# O implementare in timp liniar a SPQR-tree
# Algoritmul - Gutwenger si Mutzel

**Rezumat.** Structura de date SPQR-tree reprezinta descompunerea unui graf biconex in raport la componentele sale triconexe. SPQR-tree a fost introdus de Di Battista si Tarnassia, devenind un concept important in teoria grafurilor. Articolele teoretice privitoare la SPQR-trees sustin ca acestia poate fi implementati in timp liniar folosind o modificare a algoritmului lui Hopcroft si Tarjan pentru descompunerea unui graf in componente triconexe.
Pana acum nu se cunoaste nici o implementarea corecta -liniara in timp-, a descompunerii triconectivitatii sau a SPQR-trees. In acesta articol, aratam incorectitudinea algorimului Hopcroft si Tarjan, si corectam partile gresite. Descriem relatia dintre SPQR-tree si componentele triconexe si aplicam algoritmul rezultat in determinarea SPQR-trees.

## 1. Introducere.

Structura de date SPQR-tree reprezinta descompunerea unui graf biconex in raport la componentele sale triconexe. SPQR-tree a fost introdus de Di Battista si Tarnassia, devenind un concept important in teoria grafurilor.

Incepand de atunci, SPQR-trees au devenit o structura de date importanta. Multi algoritmi liniari care functioneaza doar pentru grafuri triconexe pot fi extinsi pentru grafuri biconexe, folosind SPQR-trees.

In acest articol prezentam un algoritm liniar pentru implementarea structurii de date SPQR-trees. Descriem relatia dintre SPQR-tree si componentele triconexe si aratam incorectitudinile algoritmului lui Hopcroft si Tarjan. Vom realiza un algoritm corect pentru descompunerea grafurilor in componente triconexe prin corectarea si inlocuirea partilor gresite, aplicandu-l si in calcularea SPQR-trees. Articolul este structurat dupa cum urmeaza:

- In sectiunea 3 sunt descrise notiunile de baza privitoare la SPQR-trees si componentele triconexe;
- In sectiunea 4 este prezentat algoritmul pentru determinarea SPQR-trees si a descompunerii triconexe.
- In sectiunea 5 se prezinta partile incorecte ale algoritmului Hopcroft /Tarjan si corectiile aduse.

## 2. Notiuni introductive

Fie graf $G=(V,E)$ un multigraf neorientat , in care $V$ reprezinta multimea de varfuri si multimea $E$ a muchiilor, continand perechi neorientate $(v,w)\in V$. O muchie $(v,v)$ este numita „self-loop”. Daca perechea $(u,v)$ se regaseste in $E$ de mai multe ori atunci ea se numeste muchie multipla. Graful $G$ este simplu daca nu contine self-loop si muchii multiple.
Daca $E'$ este o multime de muchii a lui $G$, $V(E')$ este un set de noduri incidente in una sau mai multe muchii din $E'$.

Un drum $p:v\Rightarrow w$ in $G$ este o secventa de varfuri si muchii care duce de la $v$ la $w$. Drumul este elementar(simplu) daca toate varfurile sunt distincte. Daca $p:v\Rightarrow w$ este un drum simplu atunci $p$ plus muchia $(w,v)$ este ciclu.

Un multigraf neorientat este conex daca pentru orice pereche de varfuri exista un drum care le uneste. 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.
Fie $G=(V,E)$ un multigraf biconex si $a,b \in V$. Multimea $E$ poate fi impartita in clase de echivalenta $E_1$, $E_2$..$E_n$ astfel incat doua muchii sunt in aceeasi clasa, daca orice drum comun a lor, 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 putin 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 $G$ nu contine perechi de separatie, atunci ele este triconex.

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 $v \xrightarrow{*} w$. Daca $v \rightarrow w$ atunci $v$ est eparintele lui $w$, iar $w$ este copilul lui $v$. Daca $v \xrightarrow{*} w$ atunci $v$ este ascendent pentru $w$ iar $w$ este descendent a lui $v$. 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 $G$.

Un *palm tree* $P$ este un multigraf *orientat* astfel incat fiecare arc din $P$ este fie arc din arbore, notat $v \rightarrow w$ fie *frond* notat $v \dashrightarrow w$ satisfacand urmatoarele proprietati:

i. Subarborele $T$ format din toate arcele $v \rightarrow w$ este un arbore de acoperire a lui $P$.
ii. Daca $v \dashrightarrow w$ atunci $w \xrightarrow{*} v$.

Fie $G=(V,E)$ un multigraf biconex, $\{a,b\}$ o pereche de separare a lui $G$ si $E_1,E_2,\ldots E_n$ 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 $|E'| \geq 2$, $|E''| \geq 2$. Grafurile $G_1=((V(E'), E' \cup \{e\})$ si $G_2 =(V(E''), E'' \cup \{e\})$ se numesc „split graph-uri” a lui $G$, in raport cu $\{a,b\}$, unde $e$ o noua muchie. Inlocuirea unui multigraf $G$ cu doua „split graph-uri” se numeste „impartirea” lui $G$ (splitting $G$). Fiecare split-graf este de asemenea biconex. Fiecare muchie $e$ este numita muchie virtuala si identifica operatia split.

Presupunem ca $G$ este impartit, apoi split-grafurile se impart si asa mai departe pana cand nu mai sunt posibile noi impartiri. Grafurile rezultate sunt numite componentele split ale lui $G$. Fiecare dintre ele sunt:

- Un set de trei muchii multiple (triple bond)
- Un ciclu de lungime trei
- Un graf simplu triconex.

Componentele split nu sunt in mod necesar unice.

**Lema 1**. Fie $G$ un multigraf:

i. Fie care muchie din $E$ apare in exact o componenta split si fiecare muchie virtuala apare in exact doua componente split.
ii. Numarul total de muchii din componentele split sunt cel mult 3|E|-6.

Fie $G_1=(V_1,E_1)$ si $G_2=(V_2,E_2)$ sunt doua componente „split", amandoua continand o muchiie virtuala *e*. Atunci $G'=(V_1 \cup V_2 , E_1 \cup E_2-\{e\})$ este numit „merge graph" a lui $G_1$ si $G_2$.
Inlocuirea a doua componente $G_1$ si $G_2$ prin „merge"-graful lui $G'$ se numeste operatia „merge" a lui $G_1$ si $G_2$. Componentele triconexe ale lui *G* se obtin din componentele split prin unirea legaturilor triple (triple bonds) intr-o multime maximala de muchii multiple (bonds) si a triunghiurilor in cicluri simple maximale(poligonuri)

**Lema 2**. Componentele triconexe ale lui *G* sunt unice.

Componentele triconexe sunt strans legate de SPQR-trees. SPQR-trees au fost initial definite doar pentru grafurile planare.

Fie *G* un graf biconex. O *pereche split* a lui *G* este fie o pereche de varfuri adiacente fie o pereche de separatie. O *componenta split* a unei perechi split {*u,w*}este fie o muchie (*u,v*), fie un subgraf maximal *C* al lui *G* in care {u,w} nu este o pereche split a lui *C*. Fie {*s,t*} o pereche split a lui *G*. O pereche {u,v} a lui *G este pereche split maximala* in raport cu {s,t} daca pentru orice pereche split {*u',v'*}, varfurile *u , v, s, t* sunt in aceeasi componenta split.

Fie $e = (s, t)$ o muchie a lui *G*, numita *muchia de referinta*. Arborele SPQR *T* al lui *G* raportat la *e* este un arbore cu radacina ale carui noduri sunt de patru tipuri: **S**, **P**, **Q** si **R**. Fiecare nod μ al lui *T* are asociat un multi-graf biconex numit *scheletul lui* $\mu$. Arborele *T* este definit recursiv dupa cum urmeaza:

*Cazul trivial:* Daca *G* se compune din exact doua muchii paralele intre *s* si *t* , atunci *T* se compune dintr-un singur nod **Q** al carui schelet este chiar *G*.

*Cazul paralel*: Daca perechea de separare {*s, t*} are cel putin trei componente split $G_1, \dots , G_k$, radacina lui T este un nod de tip **P** din μ, al carui schelet se compune din k muchii paralele $e=e_1,\dots,e_k$ intre *s* si *t*.

*Cazul serie* : Altfel, perechea de separatie {*s, t*} are exact doua componente split, una din ele este *e*, iar cealalta este data de *G"*. Daca *G"* are puncte de articulatie $c_1, \dots , c_{k-1}$ ( $k\geq 2$ ) care impart graful in componentele $G_1 \dots G_k$, in aceasta ordine de la *s* la *t*, radacina lui *T* este un nod de tip **S** μ, al carui schelet este ciclul $e_1, \dots , e_k$, unde $e_0 = e$, $c_0 = s$, $c_k = t$, iar $e_i = ( c_{i-1}, c_i )$ ( $i = 1, \dots , k$ ).

*Cazul rigid* : Daca niciunul din cazurile de mai sus nu se aplica, fie $\{s_1, t_1\},\dots,\{s_k, t_k\}$ perechile impartite maximale in raport cu {*s, t*} ($k\geq 1$), si, pentru $i = 1,\dots, k$, fie $G_i$ reuniunea tuturor elementelor impartite ale $\{s_i, t_i\}$ in afara celei car il contine pe *e*. Radacina lui *T* este un nod de tip *R*, al carui schelet se obtine din *G* inlocuind fiecare subgraf $G_i$ cu muchia $e_i = (s_i, t_i)$.

Cu exceptia cazului trivial, μ are fii pe $\mu_1 \dots \mu_k$ astfel incat $\mu_i$ este radacina arborelui SPQR al lui $G_i$ *U* $e_i$ in raport cu $e_i$ ( $i = 1,\dots,k$ ). Extremitatile muchiei $e_i$ sunt numite *poli* ai nodului $\mu_i$. Muchia virtuala a nodului $\mu_i$ este muchia $e_i$ a scheletului lui $\mu$. Arborele T este completat adaugand un nod de tip Q, reprezentand nodul-referinta *e*, si marcandu-l ca parinte al lui $\mu$, devenind astfel radacina.

Fiecare muchie in G este asociata cu un nod de tip Q din T. fiecare mushie $e_i$ din scheletul $\mu$ este asociata cu fiul $\mu_i$ al lui $\mu$. Este posibil sa notam ca radacina a lui T un nod

arbitrar $\mu'$ de tip Q, rezultnd un arbore SPQR raportat la muchia asociata lui $\mu'$. In implementarea noastra folosim o definitie putin diferita, dar echivalenta, a arborelui SPQR. Omitem nodurile de tip Q si facem in schimb diferenta intre muchii reale si muchii virtuale in graful schelet. O muchie in scheletul lui $\mu$ care este asociata cu un nod de tip Q din definitia originala este o muchie reala care nu este asoiata vreunui fiu al lui $\mu$, toate celelalte muchii de schelet sunt asociate cu noduri de tip P, S sau R. Folosind definitia modificata, putem demonstra ca grafurile schelet sunt componentele unice triconexe ale lui $G$:

**Teorema 1.** Fie $G$ un multi-graf biconex si $T$ arborele sau SPQR.

(i) Grafurile schelet ale lui T sunt componentele triconexe ale lui $G$. Nodurile de tip P corespund legaturilor, nodurile de tip S corespund poligoanelor, iar nodurile de tip R sunt grafuri simple triconexe.

(ii) Exista o muchie intre doua noduri $\mu$, $v$ apartin lui T daca si numai daca elementele triconexe corespunzatoare au in comun o muchie virtuala.

(iii) Dimensiunea lui T, inclusiv a tuturor grafurilor-scheleti este liniara raportat la dimensiunea lui $G$.

Demonstratie ( succinta ) Observam ca daca { $u$, $v$ } este o pereche de separatie, componentele impartite ale lui { $u$, $v$ } sunt clase de separare in raport cu { $u$, $v$ }. In cazurile paralel, serie si rigid ale definitiei arborelui SPQR, subgrafurile $G_1,\ldots,G_k$ sunt luate in considerare. Presupunem ca $G_1,\ldots,G_l$ contin mai mult de o muchie si $G_{l+1},\ldots,G_k$ contin exact o muchie. In fiecare din cele trei cazuri, pasul de dezcompunere recursiva poate fi realizat prin efectuarea a $l$ operatii de impartire, fiecare desprinzand un $G_i$, $1\leq i\leq l$ si introducand o noua muchie virtuala $e'$ in scheletul nodului $\mu$ si in scheletul unui fiu $\mu_i$ al lui $\mu$. Devreme ce $e'$ ramane in scheletul lui $\mu_i$ in pasi consecutivi, urmeaza partea (ii) a teoremei.

Grafurile-scheleti finale sunt fiecare un poligon, o legatura sau un drum simplu triconex si oricare doua poligoane, ca si oricare doua legaturi nu au in comun nicio muchie virtuala. Prin urmare, grafurile-scheleti sunt descompunerea unica in elemente triconexe a lui $G$ demonstrand partea (i). Ultima partea a teoremei reiese direct din (i) si Lema 1.

## 4. Algoritmul

Fie $G$ un multi-graf triconex fara cicluri. Conform Teoremei 1, este suficient sa calculam componentele triconexe ale lui $G$, acestea dandu-ne suficiente informatii pentru a construi arborele SPQR al lui $G$. Corectam partile defectuoase din algoritmul lui Hopcroft si Tarjan si aplicam acest algoritm pentru gasirea componentelor triconexe. Ne concentram asupra gasirii perechilor de separatie deoarece desrierea acestei parti in [15] nu este doar confuza dar contine si erori grave. Pentru o imagine de ansamblu asupra algoritmului lui Hopcroft si Tarjan, consultati [13] sau [15].

### 4.1 Identificarea arborilor SPQR

Datele de intrare ale algoritmului reprezinta un multi-graf biconex G = (V , E) si o muchie de referinta $e_r$. La primul pas, Muchiile multiple sunt inlocuite cu o muchie virtuala, dupa cum arata in Algoritmul [1]. Aceasta creaza un set de legaturi $C_1,\ldots,C_k$, rezultand un graf simplu $G'$. Sortarea necesara a muchiilor din linia 1 poate fi facuta in $O(|V| + |E|)$ folosind bucket sort de doua ori. Prima data se sorteaza dupa varful cu index mai mic, apoi dupa cel cu index mai mare, presupunand ca varfurile au indici distincti in intervalul

1,..,|V|. Bucla *for* din linia a 2-a parcurge toate muchiile deci Algoritmul 1 are complexitatea $O(|V| + |E|)$.

**Algoritmul 1:** Impartirea muchiilor multiple

```
1.1 Sortarea muchiilor astfel incat toate muchiile multiple sa vina una
dupa cealalta.
1.2 pentru fiecare grup maximal de muchii multiple e1,…,el cu l ≥ 2 executa
      fie e1,…,el muchii intre v si w
      inlocuieste e1,…,el cu o noua muchie e' = ( v , w )
      creaza o noua componenta C = { e1,…,el, e'}
     sfarsit_pentru
```

Al doilea pas determina componentele split $C_{k+1},....C_m$ a lui $G'$. Rutina este prezentata in detaliu in subsectiunea urmatoare. Componentele triconexe ale grafului initial $G$, sunt create prin reansamblarea partiala a componentelor $C_1,...C_m$. Cat timp duua legaturi(bonds) sau doia poligoane $C_i$ si $C_j$ contin aceeasi muchie virtuala, ele sunt unite. Aceasta este prezentata in algoritmul 2. Componentele sterse sunt marcate ca vide. Bucla forall din linia 2.1 treverseaza toate muchiile din $C_i$ care au fost adaugate la $C_i$ in timpul iteratiilor. Testul din linia 2.1 poate fi facut in timp constant preprocesand pentru fiecare muchie virtuala $e$ cele doua componente care-l contin pe $e$. Reprezentam muchiile dintr-o componenta $C_i$ printr-o lista de muchii care permit implementarea setului de operatii din liniile 2.3 si 2.4. in timp constant. In acord cu Lema 1, numarul total de muchii din toate componentele este $O(|E|)$, deci algoritmul 2 poate fi implementat in $O(|V| + |E|)$.

**Algoritmul 2:** Construirea componentelor triconexe

```
pentru i ← 1 , m executa
   daca Ci ≠ ∅ and Ci este „bond" sau „poligon" atunci
 2.1     pentru_toate e ∈ Ci executa
 2.2      daca exista j ≠ i cu e ∈ Cj si type(Ci)=type(Cj) atunci
 2.3            Ci ← (Ci ∪ Cj)\{e}
 2.4            Cj ← ∅
          Sfarsit_daca
      Sfarsit_pentru
    Sfarsit_daca
 Sfarsit_pentru
```

Pasul precedent ne da destule informatii pentru a contrui SPQR-tree $T$ a lui $G$. Aplicand teorema 1 este simplu de construit versiunea fara radacina (unrooted) a lui $T$. Deoarece omitem nodurile de tip Q din reprezentare, vom „root" T din nodul al carui schelete contine muchia referinta $e_r$. In timpul constructiei vom crea cross-links intre fiecare muchie a arborelui $\mu \rightarrow \nu$ din $T$ si cele doua muchii virtuale corespunzatoare din scheletul lui $\mu$ si scheletul lui $\nu$.

### 4.2. Indentificarea perechilor de separatie

Presupunem ca am obtinut un palm tree $P$ pentru un graf simplu biconex $G'=(V,E')$, iar varfurile din $G'$ sunt numerotate 1, …|V|. In cele ce urmeaza vom identifica varfurile cu numere. Introducem notatia urmatoare:

$$\textbf{lowpt1} = \min(\{v\} \cup \{w \mid v \xrightarrow{*} - \rightarrow w\})$$

$$\textbf{lowpt2} = \min(\{v\} \cup \{w \mid v \xrightarrow{*} \dashrightarrow w\} \setminus \{\textbf{lowpt1}\})$$

Deci, *lowpt1*(*v*) este cel mai de jos varf (nivel minim) ce poate fi atins din *v*, traversand zero sau mai multe muchii de avans, si o singura muchie de intoarcere (fronds). *Lowptt2*(*v*) al doilea cel mai de jos varf atins in acelasi mod(sau *v* daca nu exista).

Notam cu *Adj*(*v*) lista de adiacenta ordonata (non-ciclica) a unui nod *v*, si cu *D*(*v*) multimea descendentilor lui *v*. Cautam o numerotare a nodurilor si o ordonare a muchiilor din listele de adiacenta ce indeplinesc proprietatile urmatoare:

(P1) radacina lui *P* este 1.

(P2) daca $v \in V$ si $w_1, w_2, \ldots, w_n$ sunt copiii lui *v* in *P* conform ordonarii din *Adj*(*v*), atunci $w_i = w + |D(w_{i+1}) \cup \ldots \cup D(w_n)| + 1$.

(P3) muchiile *e* din *Adj*(*v*) sunt in ordine crescatoare dupa *lowpt*1(*w*) daca $e = v \rightarrow w$, sau dupa *w* daca $e = v \dashrightarrow w$.

Fie $w_1, w_2, \ldots, w_n$ copiii lui *v* cu *lowpt*1($w_i$) = *u* in ordinea data de *Adj*(*v*). Atunci exista un $i_0$ astfel incat *lowpt*2($w_i$) < *v* pentru $1 \le i \le i_0$, si *lowpt*2($w_j$) $\ge$ *v* pentru $i_0 < j \le n$.

Daca $v \dashrightarrow u \in E'$, atunci $v \dashrightarrow u$ intra in *Adj*(*v*) intre $v \rightarrow w_{i0}$ si $v \rightarrow w_{i0+1}$.

Este aratat in [15] cum se creaza o asemenea numerotare a nodurilor si ordonare a listelor de adiacenta in timp liniar. Spre deosebire de [15], este necesar ca un frond $v \dashrightarrow w$, daca este continut in *E*', sa intervina intre $v \rightarrow w_{i0}$ si $v \rightarrow w_{i0+1}$ in *Adj*(*v*). Asta se poate face usor adaptand functia de sortare $\phi$ folosita in [15]:

$$\phi(e) = \begin{cases} 3\text{lowpt}(w) \text{ daca } e = v \rightarrow w \text{ si lowpt2}(w) < v \\ 3w + 1 \text{ daca } e = v \dashrightarrow w \\ 3\text{lowpt1}(w) + 2 \text{ daca } e = v \rightarrow w \text{ si lowpt2}(w) \ge v \end{cases}$$

Ordonarea necesara se poate obtine sortand muchiile dupa valorile lor $\phi$ folosind bucket sort. Folosind ordonarea $\phi$ si procedura PATHSEARCH dupa cum e sugerat in [15] nu se vor putea recunoaste toate muchiile multiple si asadar nu se vor putea identifica corect componentele split ale lui *G'*.

Prespunem ca efectuam o cautare DFS pe *G'* folosind ordonarea muchiilor din listele de adiacenta. Asta il imparte pe *G'* intr-o multime de drumuri alcatuite din zero sau mai multe arce urmate de un frond. Primul drum incepe cu nodul 1 si un drum se termina cand primul frond al drumului este intalnit (vezi Fig. 1). Fiecare drum se termina in cel mai mic nod posibil, si are numai primul si ultimul nod in comun cu drumurile traversate anterior. Din fiecare drum de acest fel $p : v \Rightarrow w$, putem forma un ciclu prin adaugarea drumului de la *w* $\xrightarrow{*}$ *v* la *p* (compara [15] [16]).

*Exemplul 1*. Fig. 1 arata un palm tree cu o numerotare care satisface (P1)-(P3). Muchiile sunt numerotate conform drumurilor generate. Drumurile generate sunt

1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 13 \dashrightarrow 1$
2: $13 \dashrightarrow 2$
3: $3 \rightarrow 4 \dashrightarrow 1$
4: $4 \rightarrow 5 \rightarrow 8 \dashrightarrow 1$
5: $8 \rightarrow 9 \rightarrow 10 \rightarrow 12 \dashrightarrow 1$
6: $12 \dashrightarrow 8$
7: $12 \dashrightarrow 9$
8: $10 \rightarrow 11 \dashrightarrow 8$
9: $11 \dashrightarrow 9$
10: $5 \rightarrow 6 \rightarrow 7 \dashrightarrow 4$
11: $7 \dashrightarrow 5$
12: $6 \dashrightarrow 4$

Mai avem nevoie doar de o definitie: $u_n$ *este un prim descendent al lui* $u_0$ daca $u_0 \rightarrow ... \rightarrow u_n$ si fiecare $u_i \rightarrow u_{i+1}$ este o prima muchie in $Adj(u_i)$. Mai departe, consideram un palm tree $P$ ce indeplineste conditiile (P1)-(P3). Urmatoarea lema ne da trei conditii usor de verificat pentru perechile de separare.

**Lema 3**. (Lema 13 in [15]) Fie $G = (V, E)$ *un graf biconex si* $a, b$ *doua noduri din* $G$ *cu* $a < b$. *Atunci* $\{a, b\}$ *este o pereche de separare daca si numai daca una din urmatoarele conditii este adevarata.*

*Cazul tipului 1: Exista nodurile distincte* $r \neq a, b$ *si* $s \neq a, b$ *astfel incat* $b \rightarrow r$, $lowpt1(r) = a$, $lowpt2(r) \geq b$, *iar* $s$ *nu este un descendent de-al lui* $r$.
*Cazul tipului 2: Exista un nod* $r \neq b$ *astfel incat* $a \rightarrow r \overset{*}{\longrightarrow} b$, $b$ *este un prim descendent al lui* $r$, $a \neq 1$, *fiecare frond* $x \dashrightarrow y$ *cu* $r \leq x < b$ *are* $a \leq y$ *si fiecare frond* $x \dashrightarrow y$ *cu* $a < y < b$ *si* $b \rightarrow w \overset{*}{\longrightarrow} x$ *are* $lowpt1(w) \geq a$.
*Cazul mulchiei multiple:* $(a, b)$ *este o muchie multipla a lui* $G$ *si* $G$ *contine cel putin patru muchii.*

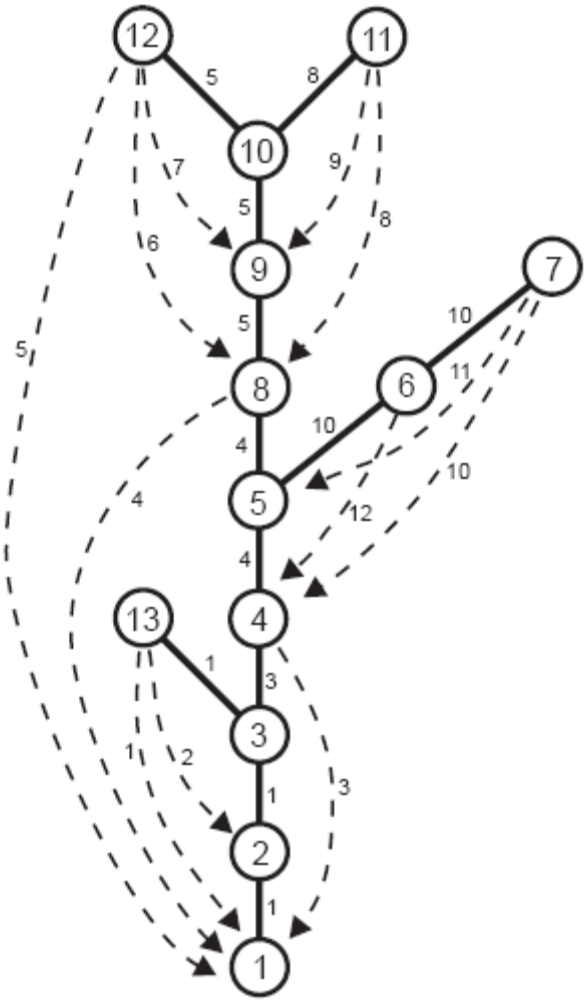


**Fig. 1.** Palm tree cu noduri numerotate si drumuri generate.

*Exemplul 2.* Consideram palm tree-ul din Fig. 1. Avem urmatoarele perechi de separare:

perechi de tipul 1: (1, 4), (1, 5), (4, 5), (1, 8), (1, 3)
perechi de tipul 2: (4, 8), (8, 12)

### 4.3. Identificarea componentelor split

Pe parcursul algoritmului vom pastra un graf $G_c$ si un palm-tree $P_c$ de la $G_c$. Vom nota cu *deg(v)* gradul lui *v* in $G_c$, cu $v \rightarrow w$ un arc in $P_c$, cu $v \dashrightarrow w$ un frond in $P_c$, cu *tatal(v)* parintele lui *v* in $P_c$. De fiecare data cand identificam o componenta split *C*, o impartim si $G_c$ si $P_c$ sunt reactualizate. Vom folosi urmatoarele functii pentru update:

- `C:=new_component(e1, ..., el)` : o noua componenta C = {$e_1$, ..., $e_l$} este creata si $e_1$, ..., $e_l$ sunt scoase din $G_c$
- `C:=C∪{e1,...,el}`: muchiile $e_1$, ...,$e_l$ sunt adaugate in *C*si scoase din $G_c$
- `e' :=new_virtual_edge(v,w,C)` : o noua muchie virtuala *e'*=(*v*,*w*) este creata si adaugata componentei *C* si $G_c$

Vom defini si functiile de acces :

- `firstChild(v)` = primul fiu al lui *v* din $P_c$ conform lui *Adj*(*v*)
- `high(w)` = $\begin{cases} 0 \text{ daca } F(w) = \phi \\ \text{nodul sursa al primei muchii vizitate din } F(w) \text{ altfel} \end{cases}$, unde $F(w) = \{v \mid v \dashrightarrow w \in E_c\}$

Vom folosi si doua stive pentru care sunt definite functiile clasice `push, pop` si `top`:

- ESTACK contine muchiile deja vizitate care nu au fost atribuite unei componente split inca
- TSTACK contine tripletele(*h*,*a*,*b*) (sau un marcaj special de sfarsit de stiva – EOS), astfel incat {*a*,*b*} este o pontentiala pereche de sperarare de tip 2, iar *h* este nodul cu numarul cel mai mare din componenta ce va fi impartita.

Algoritmul incepe prin a apela recursiv procedura `PathSearch` pentru nodul 1, radacina lui *P*. Cand se va intoarce din apel, muchiile care apartin componentei split vor fi in ESTACK.

**Algoritmul 3:** Gaseste componente split

```
    TSTACK.push(EOS);
    PathSearch(1);
    fie e1, ..., el muchii in ESTACK
3.1 C:=new_component(e1,...,el);
```

Procedura PathSearch este aratata in Algoritmul 4. Testarea perechiilor de separare prin aplicarea Lemei 3 este ilustrata separat in Algoritmul 5 pentru tipul 2 si Algoritmul 6 pentru tipul 1 de perechi de separare (algoritmul nu va gasi *toate* perechiile, dar doar pe acelea de care este nevoie la impartirea grafului in componentele split). Pentru a atinge timpul liniar, vom avea nevoie de urmatoarele structuri de date:

**Algorithm 4:** PathSearch($v$)

```
    forall e ∈ Adj(v) do
        if e = v → w then
            if e starts a path then
                pop all (h,a,b) with a > lowpt1(w) from TSTACK
                if no triples deleted then
                    TSTACK.push(w + ND(w) − 1, lowpt1(w), v)
                else
                    y := max{h | (h,a,b) deleted from TSTACK }
                    let (h,a,b) be last triple deleted
                    TSTACK.push(max(y, w + ND(w) − 1), lowpt1(w), b)
                end
                TSTACK.push(EOS)
            end

            PathSearch(w)
            ESTACK.push(v → w)

            check for type-2 pairs
            check for a type-1 pair

            if e starts a path then
                remove all triples on TSTACK  down to and including EOS
            end
4.1         while (h,a,b) on TSTACK has a ≠ v and b ≠ v and high(v) > h do
                TSTACK.pop()
            od
        else
            let e = v ↪ w
            if e starts a path then
                pop all (h,a,b) with a > w from TSTACK
                if no triples deleted then
                    TSTACK.push(v, w, v)
                else
                    y := max{h | (h,a,b) deleted from TSTACK }
                    let (h,a,b) be last triple deleted
                    TSTACK.push(y, w, b)
                end
            end
            if w = parent(v) then
                C := new_component(e, w → v)
                e' := new_virtual_edge(w, v, C)
                make_tree_edge(e', w → v)
            else
                ESTACK.push(e)
            end
        end
    od
```

**Algorithm 5:** check for type-2 pairs

```
while v ≠ 1 and (((h,a,b) on TSTACK has a = v) or (deg(w) = 2 and
firstChild(w) > w)) do
    if a = v and parent(b) = a then
        TSTACK.pop()
    else
        e_ab := nil
        if deg(w) = 2 and firstChild(w) > w then
            C := new_component()
            remove top edges (v,w) and (w,b) from ESTACK and add to C
            e' := new_virtual_edge(v,x,C)
            if ESTACK.top() = (v,b) then e_ab := ESTACK.pop()
        else
            (h,a,b) := TSTACK.pop()
            C := new_component()
            while (x,y) on ESTACK has a ≤ x ≤ h and a ≤ y ≤ h do
                if (x,y) = (a,b) then e_ab := ESTACK.pop()
                else C := C ∪ { ESTACK.pop() }
            od
            e' := new_virtual_edge(a,b,C)
        end
        if e_ab ≠ nil then
            C := new_component(e_ab,e')
            e' := new_virtual_edge(v,b,C)
        end
        ESTACK.push(e'); make_tree_edge(e', v → b); w := b
    end
od
```

**Algorithm 6:** check for a type-1 pair

```
6.1 if lowpt2(w) ≥ v and lowpt1(w) < v and (parent(v) ≠ 1 or v is adjacent to
    a not yet visited tree arc) then
        C := new_component()
        while (x,y) on ESTACK has w ≤ x < w+ND(w) or w ≤ y < w+ND(w)
        do
            C := C ∪ { ESTACK.pop() }
        od
        e' := new_virtual_edge(v, lowpt1(w), C)
        if ESTACK.top() = (v, lowpt1(w)) then
            C := new_component(ESTACK.pop(),e')
            e' := new_virtual_edge(v, lowpt1(w), C)
        end
        if lowpt1(w) ≠ parent(v) then
            ESTACK.push(e')
            make_tree_edge(e', lowpt1(w) → v)
        else
            C := new_component(e', lowpt1(w) → v)
            e' := new_virtual_edge(lowpt1(w), v, C)
            make_tree_edge(e', lowpt1(w) → v)
        end
    end
```

- palm tree $P$ este reprezentata de vectorii PARENT[$v$], TREE_ARC[$v$] (arcul de arbore care intra in $v$) si TYPE[$e$] (arc de arbore sau frond);
- valorile *lowpt1*($v$), *lowpt2*($v$) si *ND*($v$) sunt precalculate. Nu este nevoie sa le reactualizam;
- un vector DEGREE[$v$] contine gradul lui $v \in G_c$. Este reactualizat de fiecare data cand o muchie este adaugata sau scoasa din $G_c$;
- pentru a calcula *firstChild*($v$), vom reactualiza lista de adiacenta de fiecare data o muchie este adaugata sau scoasa din $G_c$;
- pentru a calcula *high*($v$), vom precalcula lista de frond $v_i \longrightarrow w$ care se termina in $w$ in ordinea in care sunt vizitate. Cand un frond este scos sau adaugat din $G_c$, lista respectiva este reactualizata;
- vom preclacula un vector START[$e$] care este *true* daca si numai daca $e$ incepe un drum;
- testul daca "$v$ este adiacent sau nu vizitat inca" de la linia 6.1 poate fi rezolvat pur si simplu numarand arcele adiacente vizitate din arbore.

## 5. Corectii la algoritmul lui Hopcroft si Tarjan

Procedura SPLIT din [15] nu imparte corect un graf in componente split. Vom rezuma schimbarile pe care le-am facut in algoritmul nostru:

- functia de sortare $\phi$ a fost modificata cum este descris in subsectiunea 4.2 pentru a satisface muchiile multiple
- crearea ultimei componente split (linia 3.1) lipsea;
- conditia de la linia 4.1 a fost schimbata. Conditia originala putea sa elimine triplete din TSTACK corespunzatoare perechilor de separare de tip 2 reale. O astfel de pereche de separare nu putea fi recunoscuta de procedura originala SPLIT;
- conditia de la linia 6.1 a fost modificata. Conditia originala putea sa identifice eronat perechi de separare dupa ce graful a fost modificat;
- reactualizarile pentru *firstChild*($v$) (care este *A1(v)* in [15]) si DEGREE($v$) nu erau suficiente
- *high(w)* ( care e HIGHPT($w$) in [15]) nu era reactualizat, ceea ce nu e corect. HIGHPT trebuie reactualizat dinamic, atunci cand $G_c$ este modificat. Am inlocuit HIGHPT($w$) printr-o lista de frond care se termina cu $w$, care este reactualizata pe masura ce $G_c$ este modificata.