G) \geq 2$. Choose any vertex r and execute DFS starting from r. A directed graph $\vec{G}=(V,\vec{E})$ is defined from G by this DFS, and directed edges in E are partitioned into two sets A(T) and BA(T): A(T) defines a directed spanning tree T=(V,A(T)), called a *DFS-tree*; BF(T) does a graph BF(T)=(V,BA(T)), called the *back forest*. Edges of T (of BF(T), respectively) are called *tree edges* (*back edges*). T is fixed in the following. T<v> denotes the subtree of T having v as the root. Let dfn(v) denoted the order of visit to v by this DFS, and, for simplicity, v is identified with dfs(v) unless otherwise stated. If G is a multigraph of Fig. 1 then an example of T and BF(T) are shown in Fig. 2, where each number denotes dfn(v). G and $\vec{G}$ are used interchangeably for notational simplicity: although any cutpair {(v,w),(x,y)} is a subset of E(G), we often consider the pair as {<v,w>,<x,y>}$\subseteq \vec{E}$ and say that {<v,w>,<x,y>} is a cutpair of G or of $\vec{G}$. Graphs G and $\vec{G}$ are given as a set of edge lists LG(v), $v \in V$. The DFS_tree T and the back forest BF(T) are represented as sets of edge lists LT(v) and LB(v), $v \in V$, respectively. Each of LG(v), LB(v) and LT(v) consists lists of edges incident upon v. An edge <v,w> is often denoted as (v,w).

Cutpairs of G are partitioned into two type: a *type 1 pair* consists of a tree edge and a back edge; a *type 2 pair* consists of two tree edges. For each $v \in V(G)$, let

$lowpt_G(v)=\min\{b \mid$ T has a <v,a>-path and there is a back edge <a,b>$\}$,

LE(v)=(a,b), where $b=lowpt_G(v)$ and the back edge <a,b> is the one by which b is set to $lowpt_G(v)$,

$medium_G(v)=lowpt_{G'}(v)$, where G'=G-LE(v).

It may happen that $lowpt_G(v)=medium_G(v)$. The value $lowpt_G(v)$ is known as the "lowpoint" number (see [1]). The subscript G is often omitted for simplicity unless any confusion arises. If v=5 in Fig. 2 then lowpt(v)=1, LE(v)=(1,8) and medium(v)=2. We partition V(G) into sets $Q_1,\ldots,Q_p$, where each Q_i is a maximal set such that $lowpt_G(u)=lowpt_G(u')$ for any pair u, u' in the set. Let F_i denote the subgraph of T such that the edge set is $\{<u,u'> \in A(T) \mid u' \in Q_i\}$, $1 \leq i \leq p$. Generally F_i consists of some directed trees. For G of Fig. 2, we have $V(F_1)=\{1,\ldots,8\}$ with lowpt(u)=1, $V(F_2)=\{5,9,10,11\}$ with lowpt(u)=5, $V(F_3)=\{9,12,13\}$ with lowpt(u)=9 and $V(F_4)=\{4,14\}$ with lowpt(u)=4. We partition each F_i into one or more edge-disjoint paths. For simplicity we explain how to partition F_i in the case where F_i itself is a tree. First choose a directed path P_{i1} from the root of F_i (from a source vertex of F_i in general) to a leaf, and $F_i \leftarrow F_i-A(P_{i1})$. IF $A(F_i) \neq \phi$ then find a maximal directed path P_{i2} containing an edge $<v',w'> \in A(F_i)$ with $v'=\min\{v'' \in V(P_{i1}) \mid <v'',w''> \in A(F_i)\}$. $F_i \leftarrow F_i-A(P_{i2})$ and repeat the same procedure until $A(F_i)=\phi$. Let $\{PB_1,\ldots,PB_n\}$ be the set of all such paths finally obtained, where the starting or ending vertex of PB_i is denoted as s_i or t_i, respectively. Note that $A(PB_i) \cap A(PB_j)=\phi$, if $i \neq j$. This set of paths are called a *path-partition* of T. It will be shown in Section 5 that a path-partition of T can be obtained in $O(|V|+|E|)$ time. A path-partition $\{PB_1,\ldots,PB_4\}$ is given in Fig. 3, where $V(PB_i)=V(F_i)$ and PB_i is denoted by bold directed lines, $1 \leq i \leq 4$. It will be shown in Section 6 that any type 2 pair is included in some PB_i.

The following definitions are given with respect to each PB_i. For any vertex $u \in V(PB_i)$, let $T'=T-V_u$, where

$$V_u=\begin{cases} V(T<v>) & \text{if } <u,v> \in A(PB_i), \\ \phi & \text{if } u=t_i. \end{cases}$$

A path P_G<u,u'> is called a *back-path* of u (with respect to PB_i) if the following (1)-(3) hold:

(1) $u' \in V(P_T<r,u>)$,

(2) any inner vertex of P_G<u,u'> is not contained in $V(PB_i)$,

(3) the last edge<u",u'> of P_G<u,u'> is a back edge and any other edge is in A(T').

There may be more than one back-path of u. For PB_1 of Fig. 3, there are two back-paths of the vertex 4 shown in Fig. 2, where both of their vertex sets are {4,14,3}. For each vertex $u \in V(PB_i)$, define

$B_i(u)=\{u' \mid$there is a back-path P_G<u, u'> of u with

respect to PB_i}$\cup${u},

$$local\_min_i(u)=\begin{cases} s_i & \text{if } minB_i(u)<s_i, \\ minB_i(u) & \text{otherwise,} \end{cases}$$

$local\_high_i(u)=max\{\{u' \in V(P_T<u,t_i>)|$ there is a back-path $P_G<u',u>$ of u' with respect to $PB_i\}\cup\{u\}\}$.

In PB_1 of Fig. 3, $local\_min_1(4)=3$, $local\_min_1(5)=5$, $local\_high_1(3)=4$ and $local\_high_1(7)=8$.

3. Type 1 pairs

First we find all type 1 pairs of G in this section.

Lemma 1. Let {(v,w),(x,y)} be a pair of edges such that $<v,w> \in A(T)$ and $<x,y> \in BA(T)$. {(v,w),(x,y)} is a type 1 pair if and only if $lowpt(w)<w$, LE(w)=(x,y) and medium(w)=w.

Proof. Suppose that $lowpt_G(w)<w$ and $medium_G(w)=w$. Then G'=G-LE(w) has $lowpt_{G'}(w)=medium_G(w)$, meaning that both v and w are cutpoints of G'. That is, {(v,w), LE(w)} is a type 1 pair.

Conversely suppose that {(v,w), (x,y)} is a type 1 pair. Since G is 2-edge_connected, LE(w)(=(x,y)) always exists and (x,y) is a bridge of G-(v,w). Hence lowpt(w)<w. We have medium(w)≤w by the definition. If suffices to show that medium(w)≥w. Assume that medium(w)<w. Then, by the definition of medium(w), $\bar{F}=\bar{G}-\{<v,w>, <x,y>\}$ has a directed path beginning from w, passing through edges of T intermediately and the edge $<a,b> \in BA(T)$ with b=medium(w) finally where b is on the <r,x>-path. This means that {(v,w), (x,y)} is not a cutpair, a contradiction. Thus medium(w)=w.◊

All type 1 pairs of G can be found by means of procedure type_1(v) in $O(|V|+|E|)$ time. First execute the following initialization:

```
i←1;
LT(v)←φ, LB(v)←φ , LE(v)←dummy for all v∈V;
choose a vertex r as a starting vertex,
```

and then the following procedure type_1(v) is repeated.

```
procedure type_1(v);
begin
  dfn(v)←i; lowpt(v)←i; medium(v)←i; i←i+1;
  for every edge (v,w) of LG(v) do begin
    [G is represented by a set of lists LG(v) of edges
    incident v∈V(G).]
    if w is unvisited then begin
      LT(v)←LT(v)∪{(v,w)};
      [LT(v) is a list of incident edges upon v]
      type_1(w);
      if lowpt(w)<lowpt(v) then begin
        medium(v)←min{lowpt(v),medium(w)};
        lowpt(v)←lowpt(w); LE(v)←LE(w) end
      else
        medium(v)←min{medium(v),lowpt(w)};
      if (lowpt(w)<w) and (medium(w)=w) then
        mark edges (v,w) and LE(w) "type 1" end
    else
      if (<v,w>∉A(T)) and
            (dfn(v)>dfn(w)) then begin
        LB(v)←LB(v)∪{(v,w)};
          [LB(v) is a list of incident edges]
        if dfn(w)<lowpt(v) then begin
          medium(v)←lowpt(v);
          lowpt(v)←dfn(w); LE(v)←<v,w> end
        else
          medium(v)←min{medium(v),dfn(w)} end
  end
end;
```

Lemma 2. All type 1 pairs of G can be found in $O(|V|+|E|)$ time by using procedure type_1(v).

Proof. For any fixed vertex v∈V, we can prove the following proposition by induction on the number that procedure type_1(v) moves (forward or backward), from v to other vertices:

Proposition 1. Let A(v) denote the set of all back edges that have been visited by the procedure until it moves from v to any other vertex. Then the current values of lowpt(v) and medium(v) satisfy the following (i) through (iii):

(i) $lowpt(v)=min\{b|<a,b> \in A(v)\}$ if $A(v)\neq\phi$.

(ii) LE(v)=(a,b) if LE(v)≠dummy, where <a,b> is the edge which sets b=lowpt(v).

(iii) $medium(v)=min\{w|<v,w> \in A(v)-\{\overline{LE(v)}\}$ if $A(v)-\{\overline{LE(v)}\}\neq\phi$, where $\overline{LE(v)}$ denotes the back edge corresponding LE(v).

Hence it is shown that procedure type_1(v) correctly compute lowpt(v), medium(v) and LE(v) when the procedure moves backward from v. This also shows that, by Lemma 1, if any type 1 pair {(v,w), (x,y)} exists then it will surely be found by the procedure (lines 13 and 14) when it moves backward from v. Clearly the procedure has $O(|V|+|E|)$ time complexity. ◊

4. Path-partition, local_min and local_high

It is shown that a path-partition of T can be found in $O(|V|+|E|)$ time by using procedure path_partition(v) and that computing two values local_min(u), local_high(u) for all u∈V can be done in $O(|V|+|E|)$ time by means of **procedure** comp_local(v).

A path-partition of T can be obtained in $O(|V|+|E|)$ time by repeating procedure path_partition(v). First execute the following initialization:

```
the starting vertex r←1; i←1;
path_number(1)←1; path_head(1)←1;
path_tail(1)←the tail of LE(1);
path_number(v)←0 for all v∈V-{1}
```

and then the following procedure path_partition(v) is repeated.

```
procedure path_partition(v);
begin
  for every edge (v,w) of LT(v)
            with path-number(w)=0 do begin
    if LE(v)=LE(w) then
      path_number(w)←path_number(v);
    else begin
      i←i+1; path_head(i)←v ; n←i;
      path_tail(i)←the tail of LE(w);
      path_number(w)←i end;
    path_partition(w) end
end;
```

Repeating procedure path_partition(v) for T assigns each v∈V a value path_number(v)≥1, for which the following (1) and (2) hold.

(1) Suppose that v<w and LE(v)=LE(w) for v,w∈V. Then, by the definition of lowpt(w) and LE(w), any inner vertex $u \in V(P_T<v,w>)-\{v,w\}$ has LE(u)=LE(v). Hence all vertices of $V(P_T<v,w>)$ have the same path_numbers.

(2) Any w' with LE(w')≠LE(v) has path_number(w')≠path_number(v).

Therefore V is partitioned into n sets $V^{(1)},\ldots,V^{(n)}$, where $V^{(i)}=\{v|path\_number(v)=i\}$, $V^{(i)}\cap V^{(j)}=\phi$ $(i\neq j)$.

Let P_i' denote the subgraph induced by $V^{(i)}$ of T. P_i' is a path. Define a path P_i as follows: $P_i = P_i'$ if path_head(i)$\in V^{(i)}$ (that is, i=1) ; P_i be the subgraph induced by $V^{(i)} \cup \{\text{path\_head(i)}\}$ if path_head(i)$\notin V^{(i)}$. A(T) is partitioned into n sets $A(P_1),\dots,A(P_n)$, where $A(P_i)\cap A(P_j)=\phi$ ($i\neq j$). Clearly $\{P_1,\dots,P_n\}$ is a path-partition of T.

If should be mentioned that each P_i is actually represented by means of $V^{(i)}$ and path_head(i). Therefore path_number(path_head(i)) is not defined if i>1.

Clearly it is shown that a path-partition of T can be obtained in $O(|V|+|E|)$ time. We denote

path_head(i)=s_i, path_tail(i)=t_i ($1\le i\le n$)

in the following.

Suppose that a path-partition $\{PB_1,\dots,PB_n\}$ is obtained. We assume that PB_i has path-number(i): these have been determined in finding a path-partition of T. The following procedure comp_local for computing these two values uses a one-dimensional array *joint*, where *joint*(i) maintains the vertex of $V(PB_i)\cap V(PB_j)$ for each PB_j with $V(PB_i)\cap V(PB_j)\neq\phi$ (see Fig. 6). This is done by the assignment *joint*(i)$\leftarrow$ v whenever the procedure moves forward from v$\in V(PB_i)$ to w$\in V(PB_j)$, j$\neq$i, such that <v,w>$\in$ A(T). The array *joint* is used in computing local_high_m. Suppose that DFS, starting from r=1, reaches u$\in V(PB_j)$ by way of some paths $PB_1,\dots,PB_i,PB_h,\dots,PB_j$ as shown in Fig. 6. If there is a back edge <u,u'> with u'$\in V(PB_i)$ then a back-path of the vertex a (with respect to PB_i) is discovered, where i=path_number(u') and the case with a=u' may happen. The point is that the vertex a is kept in *joint*(i). That is, the starting vertex a of any back-path whose ending vertex is u' can be identified as joint(i), and this makes computing $\text{local\_high}_m(u)$ efficient.

First the following initialization is done:

$\text{local\_min}_m(v)\leftarrow v$ and $\text{local\_high}_m(v)\leftarrow v$,

for every vertex v$\in$V,

where m=path_number(v). Then we repeat procedure comp_local(v) for v$\in$V.

```
procedure comp_local(v);
begin [During DFS for T, execute (1) and (2).]
[(1) if computing local_min_m(v) when DFS moves
forward from v to a son w of v in T]
for every edge (v,w)∈LT(v) do begin
  i←path_number(v); j←path_number(w);
  if i≠j then [v=path_head(j)] begin
    joint(i)←v;
    if lowpt(w)<path_head(i) then
      local_min_i(v)←path_head(i) end
    else
      local_min_i(v)←min{local_min_i(v),lowpt(w)};
comp_local(w);
[(2)computing local_high_m(v)]
  for every back edge <w,x>
                          incident upon w do begin
    q←path_number(x);
    if j≠q then
      if local_high_q(x)<joint(q) then
        local_high_q(x)←joint(q)
    else [j=q]
      if local_high_q(x)<w then local_high_q(x)←w
  end end
end;
```

Let <v,w>$\in$ A(T), v<w, i=path_number(v), j=path_number(w), i<j, v=path_head(j). Then, clearly,

$\text{local\_min}_j(\text{path\_head}(j))=\text{path\_head}(j)$, $1\le j\le n$.

Hence computing them is not incorporated in procedure comp_local(v). We will show that, for v$\neq$path_head(i), the procedure correctly computes $\text{local\_min}_i(v)$. If $B_i(v)=\{v\}$ then $\text{local\_min}_i(v)=v$ ($=\min B_i(v)$) and this is set in the initialization. Therefore $\text{local\_min}_i(v)$ is correctly computed for any v with $B_i(v)=\{v\}$. For any v with $|B_i(v)|\ge 2$, we can prove the following proposition by induction on the number k($\ge$1) of executions of Step (1) in procedure comp_local(v).

Proposition 2. Let $\text{lm}^{(k-1)}$ denote the value of $\text{local\_min}_i(v)$ just before the k-th execution. Let

$B_i^{(k)}(v)=\{v\}\cup\{v'|$there is a back-path P<v,v'> of v with respect to PB_i such that P<v,v'> begins with a visited edge <v,w'>$\notin A(PB_i)\}$.

Then we can prove that

$$\text{lm}^{(k-1)}=\begin{cases} A_i & \text{if } \min B_i^{(k-1)}(u)<s_i, \\ \min B_i^{(k-1)}(u) & \text{otherwise.}\end{cases}$$

(The proof is omitted.) Thus it is shown that $\text{local\_min}_i(v)$ is correctly computed for all v$\in$V and all PB_i.

Next, we consider local_high_m. Clearly

$\text{local\_high}_j(\text{path\_head}(j))=\text{path\_tail}(j)$, $1\le i\le n$.

Hence computing them is not incorporated in the procedure. Let q=path_number(x). We will show that $\text{local\_high}_q(x)$ is correctly computed for any x$\neq$path_head(q). Let

$C(x)=\{v|<v,w>\in BA(T)\}$.

If $C(x)=\phi$ then $\text{local\_high}_q(x)=x$: this is set in the initialization and is kept unchanged. Suppose that $C(x)\neq\phi$ in the following. We will prove the proposition by induction on the number k($\ge$1) of edges in C(x) that are visited by the procedure.

Proposition 3. Let $\text{lh}^{(k-1)}$ denote the value of $\text{local\_high}_q(x)$ just before the k-th visit. Let $C^{(k-1)}(x)\subseteq C(x)$ denote the set of the first visited edge,...,up to the (k-1)-th visited edge, and

$D^{(k-1)}(x)=\{x\}\cup\{u\in V(P_T<x,t_q>)|$there is a back-path $P_G<u,x>$ of u with respect to PB_q such that the last edge of $P_G<u,x>$ is <u',x>$\in C^{(k-1)}(x)$.

Then

$\text{lh}^{(k-1)}=\max D^{(k-1)}(x)$.

(The proof is omitted.) Thus $\text{local\_high}_q(x)$ for all x and all PB_q are correctly computed by repeating comp_local(v).

The discussion so far proves the following lemma.

Lemma 3. A path-partition can be obtained in $O(|V|+|E|)$ time by using path_partition(v). Computing $\text{local\_min}_i(u)$ and $\text{local\_high}_i(u)$ for all u$\in$V and all PB_i can be done in $O(|V|+|E|)$ time by means of comp_local(u).◊

5. Type 2 pairs

Let {(v,w),(x,y)} be any pair of edges such that <v,w>,<x,y>$\in$A(T) and v<w$\le$x<y. This pair is fixed in this section. First we define an out-edge and an in-edge. A back edge <u,u'> is called an *out-edge* or an *in-edge* of G (with respect to <v,w> and <x,y>) if it satisfies (1) or (2), respectively:

(1)u'$\in V(P_T<r,v>)$ if u$\in$V(T<w>)-V(T<y>),

(2)u'$\in V(P_T<w,x>)$ if u$\in$V(T<y>).

Since {<v,w>,<x,y>} is fixed, we omit "with respect to <v,w> and <x,y>" unless any confusion arises. Detecting type 2 pairs is based on the following lemma.

Lemma 4. {(v,w),(x,y)} is a type 2 pair if and only if BA(T) contains neither an out-edge nor an in-edge of G. (See Fig. 7.)◊

Let $\{PB_1,\ldots,PB_n\}$ be any fixed path-partition. Detection of type 2 pairs can be restricted to edges of each member PB_i by the following lemma.

Lemma 5. If {(v,w),(x,y)} is a type 2 pair then {<v,w>,<x,y>} is included in some member PB_i.◊

Based on the above path-partition, we can define directed subgraphs $G_1,\ldots,G_n$ with $V(G_i)=V(PB_i)$, $1\le i\le n$, such that {(v,w),(x,y)} is a type 2 pair of G if and only if the pair is a type 2 pair of some G_i. For each back-path P<u,u'> of $u\in V(PB_i)$, let $u''\in V(PB_i)$ be defined by

$$u'=\begin{cases}u' & \text{if } u'\in V_i,\\ s_i & \text{otherwise.}\end{cases}$$

Each edge <u,u''> is called the *shortcut* of P<u,u'>. Each $G_i=(V_i,A_i)$, $1\le i\le n$, is defined by $V_i=V(PB_i)$ and $A_i=A_i'\cup A_i''$, where

$A_i'=A(PB_i)$ and

$A_i''=\{<u,u''>\mid <u,u''>$ is the shortcut of a back-path P<u,u'> of $u\in V_i\}$.

Let E_i, E_i' and E_i'' devote the set undirected edges corresponding A_i, A_i' and A_i'', respectively. For $\{(v,w),(x,y)\}\subseteq E_i'$, an out-edge or an in-edge of G_i is similarly defined by replacing G by G_i.

We can prove the following lemma.

Lemma 6. {(v,w),(x,y)} is a type 2 pair of G if and only if there exists some G_i such that $\{<v,w>,<x,y>\}\subseteq A_i'$ and A_i'' contains neither an out-edge nor an in-edge of G_i.◊

It is easy to see that

$$local\_min_i(u)=\begin{cases}\min\{\{u'\mid <u,u'>\in A_i''\}\cup\{u\} & \text{if } u\neq s_i,\\ s_i & \text{otherwise,}\end{cases}$$

$$local\_high_i(u)=\begin{cases}\max\{\{u'\mid <u',u>\in A_i''\}\cup\{u\} & \text{if } u\neq s_i,\\ t_i & \text{otherwise.}\end{cases}$$

for each $u\in V(P_T<w,x>)$, where $<v,w>,<x,y>\in A_i$. Hence we can use $local\_min_i(u)$ and $local\_high_i(u)$ in finding type 2 pairs, instead of actually constructing G_i.

We formally define a desired edge set KE(T) which is going to be found: KE(T) is a minimal set in which any type 2 pair is included. First we define an edge set GEN_i for each i, $1\le i\le n$. For each $<x,y>\in A_i'$ with x<y, define an edge set $E_{xy}\subseteq E_i$ by

$$E_{xy}=\begin{cases}\{(x,y)\}\cup\{(v,w)\mid \{(v,w),(x,y)\} \text{ is a type 2 pair of G and } v<w\le x<y\} \\ \qquad \text{if a type 2 pair } \{(v,w),(x,y)\} \text{ exists,}\\ \phi \qquad \text{otherwise.}\end{cases}$$

Delete all empty sets among $|A(T)|$ sets E_{xy}, $<x,y>\in A(T)$. If there is no nonempty set left then $GEN_i\leftarrow\phi$. If there is at least one nonempty set then let $E^{(1)},\ldots,E^{(k(i))}$ be the set of all maximal sets (with respect to set inclusion) among them. Let $(x_j,y_j)\in E^{(j)}$ with $x_j<y_j$ be the edge such that $y_j=\max\{v\mid (u,v)\in E^{(j)}$ and $<u,v>\in A_i'\}$. The edge (x_j,y_j) is called the *generator* of $E^{(j)}$, and let

$GEN_i=\{(x_j,y_j)\mid 1\le j\le k(i)\}$.

Let

$$KP^{(j)}=\{\{(v,w),(x_j,y_j)\}\mid (v,w)\in E^{(j)}-\{(x_j,y_j)\}\},$$

$$KA^{(j)}=\{\{(v,w),(v',w')\}\mid (v,w),(v',w')\in E^{(j)},\ (v,w)\neq(v',w')\}$$

for each $E^{(j)}$, and define

$KE_i=E^{(1)}\cup\ldots\cup E^{(k(i))}$, $KP_i=KP^{(1)}\cup\ldots\cup KP^{(k(i))}$,

$KA_i=KA^{(1)}\cup\ldots\cup KA^{(k(i))}$,

$KE(T)=KE_1\cup\ldots\cup KE_n$, $KP(T)=KP_1\cup\ldots\cup KP_n$,

$KA(T)=KA_1\cup\ldots\cup KA_n$.

Remark 1. In each G_i, $1\le i\le n$, we have $E(s)\cap E(t)=\phi$, $1\le s<t\le k(i)$.◊

The following procedure type_2(v) find KP(T) in $O(|V|+|E|)$ time. A pair {<x,y>, [p,q]} with $p\le q$ denotes an element, called a *candidate*, to be added into a stack (STACK) or existing on the top of STACK. It means that if there is any edge <v,w> such that {(v,w),(x,y)} is a type 2 pair then $p\le v<w\le q$. <x,y> and [p,q] are called a *candidate-edge* and a *candidate-path* (denoted by P<p,q>), respectively. The vertex p (q, respectively) is called the starting (ending) vertex of P<p,q>.

First the following initialization is done:

STACK←{<r,r>,[r,r]}, where r is the starting vertex.

Then the following procedure is repeated.

```
procedure type_2(v)
[During DFS for T, execute (1) and (2).]
begin
for every unvisited son w of v do
  [(1)when DFS moves forward from v to a son w of v
  in T]
  if path_number(v)≠path_number(w) then
    STACK←{<v,v>,[v,v]};
    [Addition of an element into STACK. {<v,v>,[v,v]}
    (including the case with v=r) is called a dummy
    candidate, and v is equal to the starting vertex s_i
    of PB_i such that w∈V(PB_i)]
  type_2(w);
  [(2)When DFS moves backward from w to v,
    do the following {<x,y>,[p,q]} is the top of STACK.]
  i←path_number(w);
  (2.1)while (top of STACK is not a dummy candidate)
                          and (y≤local_high_i(w)) do
      STACK←STACK-{<x,y>,[p,q]};
      [Deletion of the top of STACK: the edge
      (local_high_i(w),w) is an in-edge of G_i with
      respect to (v',w')∈E(P<p,q>) and (x,y), where
      {<v',w'>,<x,y>}⊆A(T).]
  (2.2)while p>local_min_i(w) do
      STACK←STACK-{<x,y>,[p,q]};
      [The edge (w,local_min_i(w)) is an out-edge of G_i
      with respect to (v',w')∈E(P<p,q>) and (x,y).]
  (2.3)if q>local_min_i(w) then
      if p<local_min_i(w) then
        change top of STACK from {<x,y>, [p,q]}
                          to {<x,y>,[p,local_min_i(w)]};
        [The edge (w,local_min_i(w)) is an out-edge of
        G_i with respect to (v',w')∈E(P<local_min_i(w),
        q>) and (x,y).]
      else STACK←STACK-{<x,y>,[p,q]};
        [There is no (v',w') such that {(v',w'),(x,y)} is
        a type 2 pair.]
  (2.4)if w=q then
          [a type 2 pair {(v,w),(x,y)} is found] begin
      output ({(v,w),(x,y)},path_number(w));
      if p<v then
        change top of STACK from {<x,y>,[p,q]}
                                  to {<x,y>,[p,v]};
      else STACK←STACK-{<x,y>,[p,q]} end;
  (2,5)if (local_min_i(v)>q)
            and (v≠path_head(i)) then
       STACK←{<v,w>,[q,local_min_i(v)]};
  (2,6)if v=path_head(i) then begin
      while {<x,y>,[p,q]}≠{{<v,v>, [v,v]} do
        STACK←STACK-{<x,y>,[p,q]};
        [Searching PB_i which contains w as an inner
        vertex is finished.]
      STACK←STACK-{<v,v>,[v,v]} end
end;
```

Remark 2.
(1)$p \leq q$ always holds in procedure type_2(v).
(2)$q \leq local\_min_i(w)$ always holds at Step (2.5) of procedure type_2(v).◊

The following Lemmas 7 through 11 show that procedure type_2 correctly finds KP(T) in $O(|V|+|E|)$ time. The next lemma is useful in proving the other lemmas.

Lemma 7. Suppose that procedure type_2(a) moves on an edge $<a,b> \in A_i'$ backward from b to a, and consider the stage just before Step (2.5) of procedure type_2(a). If top of STACK is $\{<c,d>,[p,q]\}$ at this stage then the following (1)-(6) hold.

(1) $a \geq q$.
(2) (i) If $a>q$ then there is $<f,q> \in A_i''$ with $f \geq b$.
(ii) If $a=q$ then $(a,b) \notin GEN_i$.
(3) p is not less than the ending vertex of any candidate path currently existing under top of STACK.
(4) There is $<g,p> \in A_i''$ with $g \geq d$.
(5) If $\{<c,d>,[p,q]\}$ is not a dummy candidate then $p<q \leq c$.
(6) If $\{<c,d>,[p,q]\}$ is not a dummy candidate then there exists no edge $<z,z'> \in A_i''$ satisfying (i) or (ii).
(i) $b \leq z \leq c$ and $z' \leq p$; (ii) $d \leq z$ and $b \leq z' \leq c$.

(The proof is given in Appendices.)

The following Lemmas 8 and 9 assure that any type 2 pair $\{(v,w),(x,y)\}$ with $(x,y) \in GEN_i$ and $(v,w) \in E_{xy}$ will be kept in STACK.

Lemma 8. Let $\{(x,y),(v,w)\}$ be any type 2 pair such that $(x,y) \in GEN_i$ and $(v,w) \in \{E_{xy}-\{(x,y)\}\}$. Then there are vertices $p,q \in V_i$, $p<q$, such that the subpath $P<p,q>$ of PB_i contains $<v,w>$ and a candidate $\{<x,y>,[p,q]\}$ is added to STACK during the execution of procedure type_2(x).◊

Let $\{<x,y>,[p,q]\}$ be the candidate as in Lemma 8. Then the following lemma holds.

Lemma 9. Suppose that procedure type_2(v) moves on an edge $<v,w> \in A_i'$ backward from w to v. Then top of STACK is equal to $\{<x,y>,[p,w]\}$ at the stage just before Step (2.4) of the procedure.

(The proof is given in Appendices.)

The following two lemmas are useful in determining all 3-components in Section 7.

Lemma 10. Any pair $\{(v,w),(x,y)\}$ output by procedure type_2(v) is a type 2 pair.◊

Lemma 11. procedure type_2(r) correctly finds KP(T) in $O(|V|+|E|)$ time.◊

6. 3-edge-components

All type 1 pairs of G can be found in $O(|V|+|E|)$ time by Lemma 2. KE(T) can be obtained in $O(|V|+|E|)$ time by Lemma 11 and by the definitions of KP(T) and KE(T). The set KE(T) has the following property.

Lemma 12. If $\{(a,b),(c,d)\} \subseteq KA(T)$ then $\{(a,b),(c,d)\} \subseteq KE(T)$.◊

Let Ecut(G) be the set of edges contained in type 1 pairs or in KE(T). Since KP(T) determines KE(T), Lemmas 2 and 12 show that Ecut(G) can be obtained in $O(|V|+|E|)$ time. Let 3_com denote the class of all connected components of G-Ecut(G), and let R(G) be the class of all 3-components of G. Then we can prove the following lemma.

Lemma 13. R(G)=3_com.◊

Thus our main theorem follows for Lemma 2, 11 and 13.

Theorem 1. All 3-components of a given multigraph G=(V,E) can be obtained in $O(|V|+|E|)$ time.◊

Corollary 1. We can determine whether or not G is 3-edge-connected in $O(|V|+|E|)$ time.◊

Let KA'(T) be the set of all type 1 pairs of G. Then $KA(T) \cup KA'(T)$ consists of all cutpairs of G.

Corollary 2. All cutpairs of G can be found in $O(|V|^2+|E|)$ time.◊

Acknowledgements

The research of T. Watanabe is partly supported by the Grant in Aid for Sciectific Research of the Ministry of Education, Science and Culture of Japan under Grant: (A) 02302047; by the Telecommunications Advancement Foundation (TAF), Tokyo, Japan; and by CSK research Grant for Information Communication, Tokyo, Japan.

References

[1] A.V.Aho, J.E.Hopcroft and J.D.Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA (1974).
[2] K.P.Eswaran and R.E.Tarjan, Augmentation problems, SIAM J. comput., 5(1976), 653-665.
[3] S.Even, Graph Algorithms, Pitman, London (1979).
[4] F.Harary, Graph Theory, Addison-Wesley, Reading, MA (1969).
[5] J.E.Hopcroft and R.E.Tarjan, Dividing a graph into triconnected components, SIAM J. Comput., 2(1973), pp.135-158.
[6] T.C.Hu, Integer Programing and Network Flows, Addison-Wesley, Reading, MA (1969).
[7] H.Nagamochi and T.Watanabe, Computing k-Edge-Connected Components, IEICE Tech. Rep., COMP 90-94 (March 1991), pp.33-38.
[8] T.Watanabe and A.Nakamura, Edge-connectivity augmentation problems, J. Comput. and System Sci., 35(1987), pp.96-144.
[9] T.Watanabe, T.Narita and A.Nakamura, 3-Edge-connectivity augmentation problems, Proc. 1989 IEEE ISCAS (May 1989), pp.335-338.
[10] T.Watanabe and A.Nakamura, A smallest augmentation to 3-connect a graph, Discrete Applied Mathematics 28(1990), pp.183-186.
[11] M.Yamakado and T.Watanabe, A Linear-Time Augmenting Algorithm for the 3-Edge-Connectivity Augmentation Ploblem, Tech. Res. Rep. IEICE of Japan, COMP 90-57, pp.41-49 (Nov. 1990).
[12] T.Watanabe, M.Yamakado and K.Onaga, A Linear Time Augmenting Algorithm for 3-Edge-Connectivity Augmentation Problems, Proc. 1991 IEEE ISCAS(June 1991), to appear.
[13] T.Watanabe, M.Yamakado, S.Taoka and K.Onaga, 3-Edge-Connectivity Augmentation Problems with Unity Costs, Proc. of the 4th KARUIZAWA Workshop on Circuits and Systems, IEICE of Japan, pp. 92-97 (April 1991).

Appendices

Proof of Lemma 7. We prove the lemma by induction on the number k of backward movements of the procedure. If $<a,b>$ is the first edge traced backward then STACK contains only top of STACK which is a dummy candidate $\{<s_i,s_i>,[s_i,s_i]\}$ and $q=s_i$. Hence (1)-(5) of the lemma hold. Assume that the lemma holds for those edges traced up to the (k-1)-th backward movement, and let $<a,b>$ be the k-th edge traced backward. Let $<b,b'>$ be the (k-1)-th edge traced backward, and top of STACK be $\{<c',d'>,[p',q']\}$ at the stage just before Step (2.5) of procedure type_2(b). Then, by inductive hypothesis, (1)'-(6)' hold.

(1)' $b \geq q'$.
(2)' (i) If $b>q'$ then there is $<f',q'> \in A_i''$ with $f' \geq b'$. (ii) If $b=q'$ then $(b,b') \notin GEN_i$.
(3)' p' is not less than the ending vertex of any candidate path currently existing under top

of STACK.

(4)' There is $<g',p'> \in A_i$" with $g' \geq d'$.

(5)' If $\{<c',d'>,[p',q']\}$ is not a dummy candidate then $p'<q' \leq c'$.

(6)' If $\{<c',d'>,[p',q']\}$ is not a dummy candidate then there exists no edge $<u,u'> \in A_i$" satisfying (i) or (ii).

(i) $b' \leq u \leq c'$ and $u' \leq p'$; (ii) $d' \leq u$ and $b' \leq u' \leq c'$.

We will consider the course of Steps (2.5), (2,6) of procedure type_2(b) and Steps (2.1)-(2.4) of procedure type_2(a). For notational simplicity, let p* and p" (q* and q", respectively) denote the value of p' (of q') before and after each of these steps. There are two cases

(I) $a \geq q'$; (II) $a<q'$

for q' just before Step (2.1) of procedure type_2(a). (Note that after a candidate $\{<c',d'>,[p',q']\}$ is added to STACK, q' may be changed: the current q' may be different from the original one.) Since $b \neq s_j$, Step (2.6) of procedure type_2(b) cannot be executed.

(I) $a \geq q'$. If Step (2.5) is not executed then (1)-(5) of the lemma clearly holds after (2.5). Suppose that Step (2.5) is executed. Before the execution, $local\_min_i(b) \leq a$ (since if $local\_min_i(b)=b$ then q"=b, a contradiction) and $q^*<local\_min_i(b)$. After the execution, $p"=q^*, q"=local\_min_i(b)$, $a \geq q"$ and there exists $<b,q"> \in A_i$". Hence (1), (2)(i), (3), (4) and (5) hold. If a=q" then there is no edge making a type 2 pair with (a,b). That is, $(a,b) \notin GEN_i$, and (2)(ii) holds. Next, consider Steps (2.1)-(2.4) of procedure type_2(a). Let $\{<c",d">,[p",q"]\}$ be top of STACK after each of there steps. After (2.1), there is no edge $<z,z'> \in A_i$" with $d" \leq z$ and $b \leq z' \leq c"$. Consider (2.2) and (2.3) together then, after (2.3), there is no edge $<z,z'> \in A_i$" with $b \leq z' \leq c"$ and $z' \leq p"$. Hence (6) holds just before (2.4). If none of (2.1)-(2.3) is executed then $a \geq q"$ and Step (2.4) cannot be executed. Suppose that any one of (2.1)-(2.3) is executed. Then $a \geq q^* > q"$, and (1) holds. Use inductive hypothesis if (2.1) or (2.2) is executed, or use the definition if (2.3) is executed. Then (3), (4) and (5) hold, and there is $<f,q"> \in A_i$" with $f \geq b$. Hence (2)(i) and (ii) hold. (2.4) is not executed since $a \geq q"$. Thus (6) holds and it is shown that lemma holds for $<a,b>$ if $a \geq q'$.

(II) $a<q'$. Then $q'=b$. For the value q^* of q' just before Step (2.5) of procedure type_2(b), there are three cases:

(i) $a \geq q^*$ and $local\_min_i(b)=b$,

(ii) $a<q^*$ and $loca\_min_i(b)=b$,

(iii) $a<q^*$ and $local\_min_i(b)<b$.

If (i) holds then (2.5) is executed, while if (ii) or (iii) does then it is not. (2), (3), (4) and (5) of the lemma hold after (2.5), where (1) also holds in (ii) and (iii); (1) does not hold in (i).

First suppose that $local\_min_i(b)=b$. Note that if (2.1) is executed then we have $b=q^*>q"$ and, therefore, neither (2.2) nor (2.3) is executed. If (2.1) is not executed then only (2.4) will be executed and we have q"=a after (2.4). That is, (1), (3), (4) and (5) hold and a=q'. $\{(a,b),(c',d')\}$ is a type 2 pair, where $\{<c',d'>,[p,q]\}$=top of STACK. $(a,b) \notin GEN_i$, since $c' \geq b$. Hence (2)(i) and (ii) hold. If (2.1) is executed then $q"<b$ (or $q" \leq a$) after (2.1), and (2.4) cannot be executed. That is, (1), (3), (4) and (5) hold. If $q"<a$ then (2)(i) holds by inductive hypothesis, and if q"=a then, similarly to above, we can show that $(a,b) \notin GEN_i$.

Next suppose that $local\_min_i(b)<b$. Then Step (2.5) is not executed, and (1)-(5) of the lemma hold just before (2.1). If (2.1) or (2.2) is executed then we can show that (1)-(5) hold by using inductive hypothesis. Suppose that (2.3) is executed. Then, after (2.3), we have $q" \leq a$ and there is $<f,q"> \in A_i$" with $f \geq b$. If a=q" then no edge makes a type 2 pair with (a,b), and $(a,b) \notin GEN_i$. That is, (1) and (2)(i), (ii) hold. (3)-(5) clearly hold. Similarly to (I) it is shown that (6) holds even after (2.4). Thus it is shown that the lemma holds for $<a,b>$ if $a<q'$.

Proof of Lemma 9.. Let $<a,b>$ be any edge of A_i' with $x>a \geq v$, and suppose that procedure type_2(a) traces $<a,b>$ backward. We consider the course of (2.1)-(2.4) of the procedure. Let $\{<x',y'>,[p',q']\}$ be top of STACK at the stage just before (2.1). If this candidate is a dummy one then none of (2.1)-(2.4) is executed. Hence it suffices to consider the case where this candidate is not a dummy one.

We first show that a candidate of the form $\{<x,y>, [p",q"]\}$ exists in STACK even after the execution of procedure type_2(a) with $a \geq v$, where $p" \leq v<w \leq q"$. Suppose that $\{<x',y'>,[p',q']\}$ is deleted from STACK in one of (2.1)-(2.4). Let $<v',w'>$ be an edge of A_i' with $p' \leq v<'w' \leq q'$. For (2.1) (for (2.2), respectively) there is an in-edge $<local\_high_i(b),b>$ (an out-edge $<b,local\_min_i(b)>$) of G_i with respect to $<v',w'>$ and $<x',y'>$. For (2.3), we have $p'=local\_min_i(b)$ before the execution, and p'=q' after the execution of $q' \leftarrow local\_min_i(b)$ in (2.3). Hence $A(P<p',q'>)$ has no edge making a type 2 pair with (x',y'). Similarly for (2.4).

Suppose that q' in top of STACK is changed in Step (2.3). Then $local\_min_i(b)<b$. We have $local\_min_i(b)< q' \leq b$ before the execution of (2.3), and this means that there is an out-edge $<b,local\_min_i(b)>$ of G_i with respect to (v',w') and (x',y'), where $local\_min_i(b) \leq v'<w \leq q'$. After the execution, we have $q'=local\_min_i(b)$ and, therefore, $<v',w'> \notin A(P<p', q'>)$.

Hence, for a type 2 pair $\{(v,w),(x,y)\}$, there is a candidate of the form $\{<x,y>,[p",q"]\}$ kept in STACK, and $p" \leq v<w \leq q"$ after the execution of (2.1), (2.2), (2.3) or deletion in (2.4) of procedure type_2(a) with $x>a \geq v$. Furthermore we have $p' \leq v<w \leq q'$ even if q' in top of STACK is changed in (2.4) of procedure type_2(a) wuth $x>a \geq w$. Note that p" of candidates in STACK is kept unchanged.

Next we consider the stage just before Step (2.4) of procedure type_2(v). Clearly there is a candidate of the form $\{<x,y>,[p",q"]\}$ in STACK, where $p" \leq v<w \leq q"$. Suppose that $\{<x',y'>,[p',q']\}$ is top of STACK and $<x',y'> \neq <x,y>$.

Clearly $y' \leq x$. By Lemma 7 (3)-(6), $q" \leq p'<q' \leq x'<y \leq x$ and there is $<g,p'> \in A_i$" with $y' \leq q \leq x$. Hence $\{<x',y'>, [p',q']\}$ cannot exist in STACK after Step (2.1) of procedure type_2(u) for u such that $<u,p'> \in A_i$'. This is a contradiction.

Thus it is shown that $<x,y>$ is contained in top of STACK. Clearly p=p". Let q^* denote the ending vertex of a candidate path in top of STACK for notational simplicity. Just before the execution of Step (2.5) of procedure type_2(w), we have $w \geq q^*$ by Lemma 7 (1). Clearly $w \geq q^*$ even after the execution of (2.5). Consider the execution of (2.1)-(2.3) of procedure type_2(v). We have $w>q^*$ after executing any one of them. This means that none of (2.1)-(2.3) is executed, since we must have $w \leq q^*$ just before Step (2.4) of this procedure. Hence $w=q^*$, and the lemma follows.

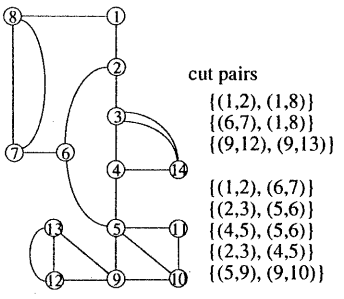

Fig. 1. A multigraph G and all the cutpairs.

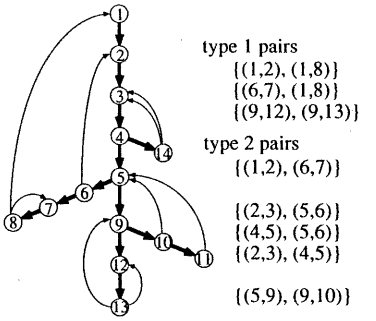

Fig. 2. A DFS-tree T and the back forest of $\overline{G}$ defined from G shown in Fig. 1, where bold directed lines denote tree edges and fine directed ones are back edges.

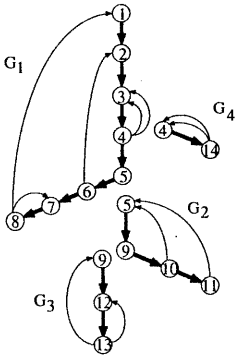

Fig. 3. A path-partition $\{PB_i | 1 \le i \le 4\}$ of T shown in Fig. 2 and the graphs G_i defined from PB_i, $1 \le i \le 4$, where each PB_i is denoted by bold directed lines.

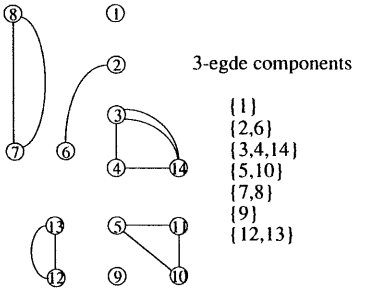

Fig. 4. All 3-components of G shown in Fig. 1.

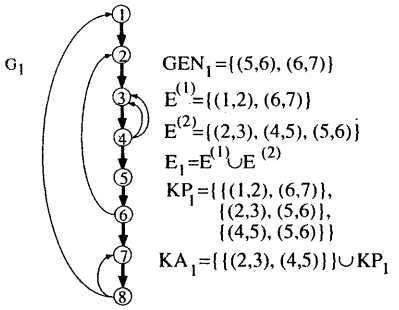

Fig. 5. Finding a desired set E_1 that includes all type 2 pairs of G_1.

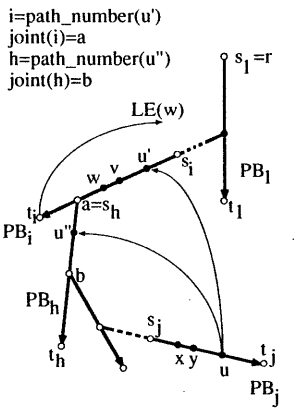

Fig. 6. A schematic explanation of the array *joint*. This also shows the reason why existence of of LE(w) prevents a pair of edges, one on PB_i and the other on PB_j ($i \neq j$), to be a type 2 pair.

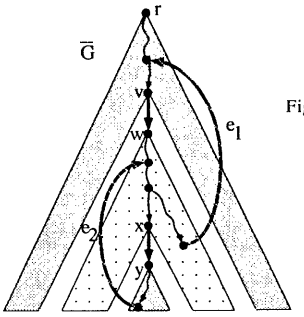

Fig. 7. A schematic explanation of an out-edge e_1 and an in-edge e_2.