Descompunerea unui graf in componente triconexe
Algoritmul - J.E. Hopcroft si R.E. Tarjan

Rezumat. Algoritmul de descompunere a unui graf in componente triconexe este prezentat in cele ce urmeaza. Algoritmul are complexitate O(V+E).
Cuvinte cheie: Puncte de articulatie, conectivitate, parcurgere DF, graf, separabilitate, separare, triconectivitate.

Introducere. Proprietatile de conexitate a grafurilor formeaza o parte importanta in teoria grafurilor. Algoritmii eficienti de determinare a unora din aceste proprietati, sunt in aceeasi masura interesanti si utili in multe aplicatii. Acest articol prezinta un algoritm de descompunere a unu graf in componente triconexe. Acest algoritm este util in:

- Analiza circuitelor electrice
- Verificarea proprietatii de planaritate a unui graf
- Verificarea proprietatii de isomorfism intre doua grafuri.

Un algoritm privind planaritatea unui graf poate fi folosit in alcatuirea sau reprezentarea circuitelor integrate.

O tehnica care folosita pentru rezolvarea problemelor legate de conectivitatea grafurilor este parcurgerea in adancime. Parcurgerea DF(depth-first) este aplicata pentru obtinerea unor algoritmi eficienti de determinare:

- a componentelor biconexe a unui graf neoriantat,
- a componentelor tare conexe a unui digraf.
- unui algoritm de testare a planaritatii unui graf, etc.

Aceelasi algoritm de parcurgere DF va fi utilizat pentru determinarea componentelor triconexe.
Articolul de fata cuprinde patru sectiuni. Prima dintre ele prezinta definitiile si lemele din teoria grafurilor, necesare in prezentarea algoritmului. A doua sectiune, descrie in mod intuitiv algoritmul de descompunere a unui graf in componente triconexe. A treia sectiune descrie calcule preliminare si teste pentru determinarea perechilor separabile din graf. In ultima sectiune se prezinta algoritmul componentelor triconexe, inclusiv demonstratiile de corectitudine si aproximarile referitoare la complixitatea in timp si in spatiu.

1. Grafuri, conexitate si parcurgerea in adancime(depth-first).

Un graf $G=(\mathcal{V},\mathcal{E})$ este format dintr-o multime $\mathcal{V}$ continand V varfuri si o multime $\mathcal{E}$ continand E muchii. Daca elementele sunt perechi ordonate (v,w) de noduri distincte –arce-, atunci graful este orientat (digraf); v este considerat *tail (*extremitate initiala*)* iar w *head (*extremitate finala*)*. Daca perechea (v,w) este neordonata, graful este de asemenea neordonat. Daca $\mathcal{E}$ este *multiset* (adica un arc se poate gasi de mai multe ori in multime) spunem ca G este multigraf. Daca $\mathcal{E}'$ este o multime de arce a lui G, $\mathcal{V}(\mathcal{E}')$ este un set de noduri incidente in una sau mai multe arce din $\mathcal{E}'$. Daca $\mathcal{S}$ este o multime de varfuri a lui G, $\mathcal{E}(\mathcal{S})$ este un set de arce incidente in cel putin un nod din $\mathcal{S}$.

Daca G este un multigraf, un drum $p:v\Rightarrow w$ in G este o secventa de varfuri si arce care duce de la v la w. Drumul este elementar(simplu) daca toate varfurile sunt distincte. Un drum $p:v\Rightarrow w$ este circuit daca toate arcele sunt distincte si singurul varf care se regaseste de exact doua ori in p

este *v*. Varianta neorientata a unui multigraf orientat se obtine prin convertirea fiecarui arc dintr-un multigraf orientat intr-o muchie.

Un multigraf neorientat este conex daca pentru orice pereche de varfuri exita un lant care le uneste. Daca $G=(V,E)$ si $G'=(V',E')$ sunt doua multigrafuri astfel incat $V' \subseteq V$ si $E' \subseteq E$ atunci G' este un subgraf a lui G. Un multigraf avand exact doua varfuri (v,w) si una sau mai multe muchii (v,w) se numeste „legatura" *bond*.

Un arbore cu radacina T este un digraf, a carui versiune neorientata este conexa, avand un varf numit radacina care nu este extremitate finala pentru nici un arc si toate celelalte varfuri reprezinta extremitate finala pentru exact un arc. Relatia „ (v,w) este un arc in T " este notata $v \rightarrow w$. Relatia „este un drum de la v la w in T „ este notata $v \overset{*}{\rightarrow} w$. Multimea descendentilor a varfului v este notata cu $D(v)$. Fiecare varf este un ascendent si un descendent al sau. Daca G este un multigraf orientat, un arbore T este un arbore de acoperire a lui G, daca T este un subgraf a lui G care contine toate varfurile lui G.

Consideram P a fi un multigraf *orientat* continand doua seturi disjuncte de arce, notate $v \rightarrow w$ si $v - \rightarrow w$. Presupunem ca P satisface urmatoarele proprietati:

1. Subgraful T care contine arcele $v \rightarrow w$ este un spanning tree al lui P
2. Daca $v - \rightarrow w$ atunci $w \overset{*}{\rightarrow} v$. Asta inseamna ca fiecare arc care nu apartine arborelui de acoperire T a lui P, conecteaza un varf cu unul din ascendentii sai din T.

P se numeste „*palm tree"*. Arcele $v - \rightarrow w$ sunt numite „*fronds"* ale lui P.

Un multigraf conex G este biconex, daca pentru orice triplet de varfuri distincte, v, w si q din V, exista un drum $p:v \Rightarrow w$, astfel incat q nu este pe acest drum. Daca este un triplet distinct v, w, q astfel incat q sa se afle pe fiecare drum $p:v \Rightarrow w$, atunci q se numeste <u>punct de articulatie</u> a lui G (separation point). Putem partitiona muchiile din G astfel incat doua muchii sunt in aceeasi submultime daca si numai daca apartin unui ciclu comun. Consideram $G_i=(V_i,E_i)$ unde E_i este o submultime de muchii din submultimea i a partitiei, si $V_i=V(E_i)$. Atunci se respecta urmatoarele:

i. Fiecare G_i este biconex
ii. Nici un G_i nu reprezinta un subgraf al unui subgraf biconex a lui G
iii. Fiecare varf a lui G care nu este punct de articulatie, se gaseste doar o singura data intre V_i si fiecare punct de articulatie se gaseste cel putin de doua ori.
iv. Pentru fiecare i , j cu $i \neq j$, $V_i \cap V_j$ contine cel mult un varf; in plus, acest varf(daca exista) este punct de articulatie.

Subgrafurile G_i ale lui G sunt numite *componente biconexe* ale lui G. Acestea sunt unice.

Consideram $\{a,b\}$ o pereche de varfuri dintr-un multigraf biconex G. Presupunem ca muchiile lui G sunt impartite in clase de echivalenta $E_1, E_2..E_n$, astfel incat doua muchii se afla se afla in aceeasi clasa daca se afla pe acelasi drum(daca exista) care nu contine varfurile {a,b} decat ca extremitati finale.

Clasele E_i se numesc clase de separatie a lui G in raport cu (a, b). Daca exista cel putin doua clase de separatie, atunci $\{a,b\}$ este o pereche separabila lui G mai putn in situatia in care (i) exista exact doua clase de separare si una dintre ele este formata din exact o muchie, sau (ii) exista exact trei clase de echivalenta, fiecare fiind formata din exact o muchie.

Daca G este un multigraf biconex, astfel incat nu exista nici o pereche de separare {a,b}, atunci graful este triconex.

Consideram $\{a,b\}$ o pereche de separare a lui G. Fie $E_1,E_2,\ldots E_n$ clasele de echivalenta a lui G in raport cu $\{a,b\}$. Fie $E' = E_1 \cup E_2 \cup \ldots \cup E_k$. Si $E'' = E_{k+1} \cup E_{k+2} \cup \ldots \cup E_n$. astfel incat $|E'| \geq 2$, $|E''| \geq 2$. Fie $G_1=((V(E'), E' \cup \{(a, b)\})$, $G_2 =(V(E''), E'' \cup \{(a, b)\})$. Grafurile G_1 si G_2 se numesc „split

graph-uri" a lui G, in raport cu $\{a,b\}$. A inlocui un multigraf G cu doua „split graph-uri" a lui G, se numeste „impartire" (splitting).

Pot fi multe modalitati de impartire a unui graf chiar si in raport cu o singura pereche $\{a,b\}$. O operatie de impartire o notam $s(a,b,i)$; i este o eticheta care identifica in mod unic operatiile de split. Noa muchie (a,b) adaugata in G_1 si G_2 se numeste *muchie virtuala*. Ele sunt etichetate pentru a putea fi identificate cu acelasi numar ca si cel al operatiei de split. O muchie virtuala (a,b) asociat operatiei $s(a,b,i)$ va fi notata (a,b,i).

Daca G este biconex atunci orice split graph a lui G este biconex.

Presupunem ca operatia de split este aplicata succesiv asupra unui multigraf G, pana cand aceasta nu mai este posibila (fiecare graf obtinut este triconex)

Grafurile obtinute astfel, se numesc „split components" a lui G, si nu sunt neaparat unice.

LEMA1. Fie $G=(V,E)$ un multigraf cu $|E|\geq 3$. Fie $G_1, G_2, \ldots G_m$ complonentele split a lui G. Atunci numarul total de muchii din $G_1, G_2, \ldots G_m$ este limitat de $3|E|-6$.

Demonstratie: Lema se demonstreaza prin inductie in functie de numarul de muchii din G. Daca G contine 3 muchii atunci nu se poate efectua o operatie split. Presupunem afirmatia adevarata pentru un graf cu n-1 muchii si faptul ca G are n muchii.Pe de alta parte presupunem ca G poatefi impartit in G' si G", unde G' are k+1 muchii si G" are n-k+1 muchii, $2 \leq k \leq n$-2. Prin inductie, numarul total de muchii din $G_1, G_2, \ldots G_m$ trebuie sa fie limitat de $3(k+1)-6+3(n-k+1)-6=3n-6$. **q.e.d**.

Pentru a obtine componente triconexe unice, trebuie sa reansamblam partial componentele „split". Presupunem ca $G_1=(V_1,E_1)$ si $G_2=(V_2,E_2)$ sunt doua componente „split", amandoua continand o muchiie virtuala (a,b,i). Fie

$$G=(V_1 \cup V_2\,,\,(E_1-\{(a,b,i)\}) \cup (E_2-\{(a,b,i)\}))$$

Atunci G este numit „merge graph" a lui G_1 si G_2. Operatia „merge" va fi notata: $m(a,b,i)$. Ea este operatia inversa operatiei „split". Daca vom efectua un numar suficient de operatii „merge" asupra componentelor „split" ale multigrafului, vom recrea multigraful original.

Componentele split ale multigrafului sunt de trei tipuri:

- Legaturi triple de forma $(\{a,b\},\{(a,b), (a,b), (a,b)\})$
- Triunghiuri de forma $(\{a,b,c\},\{(a,b),(a,c),(b,c)\})$
- Grafuri triconexe

Fie G un multigraf ale carui componente split sunt un set de legaturi triple $\mathscr{B}3$ un set de triunghiuri $\mathscr{T}$si un set de componente triconexe $\mathscr{G}$. Presupunem ca legaturile triple $\mathscr{B}3$ sunt supuse operatiei „merge" pentru a obtine un set de legaturi $\mathscr{B}$. In mod asemanator setul $\mathscr{T}$ este supus operatiei „merge" pana se obtine un set de poligoane $\mathscr{P}$. Atunci $\mathscr{B}$ U $\mathscr{P}$ U $\mathscr{G}$ este un set de componente triconexe a lui G.

Daca G este un multigraf oarecare, componentele triconexe ale componentelor biconexe ale lui G se numesc *componente triconexe* ale lui G.

LEMA 2. Componentele triconexe ale unui graf G sunt unice.

Figura 1 ilusteaza un graf biconex G avand cateva perechi de separare. Figura 2 ilustreaza componentele split ale lui G. Componentele triconexe ale lui G se formeaza prin unirea triunghiurilor (1,8,4) si a triunghiului (4,5,8).

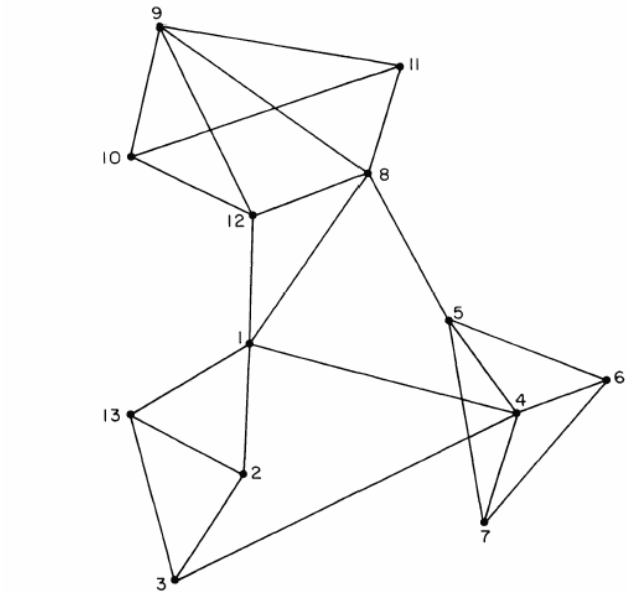


Fig. 1. *Un graful biconex G, avand perechile de separare (1,3),(1,4)(1,5),(4,5),(1,8),(4,8) si (8,12).*

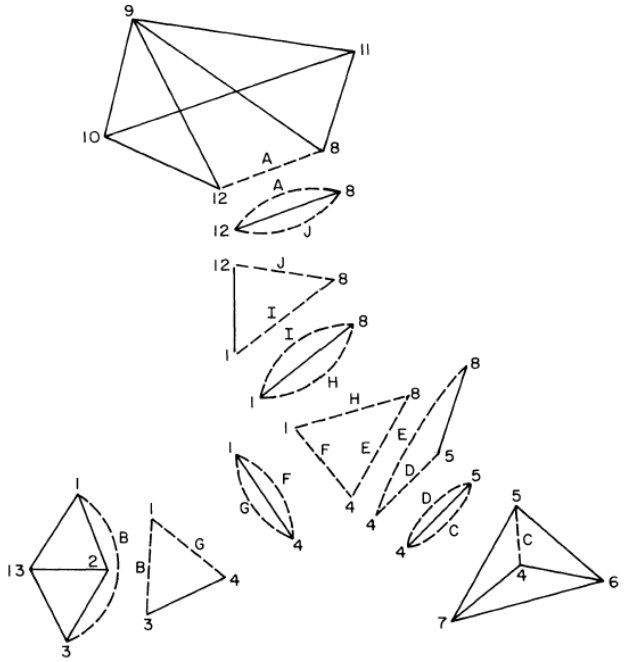


Fig. 2. *Componentele „split" ale grafului G din figura . Componentele triconexe ale lui G se formeaza prin unirea triunghiurilor (1,8,4) si a triunghiului (4,5,8).*

Algoritmi pe grafuri, necesita in mare parte un mod sistematic de explorare. Vom folosi metoda parcurgerii in adancime (depth-first search). Pentru a efectua acesta parcurgere a lui *G*, pornim dintr-un nod oarecare *s* si alegem o muchie din *s* spre a fi traversata. Parcurgerea ne conduce intr-un nou varf. Continuand in acest fel, la fiecare pas vom alege muchia neexplorata, care porneste din cel mai recent varf ce are inca muchii neexplorate. Daca *G* este conex, fiecare muchie va fi traversata o singura data.

Daca *G* este neorientat, o parcurgere a lui *G* va impune o directie pe fiecare muchie a lui *G*, data de sensul de traversare al ei in timpul parcurgerii. Astfel graful *G* se va transforma intr-un multigraf orientat *G'*.

LEMMA 3. Fie *P* un multigraf orientat generat de parcurgerea depth-first a unui multigraf neorientat conex *G*. Atunci *P* este un *Palm Tree.*

Deapth-first devine importanta si datorita structurii simple a drumurilor din „palm tree". In implementarea urmatoare a DF, s-a folosit o reprezentarea a multigrafului cu liste de adiacenta. Astfel, pentru fiecare varf *v* din multigraf, lista *A*(*v*) contine toate muchiile (*v*,*w*) din *G*. Daca *G* este neorientat, fiecare (*v*,*w*) apare memorata de doua ori.
In continuare este prezentata implementarea recursiva a parcurgerii DF. Ordinea de vizitare a varfurilor depinde de ordinea muchiilor in listele de adiacenta. Subprogramul urmator, numeroteaza varfurile de la 1 la *V* in ordinea de vizitare din *DF*. Este usor de vazut ca varfurile sunt numerotate in asa fel incat *NUMBER*(*v*)<*NUMBER*(*w*) daca $v \rightarrow w$, in arborele de acoperire.

Procedure 1

```
begin comment routine for depth-first search of a multigraph G represented by
        adjacency lists A(v). Variable n denotes the last number assigned to a
        vertex;
    integer n;
    procedure DFS (v, u); begin comment vertex u is the father of vertex v in the
            spanning tree being constructed. The graph to be searched is
            represented by a set of adjacency lists A(v);
        n := NUMBER (v) := n + 1;
    a: comment dummy statement;
        for w ∈ A(v) do begin
            if NUMBER (w) = 0 then begin
                    comment w is a new vertex;
                    mark (v, w) as a tree arc;
                    DFS (w, v);
                b: comment dummy statement;
            end
            else if (NUMBER (w) < NUMBER (v)) and ((w ≠ u) or ¬ FLAG (v))
                then begin
                    comment the test is necessary to avoid exploring an edge
                        in both directions. FLAG (v) becomes false when the
                        entry in A(v) corresponding to tree arc (u, v) is
                        examined;
                    mark (v, w) as a frond;
                c: comment dummy statement;
            end;
            if w = u then FLAG (v) = false;
        end;
    end;
    n := 0;
    for i := 1 until V do begin
        NUMBER (i) := 0;
        FLAG (i) := true;
    end;
    comment the search starts at vertex s;
    DFS (s, 0);
end;
```

Porțiunile marcate cu „dummy statement” – a ,b ,c vor fi inlocuite cand DFS va fi folosit pentru calcularea altor informatii despre graf. Figura 3 reprezinta un arbore creat prin aplicarea algoritmului DFS asupra grafului din Fig. 1.

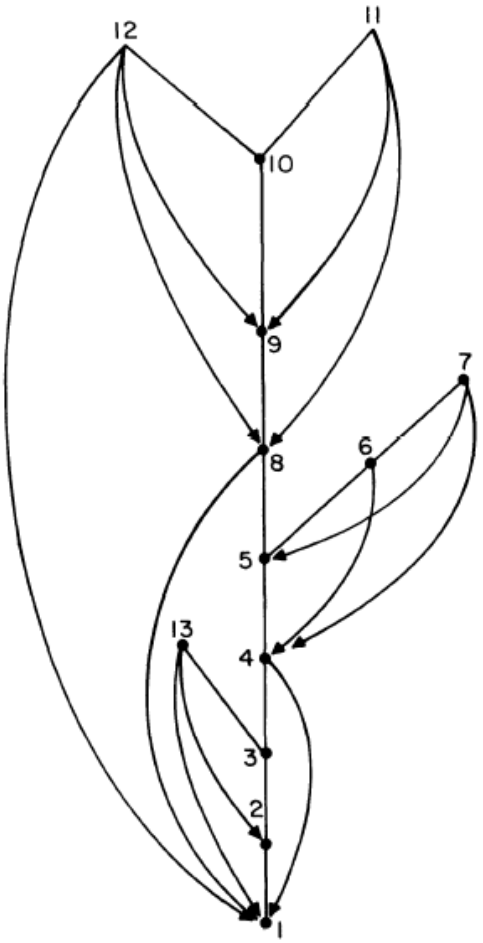


Fig. 3. Arbore rezultat din parcurgerea in adancime a grafului ***G*** din Fig. 1.

2. Conturarea unui algoritm pentru triconectivitate.

Acest algoritm schiteaza ideile din spatele algoritmului de triconectivitate. Sectiunile urmatoare trateaza in detaliu aceste componente. Algoritmul se bazeaza pe o ideea a lui Auslander, Parter si Goldstein ([18], [19]) pentru verificarea coplanaritatii grafurilor. Ideea lui Auslander, Parter si Goldstein de refera la un algoritm de complexitate $O(V)$ pentru verificarea coplanaritatii, realizat prin utilizarea parcurgerii in adancime pentru a determina ordinea operatiilor. Aceeasi idee ofera o complexitate $O(V + E)$ algoritmului de gasire a elementelor triconexe.

Fie ***G*** un multigraf biconex oarecare. Presupunem un ciclu *c* in ***G***. Cand ciclul este sters din ***G*** se obtin anumite elemente conexe. Acestea se numesc segmente. Auslander si Parter [18] demonstreaza ca un graf este coplanar daca si numai daca :

(i) orice subgraf al lui ***G*** format din *c* plus un singur segment este planar.

(ii) Segmetele pot fi combinate in mod consistent pentru a obtine o desfasurare planara a intregului graf.

Un algoritm eficient de coplanaritate poate fi creat pe baza acestui rezultat. Un rezultat similar se refera la perechile de separare ale lui ***G***, cum ar fi Lema urmatoare:

LEMA 4. Fie ***G*** un graf biconex, iar *c* un ciclu in ***G***. Fie $S_1, \ldots, S_n$ subgrafuri ale lui $G - c$ astfel incat e_1 si e_2 sunt muchii ale lui S_i daca si numai daca exista un lant *p* in ***G*** care sa contina atat e_1 cat si e_2 si niciun varv din *c* nu se afla intre e_1 si e_2 in *p*. Subgrafurile S_i si ciclul *c* partitioneaza

muchiile lui ***G***. Fie $\{a,b\}$ o pereche de separare in ***G*** astfel incat (a,b) sa nu fie muchie multipla. Atunci se indeplinesc urmatoarele:

I. Ori **a** si b sunt ambele in c, ori a si b se afla ambele intr-un segment S_i.

II. Presupunem a si b concomitent in c. Fie p_1 si p_2 cele doua lanturi care il contin pe c si unesc a si b. Atunci se intampla una din urmatoarele:
   a. un segment S_i cu cel putin doua muchii are doar pe a si b in comun cu c, si exista un nod v care sa nu se afle in S_i ($\{a,b\}$ se numeste pereche de separare "tip 1"), sau
   b. niciun segment nu contine un varf $v \neq a,b$ in p_1 si un varf $w \neq a,b$ in p_2, iar p_1 si p_2 contin fiecare un varf in afara de a si b ($\{a,b\}$ se numeste pereche de separare "tip 2").

III. In mod contrar , orice pereche $\{a,b\}$ care satisface a. sau b. este o pereche de separare.

Aceasta lema este usor de demonstrat; o versiune mai tehnica este demonstrata in urmatoarea sectiune. Lema 4 da nastere unui algoritm recursiv eficient de gasire a elementelor impartite. Gasim un ciclu in ***G*** si determinam segmentele formate cand ciclul este sters. Verificam segmentele obtinute pentru a gasi perechi de separare aplicand algoritmul recursiv si verificam existenta perechilor de separare in ciclul dupa cazurile din Lema 4. Aplicarea recursiva a algoritmului necesita gasirea ciclurilor in subgrafuri ale lui ***G*** formate prin combinarea segmentului S_i cu ciclul initial c.

Putem optimiza acest algoritm executand operatiile in ordinea parcurgerii in adancime. Fiecare apel recursiv al algoritmului solicita gasirea unui ciclu in subgraful in care se cauta perechi de separatie. Acest ciclu consta dintr-un drum simplu de muchii care nu se aflau in ciclurile gasite anterior plus un drum simplu de muchii din cicluri anterioare. Folosim parcurgerea in adancime pentru a imparti graful in drumuri simple care pot compune astfel de cicluri. Primul ciclu c va contine o secventa de arce de arbore urmat de o frond din P, arborelui obtinut prin parcurgerea in adancime. Etichetarea varfurilor va fi in asa fel incat numerele corespunzatoare varfurilor sa fie in ordine pe parcursul ciclului. Fiecare segment fa contine fie un singur frond (v, w) sau un arc din arbore (v, w) plus un subarbore cu radacina w plus toate fronds care pleaca din subarbore. Cautarea apeleaza segmentele in ordine descrescatoare dupa eticheta lui v si le partitioneaza in drumuri simple formate din secvente de arce de arbore urmate de un frond.

Gasirea drumurilor necesita defapt doua cautari deoarece cautarea drumurilor trebuie facuta intr-o anume ordine pentru a functiona si necesita anumite calcule preliminare. Sectiunea despre gasirea perechilor de separatie descrie in detaliu procesul de gasire a drumurilor si include o versiune a Lemei 4 care descrie perechile de separare in functie de drumurile generate. Sectiunea despre gasirea componentelor impartite arata cum aceste rezultate pot fi folosite pentru determinarea componentelor impartite ale unui multigraf biconex in timp $O(V+E)$.

Pentru a determina componentele triconexe ale unui multigraf arecare, eliminam muchiile multiple impartindu-le si creand seturi de legaturi cu trei muchii. Aceasta necesita ca timp $O(V+E)$ daca este implementat corect. Apoi gasim elementele biconexe ale grafului rezultat folosind algoritmul de complexitate $O(V+E)$ descris in [5] si [6]. In continuare, componentele impartite ale fiecarei componenta biconexa sunt gasite folosind algoritmul mentionat mai sus, care va fi prezentat in detaliu in urmatoarele doua sectiuni. Aceasta ne genereaza componentele impartite ale intregului graf.

Dimensiunea totala a componentelor impartite este $O(V+E)$, conform Lemei 1. In continuare identificam setul de legaturi triple B_3 si setul de triunghiuri T. Pentru fiecare din aceste doua seturi construim un graf auxiliar S ale carui varfuri sunt elemente ale setului respectiv; doua componente impartite ale lui $S(B_3)$ si $S(T)$ corespund legaturilor si poligoanelor care sunt componente triconexe ale lui G. Gasirea acestor legaturi si poligoane necesita complexitatea $O(V+E)$.

In continuare este un pseudocod prescurtat al intregului algoritm:

PROCEDURA 2.

```
procedura triconexivitate ( G );
      A: imparte muchiile multiple ale lui G pentru a forma un set de legaturi
      triple si un graf G';
      B: gaseste componentele biconexe ale lui G';
      pentru fiecare componenta biconexa C a lui G' executa:
            C: gaseste elementele impartite ale lui C;
      D: combina legaturile triple si triunghiurile in legaturi si poligoane
      prin gasirea elementelor conexe ale grafurilor auxiliare corespunzatoare.
sfarsit;
```

Pasii A, B si D necesita fiecare timp $O(V + E)$ daca sunt implementati corect. Implementarea pasului B este descris in [5]; implementarile pasilor A si D sunt lasate ca exercitiu. Pasul cel mai dificil este C, a carui implementare este descrisa in urmatoarele doua sectiuni. Pe baza reszultatelor acestor sectiuni, intregul algoritm de triconexivitate are limite de timp si spatiu egale cu $O(V + E)$.

3. Gasirea perechilor de separare.

Fie $G = (V , E)$ un multigraf biconex cu V varfuri si E muchii. Principala problema in impartirea lui G in componente impartite sta in gasirea perechilor de separatie. Aceasta sectiune ofera un criteru, bazat pe parcurgerea in adancime, pentru identificarea perechilor de separatie intr-un multigraf. Doua cautari in adancime si cateva procesari auxiliare trebuie efectuate. Aceste procesari constituie prima parte a algoritmului de determinate a componentelor impartite si sunt subiectul prezentarii denerale care urmeaza. Definitiile pentru valorile LOWPT1, ND, etc., folosite in prezentarea generala vor fi oferite ulterior.

Pasul 1. Parcurgeti in adancime multigraful G, transformandu-; intr-un arbore *P*. Etichetati nodurile lui G in ordinea in care sunt accesate in parcurgere. Calculati LOWPT1 (v), (U,) 2 (v), ND (v) si FATHER (v) pentru fiecare varf v din *P*.

Pasul 2. Construiti o structura de adiacenta acceptabila *A* arborelui *P* ordonand muchiile in structura de adiacenta dupa valorile LOWPT1 si LOWPT2.

Pasul 3. Parcurgeti in adancime graful *P* folosind structura de adiacenta *A*. Reetichetati varfurile lui *A* de la V la 1 in ordinea ultimei lor vizitari. Partitionati muchiile in drumuri simple disjuncte . Recalculati LOWPT1 (v) si LOWPT2 (v) folosind noua etichetare a varfurilor. Calculati A1 (v) , DEGREE (v) si HIGHPT (v) pentru fiecare varf v.

Detaliile acestor calcule apar mai jos. Din pasii 1, 2 si 3 obtinem suficiente informatii pentru a determina rapid perechile de separatie din *G*. Lema 13 ofera o conditie in acest scop.

Presupunem ca graful *G* este parcurs in adancime, rezultand un arbore *P*. Fie varfurile lui *P* etichetate de la 1 la V astfel incat existenta arcului $v \rightarrow w$ din *P* implica v<w, daca identificam varfurile dupa numar. Pentru orice varf v din *P*, fie FATHER (*v*) tatal lui v din arborele de acoperire a lui *P*. Fie ND (*v*) numarul descendentilor lui v. Fie $\text{LOWPT}\ 1(v) = \min\left(\{v\} \cup \left\{w \mid v \xrightarrow{*} - \rightarrow w\right\}\right)$, adica LOWPT1 (*v*) este cel mai jos varf care poate fi ajuns din *v*, traversand zero sau mai multe arce din *P*, urmate de cel mult un frond.

Fie $\text{LOWPT}\ 2(v) = \min\left(\{v\} \cup \left(\left\{w \mid v \xrightarrow{*} - \rightarrow w\right\} - \{\text{LOWPT}\ 1(v)\}\right)\right)$, adica LOWPT2 (v) este *al doilea* cel mai jos varf care poate fi ajuns din v traversand zero sau mai multe arce din P, urmate de cel mult un frond.

LEMA 5. LOWPT1 (v) $\xrightarrow{*}$ v si LOWPT2 (v) $\xrightarrow{*}$ v in *P*.

Demonstratie. LOWPT1 (v) ≤ v prin definitie. Daca LOWPT1 (v) = v, raspunsul este evident. Daca LOWPT1 (v) < v, atunci exista un frond u – → LOWPT1 (v) astfel incat v $\xrightarrow{*}$ u. Deoarece u – → LOWPT1 (v) este frond, LOWPT1 (v) $\xrightarrow{*}$ u. Deoarece *P* este un arbore, v $\xrightarrow{*}$ u si LOWPT1 (v) $\xrightarrow{*}$ u, atunci ori v $\xrightarrow{*}$ LOWPT1 (v) ori LOWPT1 (v) $\xrightarrow{*}$ v. Dar LOWPT1

(v) < v. Deci trebuie sa fie cazul LOWPT1 (v) $\xrightarrow{*}$ v $\xrightarrow{*}$ u, ceea ce demonstreaza teorema pentru LOWPT1. Demonstratia este analog si pentru LOWPT2 (v).

LEMA 6. Presupunem ca LOWPT1 (v) si LOWPT2 (v) sunt definite relativ la o etichetare a nodurilor astfel incat v $\xrightarrow{*}$ w in *P* implica NUMBER (v) < NUMBER (w). Atunci LOWPT1 (v) si LOWPT2 (v) identifica varfuri unice independente de etichetarea folosita.
***Demonstratie*.** LOWPT1 (v) identifica intotdeauna un stramos al lui v. Mai mult, LOWPT1 (v) este stramosul cu eticheta cea mai mica a lui v cu o proprietate anume fata de arborele *P*. Devreme ce ordinea stramosilor lui v corespunde cu ordinea numarului etichetei lor, LOWPT1 (v) identifica un varf unic, indepencent de numerotarea etichetelor, adica primul stramos al lui v pe drumul 1 → v care are proprietatea dorita. (Orice numerotare corecta atribuind eticheta 1 radacinii lui *P*.) Demonstratia este analog si pentru LOWPT2 (v).

Valorile LOWPT ale lui v depind numai de valorile LOWPT ale fiilor lui v si de fronds care pleaca din v; este usor de observat ca daca varfurile sunt etichetate cu numere, atunci

LOWPT1 (v) = min (({v} U {LOWPT1 (w)|v→w} U {w|v – →w})

iar

LOWPT2 (v)= min ({v} U (({LOWPT1 (w)|v→w} U {LOWPT2 (w)|v→w}
U {w|v→w})- {LOWPT1 (v)})).

Avem de asemeni ND (v) = 1 + $\Sigma_{v\rightarrow w}$ND(w). Putem calcula valorile LOWPT, ND si FATHER pentru toate nodurile in *O*(V + E), inserand urmatoarele conditii in locul conditiilor-joker a,b,c in DFS. Etichetand cu numere varfurile in ordinea in care sunt parcurse garanteaza ca v→w implica v<w.

```
PROCEDURA 3. ( completari la DFS pentru pasul 1)
a: LOWPT1 (v) := LOWPT2 (v) := NUMBER (v);
    ND (v) := 1;
b: daca LOWPT1 (w) < LOWPT1 (v) atunci
      LOWPT2 (v) := min { LOWPT1 (v) , LOWPT2 (w) };
      LOWPT1 (v) := LOWPT1 (w);
    altfel daca LOWPT1 (w) = LOWPT1 (v) atunci
            LOWPT2 (v) := min { LOWPT2 (v) , LOWPT2 (w) };
      altfel LOWPT2 (v){ LOWPT2 (v) , LOWPT1 (w) };
    ND (v) := ND (v) + ND (w);
    FATHER (w) := v;
c: daca NUMBER (w) < LOWPT1 (v) atunci
      LOWPT2 (v) := LOWPT1 (v);
      LOWPT1 (v) := NUMBER (w);
    altfel daca NUMBER (w) > LOWPT1 (v) atunci
            LOWPT2 (v) := min { LOWPT2 (v) , NUMBER (w)};
```

Este usor de verificat ca DFS modificat conform celor mentionate mai sus va calcula LOWPT1, LOWPT2, ND si FATHER corect in *O*(V + E). (Vezi [8], [17].)
LOWPT1 poate fi utilizat pentru verificarea biconectivitatii in *G*, dua cum se descrie in [5]. Urmatoarea Lema este importanta.

LEMA 7. Daca *G* este *biconex* si v→w, LOWPT1 (w) < v doar daca v≠1, caz in care LOWPT(w) = v = 1. Deasemeni, LOWPT1 (1) = 1;
***Demonstratie*.** Vezi [5].

Fie ø o functie definita pe muchiile lui *P* cu valori in {1,2,…,2V + 1} definita de:

(i) daca *e* = v – → w , ø(*e*) = 2w+1;
(ii) daca *e* = v → w si LOWPT2 (w) < v, ø(*e*) = 2LOWPT1 (w).

(iii) daca $e = v \rightarrow w$ si LOWPT2 (w) $\geq$ v, ø(e) = 2LOWPT1 (w) + 1.

Fie *A* o structura de adiacenta pentru *P*. *A* se numeste *acceptabila* daca muchiile *e* in fiecare lista de adiacenta a lui *A* sunt sortate crescator dupa valorile lui ø(*e*).

LEMA 8. Fie *P* un arbore al unui graf biconex *G* ale carui varfuri sunt numerotate in asa fel incat $v \rightarrow w$ in *P* implica v<w. Atunci structurile de adiacenta acceptabile ale lui *P* sunt independente de etichetarea stricta a nodurilor.

Demonstratie. Daca $v \rightarrow w$ in *P*, atunci conform Lemei 5, LOWPT2 (w) este un stramos al lui w. Conform Lemei 6, LOWPT2 (w) este un varf fix independent de etichetare. Devreme ce ordinea stramosilor este independenta de etichetare, raspunsul intrebarii deca LOWPT2 (w) este mai mic dacat v este independent de etichetare. Devreme ce *G* este biconex daca $v \rightarrow w$ in *P*, atunci LOWPT1 (w) $\leq$ v conform Lema 7. Conform Lema 5, LOWPT1 (w) este un stramos al lui w. Devreme ce LOWPT1 (w) $\leq$v, LOWPT1(w) trebuie sa fie un stramos al lui v. Conforma Lemei 6, varful corespunzator lui LOWPT1 (w) este independent de etichetare. Similar, daca $v - \rightarrow w$, atunci conform Lemei 3 si definitiei arborelui, w este un stramos al lui v. Dar ordinea stramosilor lui v este identica cu ordinea numerelor lor si aceasta ordinea este independenta de etichetare. Deci, structurile de adiacenta *A* ale lui *P* sunt independente de etichetarea exacta.

In general, un arbore *P* are mai multe structuri de adiacenta acceptabile. Fiind data o etichetare satisfacatoare a varfurilor lui *P* putem construi cu usurinta o structura de adiacenta acceptabila *A* folosind radix sort cu 2V + 1 buckets. Urmatoarea procedura detaliaza algoritmul de sortare, ceea ce reprezinta pasul 2 al procesarilor. Toate varfurile sunt identificate de cate un numar. Este evident ca procedura de sortare necesita $O(V + E)$ timp.

```
      PROCEDURA 4. (construirea listelor ordonate de adiacenta)
pentru i := 1 pana la 2V + 1 executa BUCKET (i) := lista goala;
pentru (v,w) o muchie a lui G executa
      calculeaza ø( (v,w) );
      adauga (v,w) lui BUCKET ( ø(v,w) );
pentru i := 1 pana la V executa A(i) := lista goala.;
pentru i := 1 pana la 2*V + 1 executa
      pentru (v,w) є BUCKET (i) executa adauga w la sfarsitul lui A(v);
```

In pasul 3 al procesarii facem o cautare depth first in *P* utilizand structura de adiacenta acceptabila *A* rezultata de la pasul 2. Aceasta parcurgere genereaza un set de drumuri in urmatoarul mod: de fiecare data cand traversam o muchie o adaugam caii pe care o construim la momentul curent. Astfel, fiecare drum contine o secventa de arce urmate de un singur frond. Din cauza ordinii impusa de ***A***, fiecare drum determina, la cel mai de jos varf posibil drumul initial este un ciclu, iar fiecare drum cu exceptia primului este simplu si are doar avrful initiale si cel terminal in comun cu drumurile generate anterior.

Daca $p{:}s \Rightarrow f$ este un drum generat, putem crea un ciclu adaugand calea din arbore $f \xrightarrow{*}$ s lui p. Ciclurile formate astfel sunt cicluri generate de apelari recursive asupra algoritmului de triconectivitate de baza explicat in sectiunea precedenta.

Avem nevoie de informatii minime despre drumuri. Fie varfurile lui *P* numerotate astfel incat $v \xrightarrow{*}$ w sa implice v $\leq$ w. Fie A_1(v) primul varf din *A*(v). Daca $v - \rightarrow w$ este primul frond accesat in pasul 3 care se termina la w, fie HIGHPT (w) =v. Fie DEGREE (v) numarul de muchii incidente in varful v. Pasul 3 eticheteaza varfurile cu numere de la V la 1 in ordinea ultimei lor vizitari. Este clar ca etichetarea garanteaza ca v < w faca $v \xrightarrow{*}$ w. Pasul 3 calculeaza deasemeni LOWTP1 (v), HIGHPT (v), A_1(v) si DEGREE (v) conform noii etichetari. Procedura 5, bazata pe DFS va executa pasul 3 in $O(V + E)$.

PROCEDURA 5.

pasul 3: inceput comentariu metoda de generare a drumurilor intr-un arbore biconex folosind lista de adiacenta ordonata *A*(v). Varful *s* este o variabila globala reprezentand nodul

de inceput al drumului curent. *s* este initializat cu 0. Variabila m reprezinta ultimul numar asociat unui varf.

```
procedura PATHFINDER (v);
NEWNUM (v) := m – ND (v) + 1;
        for w ∈ A(v) do
            if s = 0 then begin
                s := v;
                start new path;
            end;
            add (v, w) to current path;
            if v → w then begin
                PATHFINDER (w);
Y:              m := m − 1;
        end else begin comment v − → w;
            if HIGHPT (NEWNUM (w)) = 0 then
                HIGHPT (NEWNUM(w)) :=
                    NEWNUM (v);
            output current path;
            s := 0;
        end;
            end;
            s := 0;
Z:          m := V;
            for i := 1 until V do NEWNUM (i) := HIGHPT (i) := 0;
            comment vertex 1 is the start vertex of the search;
            PATHFINDER (1);
            for all vertices V do
                compute A1 (v), DEGREE (v), LOWPT1 (v), and
                LOWPT2 (v) using the new numbering;
        end;
```

Subprogramul de la pasul 3 numara nodurile de la *V* la 1 in ordinea ultimei vizitari ale fiecaruia in timpul cautarii. Cu toate acestea, fiecarui nod *i* se atribuie de fapt cate un numar la prima vizitare, pentru a se putea efectua corect calcularea valorii HIGHPT. Pentru a se putea realiza aceasta, variabila i este initializata la inceput cu V (instructiunea Z). Valoarea lui i este scazuta cu 1 de fiecare data cand un nou nod este vizitat (instructiunea Y). Asadar cand un nod *v* este vizitat pentru prima data, i este egal cu numarul pe care vrem sa i-l atribuim lui *v minus* numarul de noduri care mai trebuie vizitate pana ce v este vizitat pentru ultima data.

Insa nodurile ce mai trebuie vizitate intre prima si ultima vizitare a lui v sunt doar descendenti valizi ai lui v. Asadar, daca atribuim numarul i – ND(*v*) + 1 lui v cand v este vizitat pentru prima data (instructiunea X), numerotarea va fi corecta. Restul calcularilor effectuate la pasul 3 sunt evidente si usor de implementat. Palm tree-ul grafului G din figura 1 este ilustrat in figura 4 impreuna cu valorile LOWPT si setul de drumuri generate la pasul 3.

Fie G un multigraf biconex asupra caruia pasii 1, 2 si 3 au fost efectuati, dandu-se un palm tree P si seturile de valori definite mai sus. Fie A, cu listele de adiacenta A(*v*), structura acceptabila de adiacenta construita la pasul 2. Nodurile din G vor fi identificate de catre numerele atribuite acestora la pasul 3. Mai e nevoie de o singura definitie. Daca $u \rightarrow v$ si *v* este primul nod din A(*u*), atunci *v* este numit primul fiu al lui *u*. (Pentru fiecare nod *v*, A1(*v*), primul fiu al lui *v* daca exista, este calculate la pasul 3).

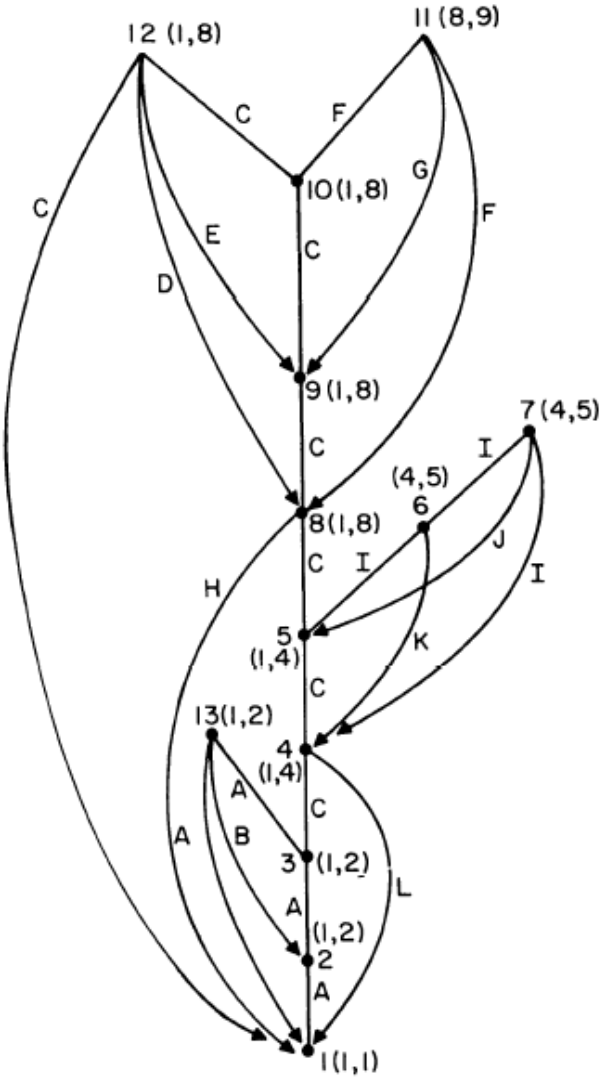


FIG. 4. *Palm tree ordonat dupa cautarea de drumuri cu valorile LOWPT1 si LOWPT2 intre paranteze*
Perechi de tipul 1: (1, 4), (1, 5), (4, 5), (1, 8), (1, 3) *Perechi de tipul 2:* (4, 8), (8, 12).
Drumuri: A:(1,2,3,13,1) B:(13,2) C:(3,4,5,8,9,10,12,1) D:(12,8) E:(12,9) F :(10,11, 8)
G:(11,9) H:(8,1) I:(5,6,7,4) J:(7,5) K:(6,4) L:(4,1)

Daca $u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_n$, si u_i este primul fiu al lui u_{i-1} pentru $1 \leq i \leq n$, atunci u_n este numit *primul descendent* al lui u_0. Secventa arcelor $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n$ este inclusa intr-un drum generat la pasul 3. Lemele de mai jos denota proprietatile necesare pentru a determina perechile de separare ale lui G.

Lema 9. *Fie A(u) lista de adiacenta a nodului u. Fie $u \rightarrow v$ si $u \rightarrow w$ arce, cu v aparand inainte de w in A(u). Atunci $u < v < w$.*

Demonstratie. Pasul 3 numara nodurile de la *V* la 1 in ordinea ultimei vizitari in cautare. Daca $u \rightarrow v$ este vizitat inainte de $u \rightarrow w$, *v* va fi vizitat ultima oara inainte ca *w* sa fie si el vizitat pentru ultima oara, si *v* va primi un numar mai mare. In mod sigur *u* va fi vizitat ultima oara dupa ce si *v* si *w* vor fi vizitate ultima oara, asadar *u* primeste cel mai mic numar dintre cele trei noduri.

Lema 10. *A este acceptabil in raport cu numerotarea data de pasul 3.*

Demonstratie. Sortarea de la pasul 2 creeaza o lista de adiacenta acceptabila pentru numerotarea originala. Din lema 9 rezulta ca $u \rightarrow v$ implica $u<v$, asadar, conform lemei 8, *A* este acceptabil pentru numerotarea noua.

Lema 11. *Daca v este un nod si D(v) este setul de descendenti ai lui v, atunci D(v) = $\{x|v \leq x < v +$* ND(*v*)$\}$. *Daca w este un prim descendent al lui v, atunci D(v) – D(w) = $\{x|v \leq x < w\}$.*

Demonstratie. Presupunem ca inversam toate listele de adiacenta A(*v*) si le folosim pentru a specifica o cautare in adancime a lui P. Nodurile vor fi vizitate prima data in ordine ascendenta de la 1 la

V, daca ele sunt identificate de catre numarul atribuit la pasul 3. Asadar descendentilor lui *v* le sunt atribuite numere consecutive de la *v* la *v* + ND(*v*) – 1. Daca *w* este un prim descendent al lui *v*, nodurilor din D(*w*) li se vor atribui numere dupa toate nodurile din D(*v*) – D(*w*). Asadar D(*v*) – D(*w*) = {x|*v* ≤ *x* < *w*}.

Lema 12. *Fie {a, b} o pereche de separare in G cu a < b. Atunci a* $\xrightarrow{*}$ *b in arborele de acoperire T al lui P.*

Demonstratie. Fiindca *a* < *b*, *a* nu poate fi un descendent al lui *b*. Presupunem ca *b* nu este un descendent de-al lui *a*. Fie E_i, pentru 1 ≤ *i* ≤ *k*, clasele de separare ale perechii {*a, b*}. Fie *S* = V – D(*a*) – D(*b*). Nodurile din *S* definesc un subarbore al lui T ce nu contine pe *a* sau pe *b*, asadar *E*(*S*) trebuie sa apartina unei clase de separare, sa zicem E_1. Fie *c* orice fiu al lui *a*. *E*(*D*(*c*)) trebuie sa apartina unei clase de separare. Dar fiindca *G* este biconvex, si *a* ≠ 1, LOWPT1 (*c*) < *a*, conform lemei 7. Asadar exista o muchie incidenta intr-un nod din *S* si intr-un nod din *D*(*c*). Asadar *E*(*D*(*c*)) ⊆ E_1. Un argument similar arata ca muchiile incidente in orice descendent de-al lui *b* sunt in E_1. Dar asta ar insemna ca E_1 = *E*, si {*a, b*} nu poate fi o pereche de separare.

Lema 13. *Presupunem a < b. Atunci {a, b} este o pereche de separare a lui G daca si numai daca* (i), (ii) sau (iii) este adevarata.

(i) *Exista noduri distincte r* ≠ *a, b si s* ≠ *a, b astfel incat b* → *r,* LOWPT1(*r*) = *a,* LOWPT2(*r*) ≥ *b, iar s nu este un descendent de-al lui r. (Perechea {a, b} este numita pereche de separare de "tipul 1". Perechile de tipul 1 pentru graful din Fig. 4 sunt* (1, 3), (1, 4), (1, 5), (4, 5) si (1, 8).)

(ii) *Exista un nod r* ≠ *b astfel incat a* → *r* $\xrightarrow{*}$ *b; b este un prim descendent al lui r* (*adica a, r si b se afla pe un drum generat comun*); *a* ≠ 1; *fiecare frond x* → *y cu r* ≤ *x < b are a* ≤ *y; si fiecare frond x* → *y cu a* < y < *b si b* → *w* $\xrightarrow{*}$ *x are* LOWPT1(*w*) ≥ *a*. ({*a, b*} *se cheama pereche de separare de "tipul 2". Perechile de tipul 2 pentru graful din Fig. 4 sunt* (4, 5) *si* (8,12)).

(iii) (*a, b*) *este o muchie multipla a lui G si G contine cel putin 4 muchii.*

Demonstratie. Contrara lemei e cel mai usor de demonstrat. Presupunem ca prechea {*a, b*} satisface (i), (ii) sau (iii). Fie E_i pentru 1 ≤ i ≤ k clasele de separare ale lui G in raport cu {*a, b*}. Presupunem ca {*a, b*} satisface (i). Atunci muchia (*b, r*) apartine unei clase de separare, sa zicem E_1. Fiecare arc cu un capat in *D*(*r*) are celalalt capat in *D*(*r*) ∪ {*a, b*}. De asemenea, fiindca LOWPT1 (*r*) = *a* si LOWPT2 (*r*) ≥ b, fiecare frond cu un capat in *D*(*r*) are celalalt capat in *D*(*r*) ∪ {*a, b*}. Asadar E_1 este multimea tuturor muchiilor incidente intr-un *D*(*r*). Nu exista alte muchii in E_1, si muchiile incidente in nodul *s* trebuie sa fie in alta clasa, sa zicem E_2. Deoarece E_1 si E_2 contin fiecare cel putin doua noduri, {*a, b*} este o pereche de separare.

Presupunem ca {*a, b*} satisface (ii). Fie *S* = *D*(*r*) – *D*(*b*). Toate muchiile incidente intr-un nod din *S* se afla in aceeasi clasa de separare, sa zicem E_1. Din moment ce *d* este un prim descendent al lui *r*, *S* = {*x*|*r* ≤ *x* < *b*} conform lemei 1. Fie $b_1, b_2, \ldots, b_n$ fiii lui *b* in ordinea aparitiei lor in *A*(*b*). Fie i_0 = min {*i*|LOWPT1 (b_i) ≥ *a*}. Conform ordonarii lui *A*, *i* < i_0 implica LOWPT1 (b_i) < *a*, si *i* ≥ i_0 implica LOWPT1 (b_i) ≥ *a*. Din (ii) reiese ca fiecare frond cu originea in *S* isi are varful in *S* ∪ {*a*}. De asemenea, tot din (ii), fiecare frond cu originea in *S* isi are varful in *S* ∪ {*b*} ∪ ($\bigcup$ $i \geq i_0$ *D*(b_i)).

Fiecare muchie cu un capat in *D*(b_i), *i* ≥ i_0, isi are celalalt capat in *S* ∪ {*a, b*} ∪ *D*(b_i). Asadar clasa E_1 contine *cel putin* toate muchiile cu un capat in *S*, si *cel mult* toate muchiile cu un capat in *S* ∪ ($\bigcup$ $i \geq i_0$ *D*(b_i)). Cum *a* ≠ 1, muchiile incidente in radacina lui *P* nu pot fi in E_1, asadar {*a, b*} este o pereche de separare.

Acum sa demonstram partea directa a lemei. Presupunem {*a, b*} este o pereche de separare cu *a* < *b*. Daca (*a, b*) este o muchie multipla a lui *G*, atunci e clar ca {*a, b*} satisface (iii). Asadar presupunem ca (*a, b*) nu e muchie multipla. Conform lemei 12, *a* $\xrightarrow{*}$ *b*. Fie E_i, pentru 1 ≤ *i* ≤ *k*, clasa de separare a lui G in raport cu {*a, b*}. Fie *v* fiul lui *a* astfel incat *incat a* → *v* $\xrightarrow{*}$ *b*, *S* = *D*(*v*) – *D*(*b*), si *X* = *V* – *D*(*a*). (Ori *S* ori *X* ori ambele pot fi goale). *E*(*S*) si *E*(*X*) apartin fiecare unei clase de separare, sa zicem *E*(*S*) ⊆ E_1 si *E*(*X*) ⊆ E_2.

Fie $a_i \neq v$ fiul lui a. Daca a are un asemenea fiu, LOWPT1 (a_i) < a. Asta inseamna ca $E(D(a_i)) \subseteq E_2$. Fie $Y = X \cup (\bigcup_i D(a_i))$. Fie $b_1, b_2, \ldots, b_n$ fiii lui b in ordinea aparitiei lor in lista de adiacenta a lui b. Fie $E(D(b_i))$ multimea muchiilor cu un capat in $D(b_i)$. Clasele de separare trebuie sa fie reuniuni ale multimilor $E(S)$, $E(Y)$, $\{(a, b)\}$, $E(D(b_1))$, $E(D(b_2))$, …, $E(D(b_n))$.

Daca $E(D(b_i)) = E_j$ pentru i si j oarecare, atunci LOWPT1 (b_i) = a pentru ca G e biconex, si asta inseamna ca LOWPT1 (b_i) < b conform lemei 7. De asemenea, LOWPT2 (b_i) $\geq$ b. Cum $\{a, b\}$ este o pereche de separare, trebuie sa existe o alta clasa de separare in afara de E_j si $\{(a, b)\}$. Asadar exista un nod s astfel incat $s \neq a$, $s \neq b$, si $s \notin D(b_i)$. Asta inseamna ca $\{a, b\}$ satisface (i) cand r este b_i.

Acum presupunem ca niciun $E(D(b_i))$ nu constituie o clasa de separare de unul singur. Fie i_0 = min$\{i|$LOWPT1 (b_i) $\geq a\}$. Daca $i \geq i_0$, atunci fiindca G este biconex, trebuie ca LOWPT1 (b_i) < b, si clasele de separare sunt $E_1 = E(S) \cup (\bigcup_{i \geq i_0} E(D(b_i))$, $E_2 = E(Y) \cup (\bigcup_{i<i_0} E(D(b_i))$, $E_3 = \{(a, b)\}$. (E_3 ar putea fi vida). Avem $v \neq b$ fiindca $\{a, b\}$ nu este o pereche de tipul 1 si $a \neq 1$ deoarece E_2 este nevida. Daca $x \dashrightarrow y$ este un frond cu $v \leq x < b$, atunci $x \in S$, $(x, y) \in E_1$, si $a \leq y$. Daca $x \dashrightarrow y$ este un frond cu $a < y < b$ si $b \rightarrow b_i \overset{*}{\longrightarrow} x$, atunci $y \in S$, $(x, y) \in E_1$, si $i \geq i_0$, ceea ce inseamna ca LOWPT1 (b_i) $\geq a$. Mai trebuie sa verificam o singura conditie pentru a demonstra (ii), si anume ca b este un prim descendent al lui v. Cum G este biconex, LOWPT1 (v) < a. Asadar un frond cu originea in $D(v)$ are varful mai mic decat a. Conform ordonarii lui A si definitiei unui prim descendent, exista un frond $x \dashrightarrow y$ cu $x \in D(v)$ si $y < a$ astfel incat x este un prim descendent al lui v. Daca b nu ar fi fost un prim descendent al lui v, atunci x ar apartine lui S, si E_1 si E_2 nu ar putea fi doua clase de separare diferite. Asadar b este un descendent al lui v, si (ii) este adevarata cu $r = v$. Aceasta completeaza demonstratia directa a lemei.

Lema 13 si demonstratia acesteia trebuie studiate cu grija. Aceasta lema da trei conditii usor de aplicat pentru perechile de separare. Conditiile (i) si (ii) identifica perechile de separare nontriviale ale multigrafului. Conditia (iii) se ocupa cu mai multe muchii. Conditia (i) necesita efectuarea unui test simplu asupra fiecarui arc din P. Asadar cautarea perechilor de tipul 1 necesita O(V) timp. Cautarea perechilor de tipul 2 este putin mai dificila, dar poate fi efectuata in O($V + E$) timp folosind o alta cautare DF. Fie $\{a, b\}$ o pereche de tipul 2 ce satisface $a \rightarrow r \overset{*}{\longrightarrow} b$, si i_0 = min$\{i|$LOWPT2 (b_i) $\geq a\}$, unde $b_1, b_2, \ldots, b_n$ sunt fiii lui b in ordinea aparitiei lor in $A(b)$. Atunci o clasa de separare in raport cu perechea $\{a, b\}$ este $E(\{x|r \leq x < b_{i0}\} - \{b\})$. Asta reiese din demonstratia lemei 13. Noua numerotare, ce satisface conditia oarecum dubioasa a lemei 9, faciliteaza determinarea claselor de separare si diviziunea grafului, odata ce o pereche de separare a fost localizata. Un algoritm pentru gasirea componentelor split bazat pe lema 13 este descris in sectiunea urmatoare.

4. Identificarea componentelor split. Componentele split se pot identifica examinand drumurile generate in ordine si cautand perechile de separare folosind lema 13. Perechile de separare vor fi de diferite tipuri. Muchiile multiple si perechile de tipul 1 sunt usor de recunoscut. La fel sunt perechile de tipul 2 $\{a, b\}$, unde $a \rightarrow v \rightarrow b$ si b are gradul 2. Alte perechi de tipul 2 sunt mai greu de recunoscut. Fie c primul drum generat (un ciclu). Ciclul consta intr-un set de arce $1 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$ urmate de un frond $v_n \dashrightarrow 1$. Numerotarea nodurilor este facuta intocmai astfel incat $1 < v_1 < v_2 < \ldots < v_n$. Cand c este eliminat, graful se imparte in mai multe componente conexe, numite *segmente*. Fiecare segment este alcatuit ori dintr-un singur frond (v_i, v_j), ori dintr-un arc (v_i, w) plus un subarbore cu radacina w plus toate frondurile care pleaca din acel subarbore. Ordinea generarii drumurilor este astfel incat toate drumurile intr-un segment sunt generate inainte de drumurile din orice alt segment, si segmentele sunt vizitate in ordinea inversa a v_i.

Presupunem ca se repeta cautarea drumurilor, acuma fiind folosita pentru a gasi componente split. Vom pastra o stiva de muchii, adaugand muchii la stiva pe masura ce ne intoarcem pe ele in cautarea noastra. De fiecare data cand gasim o pereche de separare, eliminam din stiva o multime de muchii ce corespund perechii de separare. Adaugam o muchie virtuala corespunzand componentei split la componenta si la stiva respectiv. De asemenea trebuie sa actualizam multe informatii, de vreme ce tatii nodurilor si gradurile nodurilor se pot schimba cand graful este divizat (split). Cautarea completa a drumurilor va crea o multime completa de componente split. Aasamblarea componentelor split pentru a ne da componentele triconexe este apoi o chestie simpla.

Pentru a identifica perechile de tipul 2, pastram o stiva (numita **TSTACK**) de triplete (h, a, b). Perechea $\{a, b\}$ ar putea fi de tipul 2 si h denota nodul numerotat maxim din componenta split corespunzatoare. Perechile se afla in ordine imbricata in stiva; cu alte cuvinte, daca v_i este nodul current (vizitat de cautarea de drumuri), si (h_1, a_1, b_1), (h_2, a_2, b_2), …, (h_k, a_k, b_k) se afla in **TSTACK**, atunci $a_k \leq a_{k-1} \leq \ldots \leq a_2 \leq a_1 \leq v_i \leq b_1 \leq b_2 \leq \ldots \leq b_k$. Mai mult chiar, toate a_i si b_j sunt noduri apartinand ciclului c.

Actualizam **TSTACK** in felurile urmatoare.

1. De fiecare data cand traversam un nou drum $p : s \overset{*}{\Rightarrow} f$, eliminam toate tripletele (h_j, a_j, b_j) din varful stivei cu $a_j > f$. Daca p are un nod secundar $v \neq f$, fie $x = v + \mathrm{ND}(v) - 1$. Altfel fie $x = s$. Fie $y = \max\{h_j | \text{tripletul } (h_j, a_j, b_j) \text{ a fost eliminat din } \textbf{TSTACK}\}$. Daca (h_k, a_k, b_k) a fost ultimul triplet eliminat, adaugam $(\max(x, y), f, s)$ la stiva. Daca nici un triplet nu a fost eliminat, adaugam (x, f, s) la stiva.

2. Cand ne intoarcem pe un arc $v_i \rightarrow v_{i+1}$ cu $v_i \neq 1$, eliminam toate tripletele (h_j, a_j, b_j) din varful lui **TSTACK** care satisfac **HIGHPT** $(v_i) > h_j$. Acest test este necesar pentru a garanta ca elementele ce nu corespund perechilor de tipul 2 nu se acumuleaza in **TSTACK**.

Putem folosi **TSTACK** pentru a gasi perechile de separare in felul urmator: de fiecare data cand ne intoarcem pe un arc de arbore $v_i \rightarrow v_{i+1}$ in timpul cautarii drumului, examinam tripletul din varf (h_1, a_1, b_1) din **TSTACK**. Daca $v_i \neq 1$, $a_1 \neq v_i$, iar $a_i \neq$ **FATHER** (b_i), $\{a_1, b_1\}$ este o pereche de separare de tip 2. Daca **DEGREE** $(v_{i+1}) = 2$ si v_{i+1} are un fiu, atunci v_i si fiul lui v_{i+1} formeaza o pereche de separare de tipul 2.

Impartim componentele care apartin tipului 2 de perechi pana cand aceste doua conditii nu mai furnizeaza componente noi (vom testea simultan componentele care corespund mai multor muchii si le vom imparti si pe acestea). Apoi vom aplica Lema 13 pentru a testa daca $\{v_i, \mathrm{LOWPT1}(v_{i+1})\}$ este de tipul 1, creand o noua componenta daca este cazul (din nou trebuie sa testam daca este vorba de o componenta cu mai multe muchii).

Ne vom ocupa de partea recursiva a algoritmului in felul urmator: traversarea unui drum $p : s \overset{*}{\longrightarrow} f$ care incepe in c inseamna intrarea intr-un nou segment. Nodul f trebuie sa fie cel mai de jos nod al segmentului conform ordinii impuse de cautarea drumului. Dupa ce actualizam TSTACK asa cum a fost descris mai sus, daca p contine mai mult de o muchie, vom plasa un marcaj de sfarsit de stiva in TSTACK si continuam sa cautam drumuri. Asta corespunde unei apelari recursive clasice a algoritmului clasic pentru triconectivitate. Cand ajungem din nou la primul nod din p, stergem lementele din TSTACK pana la marcajul de sfarsit de stiva. Asta corespunde evenimentului de eliberare a stivei din recursivitate.

Mai trebuie explicat modivul din care folosim atat LOWPT2 cat si LOWPT1 pentru a construi A, structura de adiacenta adecvata pentru a determina ordinea de cautare a drumurilor. Acest pas este necesar pentru a putea prelucra toate muchiile multiple in mod corect. Sa presupunem v un nod si $w_1, w_2, \ldots, w_k$ fii lui astfel incat $\mathrm{LOWPT1}(w_i) = u$. Sa consideram $v - \rightarrow u$. Fie w_i ordonate ca in $A(v)$. Exista un i_0 astfel incat $i \leq i_0 \Rightarrow \mathrm{LOWPT}(w_i) < v$ si $i > i_0 \Rightarrow \mathrm{LOWPT}(w_i) \geq v$. In $A(v)$, u va aparea dupa toti w_i cu $1 \leq i \leq i_0$. Daca $i > i_0$, atunci $\{u, w_i\}$ este o pereche de separatie de tip 1; impartind componenta respectiva, vom obtine o noua frond $v - \rightarrow u$. Este important ca w_i cu $i > i_0$ vor aparea impreuna in $A(v)$ astfel incat frondurile virtuale pot fi localizate si imbinate pentru a rezulta componente impartite (limite).

In continuare este o procedura care se bazeaza pe ideile subliniate anterior. Procedura este aplicabila oricarui multigraf biconex pentru care pasii 1, 2 si 3 ai sectiunii precedente au fost realizati. G este reprezentat printr-o serie ordonata de liste de adiacenta $A(v)$. TSTACK contine triplete de posibile perechi de separare tip 2. ESTACK contine muchii salvate pe parcursul cautarii. Restul variabilelor au fost definite anterior.

Procedura recursiva PATHSEARCH repeta cautarea drumului, gasind perechile de separare si impartind componentele. Este bazata pe cele prezentate in aceasta sectiune si in cea precedenta. Nodul v este nodul curent dintr-o parcurgere DF.

PROCEDURE 6.

```
procedure SPLIT (G); begin
    comment procedure to determine split components of G, a biconnected
        multigraph on which steps 1, 2 and 3 have been carried out. G is repre-
        sented by a set of properly ordered adjacency lists A(v). TSTACK
        contains triples representing possible type 2 separation pairs. ESTACK
        contains edges backed up over during search. Other variables have been
        defined in the previous section;
    procedure PATHSEARCH (v); begin
        comment this recursive procedure repeats the pathfinding search, finding
            separation pairs and splitting off components as it proceeds. It is
            based on the material in this section and the last. Vertex v is the
            current vertex in the depth-first search;
        for w ∈ A(v) do
            if v → w then begin
A:              if v → w is first edge of a path then begin
                    y := 0;
                    while (h, a, b) on TSTACK has a > LOWPT1 (w) do begin
                        y := max (y, h);
                        delete (h, a, b) from TSTACK;
                    end;
                    if no triples deleted from TSTACK then add
                        (w + ND (w) − 1, LOWPT1 (w), v) to TSTACK
                    else if (h, a, b) last triple deleted then add
                        (max {y, w + ND (w) − 1}, LOWPT1 (w), b) to
                        TSTACK;
                    add end-of-stack marker to TSTACK;
                end;
                PATHSEARCH (w);
                add (v, w) to ESTACK;
B:              while v ≠ 1 and ((DEGREE (w) = 2) and (A1 (w) > w) or
                        (h, a, b) on TSTACK satisfies (v = a)) do begin
                        comment test for type 2 pairs;
                    if (h, a, b) on TSTACK has (a = v) and
                        (FATHER (b) = a)
                        then delete (h, a, b) from TSTACK;
                    else begin
                        if (DEGREE (w) = 2) and (A1 (w) > w) do begin
                        j = j + 1;
                            add top two edges (v, w) and (w, x) on ESTACK
                                to new component;
                            add (v, x, j) to new component;
                            if (y, z) on ESTACK has (y, z) = (x, v) then begin
                                FLAG := true;
                                delete (y, z) from ESTACK and save;
                            end;
```

```
E:          end else if (h, a, b) on TSTACK satisfies v = a and
                    a ≠ FATHER (b) then begin
                j = j + 1;
                delete (h, a, b) from TSTACK;
                while (x, y) on ESTACK has (a ≦ x ≦ h) and

                    (a ≦ y ≦ h) do
                    if (x, y) = (a, b) then begin
                    FLAG := TRUE;
                        delete (a, b) from TSTACK and save;
                end else begin
                    delete (x, y) from ESTACK and add to current
                        component;
                    decrement DEGREE (x), DEGREE (y);
                end
                add (a, b, j) to new component;
                x := b;
            end;
            if FLAG then begin
                FLAG := false;
                j := j + 1;
                add saved edge, (x, v, j − 1), (x, v, j) to new
                    component;
                decrement DEGREE (x), DEGREE (v);
            end;
            add (v, x, j) to ESTACK;
            increment DEGREE (x), DEGREE (v);
            FATHER (x) := v;
            if A1 (v) ↛ x then A1 (v) = x;
            w := x;
        end;
        comment test for a type 1 pair;
G:      if (LOWPT2 (w) ≧ v) and ((LOWPT1 (w) ≠ 1) or
            (FATHER (v) ≠ 1)
                or (w > 3))
            then begin
                j := j + 1;
                while (x, y) on top of ESTACK has
                    (w ≦ x < w + ND (w)) or
                        (w ≦ y < w + ND (w))
                    then begin
                        delete (x, y) from ESTACK;
                        add (x, y) to new component;
                        decrement DEGREE (x), DEGREE (y);
                    end;
                add (v, LOWPT1 (w), j) to new component;
                if A1 (v) = w then A1 (v) := LOWPT1 (w);
                comment test for multiple edge;
```

```
                    if (x, y) on top of ESTACK has
                        (x, y) = (v, LOWPT1 (w))
                            then begin
                        j := j + 1;
                        add (x, y), (v, LOWPT1 (w), j − 1),
                            (v, LOWPT1 (w), j) to new component;
                        decrement DEGREE (v),
                            DEGREE (LOWPT1 (w));
                    end;
                    if LOWPT1 (w) ≠ FATHER (v) then begin add
                            (v, LOWPT1 (w), j) to ESTACK;
                        increment DEGREE (v),
                            DEGREE (LOWPT1 (w));
                    end else begin
                        j := j + 1;
                        add (v, LOWPT1 (w), j − 1),
                            (v, LOWPT1 (w), j), tree arc
                                (LOWPT1 (w), v) to new component;
                        mark tree arc (LOWPT1 (w), v) as virtual edge j;

                    end;
                end;
C:          if v → w is the first edge of a path then delete all entries on
                TSTACK down to and including end-of-stack marker;
D:          while (h, a, b) on ESTACK has HIGHPT (v) > h, do delete
                (h, a, b) from TSTACK;
            end else begin comment v− → w;
F:          if v− → w is first (and last) edge of a path then begin
                y := 0;
                while (h, a, b) on TSTACK has a > w do begin
                    y := max (y, h);
                    delete (h, a, b) from TSTACK;
                end;
                if no triples deleted from TSTACK then add (v, w, v) to
                    TSTACK;
                if (h, a, b) last triple deleted then add (y, w, b) to TSTACK;
            end;
            if w = FATHER (v) then begin
                j := j + 1;
                add (v, w), (v, w, j), tree arc (w, v) to new component;
                decrement DEGREE (v), DEGREE (w);
                mark tree arc (w, v) as virtual edge j;
            end else add (v, w) to ESTACK;
```

```
    end; end;
    j := 0;
    FLAG := false;
    PATHSEARCH (1);
end;
```

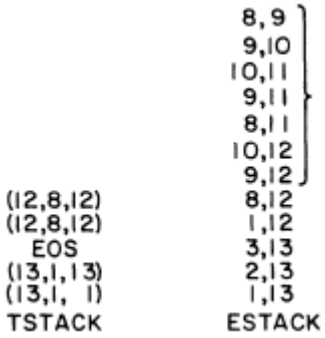
LEMA 14. SPLIT imparte un multigraf biconex *G* in mod corect in componentele sale.

Demonstratie: trebuie demonstrate doua lucruri: (1) daca *G* este triconex, SPLIT nu il va imparti si (2) daca *G* nu e triconex, algoritmul il va imparti in componente triconexe. Odata ce stim aceste doua lucruri, putem demonstra lema prin inductie dupa numarul de muchii din graf. Testele pentru muchii multiple, pentru tipul 1 de perechi de separare si pentru muchiile de grad 2 sunt evidente (testele de tip 1 (*G* in PATHSEARCH) include conditia (LOWPT1(*w*)≠1) sau (TATA(*v*) ≠1) sau (*w*>*3*) pentru a fi siguri ca un nod sta in afara componentei corespunzatoare). Aceste teste vor descoperi o pereche de separare daca exista. Astfel trebuie doar sa demonstram ca testele de tip 2 functioneaza corect pe un multigraf fara noduri de grad 2, muchii multiple sau perechi de separare de tip 1 si vom demonstra (1) si (2).

Sa presupunem ca *G* este un multigraf biconex fara noduri de grad 2, muchii multiple su perechi de separare de tip 2. Sa consideram testele de tip 2 si schimbarea continutului lui TSTACK pe parcursul cautarii in *G*. Daca (h_1 , a_1 , b_1), ..., (h_k , a_k , b_k) apartin de TSTACK si se afla deasupra celui mai inalt marcaj de sfarsit de stiva, si daca *v* este nodul curent, atunci $a_k \leqq a_{k-1} \leqq ... \leqq a_1 \leqq v \leqq b_1 \leqq ... \leqq b_k$. Acestea rezulta din inductie dintr-o prelucrare a schimbarilor posibile ce pot aparea in TSTACK (propozitiile A, B, C, D, E, F in PATHSEARCH). In mod similar, daca niste frond $x - \rightarrow y$ cu $a<y<b$ si $b \rightarrow w \xrightarrow{*} x$ ar fi avut LOWPT1(*w*) < *a*, si tripletul din TSTACK corespundea lui (h,a,b) ar fi fost sters de HIGHPT cand nodul *y* a fost vizitat (D in PATHSEARCH). Rezulta ca {*a*,*b*} este o pereche de separare de tip 2 din Lema 13.

Sa presupunem ca *G* are perechea de tip {*a*,*b*}. Fie b_1 , ... , b_n fii lui *b* in ordinea in care apar in *A(b)*. Fie $i_0 = \min\{i \mid \text{LOWPT1}(b_i) \geqq a\}$. Daca i_0 exista atunci (b_{i_0} + ND(b_{i_0}), LOWPT1(b_i), *b*} vor fi plasati in TSTACK atunci cand este procesata muchia $b \rightarrow b_i$. Acest triplet poate fi sters din TSTACK, dar va fi intotdeauna inlocuit de un triplet (*h*,*x*,*b*), cu LOWPT1(b_i) $\geqq x \geqq a$. In cele din urma, un astfel de triplet va satisface testele de tip 2, doar daca o alta pereche de tip 2 nu a fost gasita inainte. Daca i_0 nu exista, fie (*i*,*j*) prima muchie traversata dupa ce a fost atins *b* astfel incat $a \leqq i$ si $j \leqq b$. Daca $i - \rightarrow j$ atunci (*i*,*j*,*i*) va fi introdus in TSTACK, probabil modificata si eventual selectata ca si o pereche de tip 2, daca alta pereche de tip 2 nu este gasita anterior. Daca $i \rightarrow j$, atunci (*j*+ND(*j*), LOWPT1(*j*), *i*) va fi inserat in TSTACK, probabil modificata, si daca nu a fost gasita alta pereche de tip 2. Astfel, daca exista pereche de tip 2, atunci cel putin una va fi gasita de algoritm. Rezulta ca testul de tip 2 functioneaza corect si algoritmul imparte un multigraf daca si numai daca exista o pereche de separare.

Lema rezulta prin inductie dupa numarul de muchii din *G*. Sa presupunem ca lema e corecta pentru grafurile cu mai putin de *k* noduri. Daca *G* nu poate fi impartit, algoritmul functioneaza corect datorita demonstratiei. Daca *G* poate fi impartit, va fi impartit. Sa consideram prima impartire realizata in algoritm, care va genera grafurile G_1 si G_2. Comportamentul algoritmului pe graful *G* rezulta din comportarea pe subgrafurile G_1 si G_2. Din moment ce G_1 si G_2 sunt impartite corect din ipoteza inductiei, atunci si *G* trebuie sa fie impartit corect. Figura 5 arata continuturile lui ESTACK si TSTACK atunci cand prima pereche (8,12) este gasita in graful din Fig. 1.



Lema 15. Algoritmul elementelor triconexe proceseaza un graf *G* cu *V* noduri si *E* muchii in O(*V*+*E*).

Demonstratie: Numarul de muchii intr-un set de componente ale lui *G* este limitat la *3E-6*, conform lemei 1. Toti pasii, mai pucin gasirea componentelor impartite, necesita O(*V*+*E*) timp, conform ultimelor doua sectiuni. Sa consideram executia algoritmului SPLIT. Fiecare muchie este plasata in ESTACK odata si este stearsa o data. Parcugerea DF necesita O(*V*+*E*) timp, incluzand testele. Numarul de triplete adaugate in TSTACK este O(*V*+*E*). Fiecare triplet poate fi modificat doar daca este in varful stivei. Astfel timpul maxim pentur a intretine TSTACK este de asemenea O(*V*+*E*), iar SPLIT necesita O(*V*+*E*) timp.

Astfel se completeaza prezentarea unui algoritm O(*V*+*E*) – liniar – pentru a determina componentele triconexe. Acest algoritm poate fi folosit in costruirea unui algoritm O(*V logV*) pentru a determina izomorfisml grafurilor planare. Algoritmul nu este optim doar teoretic, dar si practic.