ELSEVIER

Science of Computer Programming 25 (1995) 63–77

Science of Computer Programming

Systematic design of an algorithm for biconnected components

K. Madhukar[a], D. Pavan Kumar[a], C. Pandu Rangan[a,*], R. Sundar[b]

[a] *Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India*

[b] *Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India*

Received March 1993; revised April 1994

Communicated by M. Nivat

Abstract

In this paper we present a new linear algorithm for finding the biconnected components of an undirected simple graph. The presentation of this algorithm is done as an exercise in the use of modern principles and techniques for systematic development of algorithms.

1. Introduction

We present the development of a new linear algorithm for finding the blocks (both vertex and edge sets) of a graph. The traditional algorithm uses a stack to "pop out" the edges of a block. This takes $O(\#V + \#E)$ time to get the edge set and then getting the vertex set from the edge set takes $O(\#V * \#I)$ time, where $\#I$ is the number of blocks. Also it is usually presented in the program form and thereafter all operational details and corresponding properties are established. This is quite confusing and unintuitive. Here we make use of modern ideas for the development of the program and the proof. A prime principle is to structure an algorithm to reflect the theory upon which it is based [3]. We have built the algorithm in a top down fashion, and we have always presented the properties on which the next refinement was done, before performing the refinement.

This paper is organized as follows. Section 2 presents our notations along with some basic graph theoretic terms and conventions. Section 3 gives the general outline of the algorithm assuming no particular representation. Section 4 presents some desired

* Corresponding author. Email: rangan@iitm.ernet.in.

SSDI 0167-6423(95)00005-4

properties of the representation and refines the algorithm taking them into account. Section 5 shows how to change the graph into the one satisfying these properties. Section 6 discusses the computation of some of the associated functions. Section 7 finds the set of vertices for each block and Section 8 the set of edges. Section 9 gives the implementation details and analyses the time complexity. Section 10 concludes with a discussion.

2. Notation

The booleans *and*, *or* and *not* are represented by $\wedge$, $\vee$ and $\neg$ respectively. For specifiers/quantifiers we follow [2, pp. 64–69]. Commonly used quantifiers are the existential quantifier $\exists$, universal quantifier $\forall$, and the specifiers are the counting specifier #, the union specifier $\bigcup$, summation specifier $\sum$, and the maximum and minimum specifiers **MAX**, **MIN**. For clarity, we modify the notation for summation specifier and define the notation for union specifier analogous to the counting specifier. In general, for a quantifier/specifier Q, $Q(r : s : t)$ is an expression in which r generates the values operated upon by the quantifier/specifier, s the range and t the property to be satisfied. For presenting the details of the algorithm we follow [2].

Our algorithms manipulate arrays and sequences. A sequence $s = [s_0, s_1, \ldots, s_{\#s-1}]$ denotes a sequence of $\#s$ elements. An element of the array is referenced using $s.k$ instead of the usual $s[k]$. The notation $s.(h \ldots k)$ denotes the subsequence of s consisting of $s.h$ up to $s.k$; finally $s \bowtie t$ denotes the concatenation of the sequences s and t.

We deal with a finite undirected graph $G = (V, E)$ where $V := \{i \mid 0 \leqslant i \leqslant \#V - 1\}$ is the set of vertices and E is the set of edges. An edge is represented by an unordered pair (u, v) where u and v are the end vertices of the edge. A simple graph is one that has no self-loop, i.e. no edge of the form (u, u), and no multiple edges.

A graph $H = (V', E')$ is called a *subgraph* of G iff $V' \subseteq V$ and $E' \subseteq E''$ where $E'' = \{(u, v) \mid (u, v) \in E \wedge (u, v \in V')\}$. If $E' = E''$ then H is called the maximal subgraph of G with the node set V'. If $V = V'$ then H is called a spanning subgraph of G.

By a (simple) path we mean a sequence of vertices $p = (v_0, v_1, \ldots v_{\#p-1})$ such that each pair (v_i, v_{i+1}) is an edge of G and the first $\#p - 1$ vertices are distinct. A path of at least length two is a cycle if its first and last vertices are the same.

The predicate $(u, v) \in E$ is denoted by $u \smile v$ and its reflexive transitive closure $u \smile^* v$ denotes the existence of a path of length zero or more between u and v. The predicate $cyc(e_1, e_2)$ denotes the existence of cycles containing the edges e_1 and e_2.

G is said to be connected iff $(\forall u, v : (u, v) \in V : u \smile^* v)$. The connected components of G are the maximal connected subgraphs G. G is *k-connected* iff the deletion of any set of $k - 1$ or fewer nodes leaves G connected. A spanning tree of G is a tree with node set V.

A digraph $G = (V, E)$ is like an undirected graph, except that its edge set consists of ordered pairs of elements of V. The ordered pair $\overrightarrow{(u, v)}$ denotes an arc from u to v, with

u being called the tail of the arc and v the head. The indegree of a vertex is defined by

$$ind(v) = \#(u : u \in V : \overrightarrow{(u,v)} \in E)$$

and the outdegree is analogously defined as

$$outd(v) = \#(u : u \in V : \overrightarrow{(v,u)} \in E).$$

The underlying undirected graph of a directed graph is the graph obtained by including (u,v) as an edge whenever $\overrightarrow{(u,v)}$ or $\overrightarrow{(v,u)}$ is in the arc set. A directed path and a directed cycle are defined in a manner similar to undirected graphs. The predicate $u \xrightarrow{G}{}^{*} v$ indicates the existence of a directed path of length zero or more from u to v in G and $u \xrightarrow{G}{}^{+} v$ denotes the existence of path of length at least one. A directed tree $T = (V,E)$ is a digraph with a distinguished vertex r, called the *root* satisfying $(ind(r) = 0)$, and $(\forall v : v \in V - \{r\} : ind(v) = 1)$. For each arc $\overrightarrow{(u,v)}$ of a tree, u is called the parent of v and v the child of u. If $u \rightarrow^{*} v$ then u is an ancestor of v and v is a descendant of u. A node u is a leaf if $outd(u) = 0$.

3. The basic biconnectivity algorithm

In this section we define the terms *cut vertex* and *biconnectivity* and then present the basic algorithm for identifying all the biconnected components.

A vertex c is said to be a *cut vertex* of a graph iff

$$(\exists u, v : (u,v) \in V : (u \overset{G}{\smile}{}^{*} v) \wedge \neg(u \overset{G-\{c\}}{\smile}{}^{*} v))).$$

An undirected graph is said to be *biconnected* iff it has no cut vertices. Biconnected components (or *blocks*) are maximal biconnected subgraphs of the graph G. It can be shown that a graph is biconnected iff every pair of edges lie on a common cycle [4].

Define the relation R on the edge set E of G as follows

$$e_1 \, R \, e_2 \Leftrightarrow (e_1 = e_2) \vee cyc(e_1, e_2).$$

Each set in the partition of E induced by R defines a block of G [1].

Note that it is enough if we develop an algorithm for a connected graph as every graph is a union of connected subgraphs.

Specification 3.1.

INPUT: A connected graph $G = (V,E)$.

OUTPUT: Biconnected components of G.

A vertex can belong to more than one block whereas an edge belongs to exactly one. This suggests a rudimentary biconnectivity algorithm which uses the above fact to compute the vertex and edge sets of each block in different styles.

Algorithm 3.1 (*Find_Blocks*).
{ **precondn**: Connected graph $G = (V, E)$ }
begin
(1) Find the unique representative edge for each block
and use it to name the block.
(2) Compute the vertex set for each block.
(3) Compute the edge set for all the blocks, by partitioning E.
end
{ **postcondn**: Blocks of G are listed }

4. Defining suitable representation of the graph

Algorithm 3.1 contains an operation where we have a choice. There may be several possible edges representing a block. We now define a representation of the graph where the choice is already made.

The original graph G was undirected. We now direct the edges, partition the edge set $E = T \cup F$ and renumber the vertices. The directed edges, the partition and the renumbered vertices must satisfy the following properties.

Property 4.1. $V = \{i \mid 0 \leqslant i < \#V\}$ *is a directed spanning tree.*

Property 4.2. $\overrightarrow{(u,v)} \in F \implies v \xrightarrow{T}{}^{*} u$.

Property 4.3. $\overrightarrow{(u,v)} \in T \implies u < v$.

As the conjunction of the above three properties is frequently used, we record it as Property 4.4.

Property 4.4. (*Property* 4.1) $\wedge$ (*Property* 4.2) $\wedge$ (*Property* 4.3).

A graph $D = (V, T, F)$ satisfying Properties 4.1 and 4.2 is called a palm graph. Edges in T are called *tree arcs* and those in F are called *fronds*. A *span-frond path* is a sequence of zero or more spanning tree edges (tree arcs) followed by exactly one frond.

Some useful functions of a palm graph are defined below.

4.1. The parent function $p(\cdot)$

The *parent* of a vertex u is defined as follows:

$$(p(v) = u) \Leftrightarrow \overrightarrow{(u,v)} \in T \tag{1}$$

Note that the parent of the root is not defined.

4.2. The child function ch(·)

$$ch.v = \bigcup(u : u \in V : \overrightarrow{(v,u)} \in T)$$

4.3. The lowest reachability function low(·)

$low(v)$ is defined as the minimum of v and w where w is the lowest numbered vertex that can be reached from v by a span-frond path. If there is no span-frond path starting from v then $low(v) = v$. In quantifier notation

$$low.v = \min(v, \mathbf{MIN}(u : u \in V : (\exists w : w \in V : v \xrightarrow{T}{}^{*}\ w \wedge \overrightarrow{(w,u)} \in F))) \tag{2}$$

From (2), by separating the cases for zero and for higher lengths in $v \xrightarrow{T}{}^{*}\ w$ we obtain

$$low.v = \min(v, \mathbf{MIN}(u : u \in V : \overrightarrow{(v,u)} \in F), \mathbf{MIN}(low.u : u \in V : \overrightarrow{(v,u)} \in T)) \tag{3}$$

4.4. Independent vertex

A vertex which satisfies $low(v) \geqslant p(v)$ is called an *independent* vertex. Note that the parent of an independent vertex is either the root or a cut vertex.

Theorem 4.1. *Every block B has exactly one independent vertex i such that* $(p(i), i)$ *is an edge of that block.*

Proof. *Claim*: Let c be the smallest vertex of the block B. The child of c that belongs to the block B is the required independent vertex i.

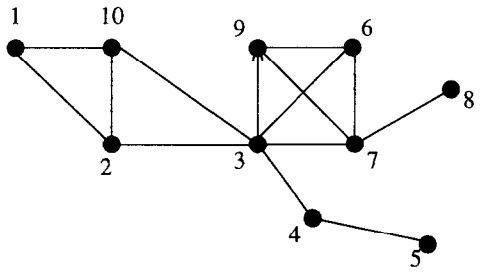
Proof of Claim: It is trivial to show that i is an independent vertex. We will use the next lemma to show that it is the *only* vertex satisfying the requirements of the theorem.

Lemma 4.1. $c = p(i)$ *has only one child i in the block B.*

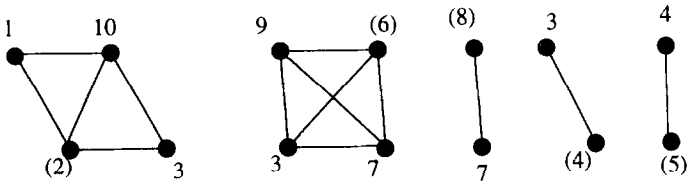
Proof. If $j \in ch(p(i))$ and $j \neq i$ then any path from i to j in the undirected graph has to pass through $p(i)$ (from Properties 4.1 and 4.2). There is no cycle containing i and j and hence they do not lie in one block [4]. Therefore $p(i)$ has only one child that belongs to the block B. This proves the lemma. □

If j ($\neq i$) is independent and $(p(j), j)$ is in the block B, then by the above lemma $p(j)$ is the descendant of i. Hence $p(j) \geqslant i$. We also have $low(j) \geqslant p(j)$. Hence $(p(i), i)$ and $(p(j), j)$ do not lie on the common cycle, contradicting the assumption that B is a block. This completes the proof of the claim and hence of the theorem. □

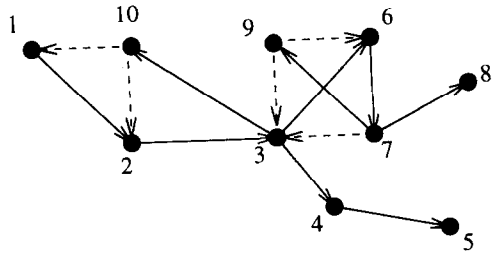
Corollary. *The number of blocks is equal to the number of independent vertices.*



(a) Graph G.



(b) The blocks of G.



(c) A palm graph representation of G. Dotted lines are *fronds* and continuous lines are *tree edges*.

(d) Values of some palm graph functions:

$I = \{2,4,5,6,8\}$,
$p = \{-,1,2,3,4,3,6,7,7,3\}$,
$low = (1,1,1,4,5,3,3,8,3,1)$,
$nd = (10,9,8,2,1,4,3,1,1,1)$.

(e) In Fig. (b), for each block, the vertex of I used to name it, is given within parentheses.

Fig. 1.

We may choose $(p(i), i)$ as the representative edge of the block to which it belongs. But as $p(i)$ is known once i is fixed we can address the block by the name i. Now the step 1 of algorithm 3.1 can be restated as "Find all independent vertices". For the graph in Fig. 1(a) all the blocks, along with the vertices used to address (represent) each block, are shown in Fig. 1(b).

5. Constructing the palm graph representation

It can be shown that the depth first search can be used to construct the palm graph efficiently. In this section we attempt to systematically develop a depth first search algorithm from the characterization of the palm tree. Since (V,T) should be a directed tree according to Property 4.1 the traversal must cover the vertex set, have a consistent way of directing the edges with respect to the direction of traversal and must avoid constructing cycles with T edges. One natural scheme is to mark the edges and vertices visited and point the edges in the direction of travel, so that the cycles can be prevented by avoiding revisits. Property 4.2 implies that we should not add a frond unless there is a tree path between the end points. This can be satisfied by continuing the travel from the last vertex visited until one encounters an already visited vertex, so that the path of traversing new vertices will provide the $v \xrightarrow{T}{}^{*} u$ path needed by the above condition. As long as the last visited vertex has some more unvisited neighbors, one of them can be chosen. But if all its neighbors are already visited, we may have to search for the previously visited vertices. We number the vertices in the order in which they are visited, observing that a vertex is first visited by a tree arc. We store this numbering in the array *pre*. We present the algorithm giving only the relevant steps and defer some implementation details (statements **A**–**E**) to later sections.

Algorithm 5.1 (*Construct_Palm_Tree*).
{ **precondn**: $G = (V,E)$ is an undirected graph }
var *pre*: **array** $[0 \ldots (\#V-1)]$ of $-1 \ldots \#V-1$;
for $i \in 0 \ldots \#V-1$ **do** $pre.v := -1$;
A (deferred to Section 6.1)
var v : vertex := *choose*(V);
var S : set of vertices := $\{v\}$;
$pre.v := 0$;
mark v visited;
var T, F : set of edges := ϕ, ϕ;
C (deferred to Section 9)
{ **invariant**: $S = \{v \mid v \in V \ \wedge$ *visited*$(v)\}$
$\wedge\ (\forall u, v : u, v \in S : pre.u < pre.v \Rightarrow u$ was visited before $v)$
$\wedge\ (\forall u, v : \overrightarrow{(u,v)} \in F \ \wedge\ u, v \in S : v \xrightarrow{T}{}^{*} u)$
$\wedge\ (S,T)$ is a directed tree $\wedge$ **H** (deferred to Section 9) }
do (while not all edges considered) $\rightarrow$
var $\overrightarrow{(v,u)}$: directed edge;
(1) $\overrightarrow{(v,u)}$:= "Choose an edge $\overrightarrow{(v,u)}$ such that v is the most recently visited vertex having an untraversed edge (u,v)";
if *visited*(u) $\rightarrow$
$F := F \cup \{\overrightarrow{(v,u)}\}$;
D (deferred to Section 9);

```
    [] unvisited(v)
       S := S ∪ {u};
       T := T ∪ {(v,u)→};
(2)    mark u visited;
       pre.u := #S − 1;
       B (deferred to Section 6.2)
       E (deferred to Section 9)
    fi
od
{ postcondn: P = (V,T,F) satisfies Properties 4.1 and 4.2
  ∧ (∀u,v : (u,v)→ ∈ E : (u,v)→ ∈ T ⇒ pre.u < pre.v) }
```

We now elaborate the step (1) of Algorithm 5.1 in more concrete terms while maintaining the invariant. The *pre* numbering suggests that the above step can be translated to "choose the highest *pre* numbered vertex which has an unvisited edge". In terms of predicates this is

$$\overrightarrow{(u,v)} = \overrightarrow{(r,q)} \quad \text{where } r \text{ is such that}$$

$$pre.r = \mathbf{MAX}(pre.p : p \in S \wedge (p,q) \in E : \neg(\overrightarrow{(p,q)} \in T \vee \overrightarrow{(p,q)} \in F \vee \overrightarrow{(q,p)} \in T \vee \overrightarrow{(q,p)} \in F))$$

and (r,q) satisfies

$$(r,q) \in E \wedge \neg(\overrightarrow{(r,q)} \in T \vee \overrightarrow{(r,q)} \in F \vee \overrightarrow{(q,r)} \in T \vee \overrightarrow{(q,r)} \in F) \qquad (4)$$

Note that the predicate *visited*(v) is the same as $pre.v \neq -1$. Hence step 2 of Algorithm 5.1 is redundant. The condition "not all edges considered" can be checked by $\#E > \#T + \#F$. Statements **C**, **D**, **E**, **H** concern the implementation details and are not required till Section 9. For the graph in Fig. 1(a), a palm graph representation is shown in Fig. 1(c).

6. Computing some palm graph functions

In this section we develop algorithms for the computation of palm graph functions defined in Section 4.

6.1. The parent function $p(\cdot)$

By definition, $(\forall u,v : u,v \in V : p.v = u \Leftrightarrow \overrightarrow{(u,v)} \in T)$. It suffices to update the parent array whenever a new edge is added to T, since an edge is never modified once it is added.

The following statements for **A** and **B** in Algorithm 5.1 will compute the parent array.

A: **var** p : **array**$[0 \ldots \#V-1]$**of** $-1 \ldots \#V-1$
for $i \in 0 \ldots \#V-1$ **do** $p.i := -1$;
B: $p.(pre.u) := pre.v$;

Once the *pre* numbers are obtained the vertices of the graph can be renumbered using the *pre* numbers as the new numbering. This ensures that Property 4.4 is satisfied. From now on we use the *pre* numbering for $P = (V,T,F)$. For the graph in Fig. 1(a), the original numbering of the vertices coincides with the *pre* numbering (Fig. 1(c)) and hence no renumbering is needed.

6.2. *The lowest reachability function low(·)*

We can compute *low*(·) using Eq. (3) making sure that the *low* values for all children of a vertex are evaluated before the vertex itself. This can be achieved by calculating *low* in the descending order of vertices.

Algorithm 6.1 (*Compute_low*).
{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4 }
var *low* : **array**$[0 \ldots \#V-1]$ **of** $0 \ldots \#V-1$;
for $i \in 0 \ldots \#V-1$ **do** $low.i := i$;
var w : vertex $:= \#V-1$;
{ **invariant**: $(\forall u : u > w : low.u$
$= \min(u, \mathbf{MIN}(v : v \in V : \overrightarrow{(u,v)} \in F), \mathbf{MIN}(low.v : v \in V : \overrightarrow{(u,v)} \in T)))$ }
do $w > 0 \rightarrow$
(1) $low.w := \min(w, \mathbf{MIN}(u : u \in V : \overrightarrow{(w,u)} \in F),$
$\mathbf{MIN}(low.u : u \in V : \overrightarrow{(w,u)} \in T))$;
$w := w-1$;
od
{ **postcondn**: $(\forall v : v \in V :$ Eq. (3)) }

Step 1 of the above algorithm needs elaboration. With the implementation scheme for constructing the palm tree, this step can be easily computed in multiple stages. We maintain the invariant that Eq. (3) is valid to the lowest-valued vertex already covered by the loop. The step can be replaced by

Refinement 6.1.
do$(u \in V \wedge \overrightarrow{(w,u)} \in F) \rightarrow$
if $u < low.w \rightarrow low.w := u$;
[] $u \geqslant low.w \rightarrow skip$;
fi
od
if $low.(p.w) > low.w \rightarrow low.(p.w) := low.w$;
[] $low.(p.w) \leqslant low.w \rightarrow skip$;
fi

The *parent* array and the *low* array for the palm graph in Fig. 1(c), are shown below Fig. 1(c).

6.3. Set of independent vertices I

By definition a vertex v is independent iff $low.v \geqslant p.v$. Hence the algorithm to compute I is straightforward.

Algorithm 6.2 (*Compute_Independent_vertex_set*).
{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4
$\wedge$ array p satisfies Eq. (1) $\wedge$ array *low* satisfies Eq. (3) }
var I : set of vertices := ϕ;
var v : vertex := 1;
{ **invariant**: $I = \{u \mid (u < v) \ \wedge \ (low.u \geqslant p.u)\}$ }
do $(v < \#V) \rightarrow$
  if $p.v \leqslant low.v \rightarrow I := I \cup \{v\}$;
  [] $p.v > low.v \rightarrow skip$;
  fi
  $v := v + 1$;
od
{ **postcondn**: $I = \{u \mid low.u \geqslant p.u\}$ }

7. Finding the vertex set for each block

We now refine the step 2 of Algorithm 3.1. Let $BV.i$ be the set of all vertices of the block defined by an independent vertex i.

It follows from definition that any member of $BV.i$ apart from $p.i$ is a descendant of i (Lemma 4.1). But not all descendants of i belong to BVi. The set $BV.i$ is given by

$$BV.i = \{p.i\} \cup \{u \mid (i \xrightarrow{T}{}^{*} u) \ \wedge \ (\forall j : (j \in I) \wedge (i \xrightarrow{T}{}^{+} j) : \neg(j \xrightarrow{T}{}^{*} u))\} \qquad (5)$$

For a vertex v, the set of its descendants $des(v) = \{u \mid v \xrightarrow{T}{}^{*} u\}$ and

$$\#des(v) = nd(v) \qquad (6)$$

The clause $i \xrightarrow{T}{}^{*} u$ can be translated to $u \in des(i)$. It follows from Property 4.1, Property 4.3 and Eq. 6 that $u \in des(i)$ is same as $i \leqslant u < i + nd(i)$.

Further the set explicitly excluded by the clause

$$(\forall j : (j \in I) \wedge (i \xrightarrow{T}{}^{+} j) : \neg(j \xrightarrow{T}{}^{*} u))$$

is the set $\bigcup(des.j : j \in I : i < j < i + nd.i)$. This gives the following algorithm for $BV.i$.

Algorithm 7.1 (*Compute_BV.i*).

{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4 $\wedge$ array p satisfies Eq. (1)
$\wedge$ array *low* satisfies Eq. (2) $\wedge$ $(i \in I) \wedge I = \{u \mid low.u \geqslant p.u\}$
$\wedge$ array *nd* satisfies Eq. (6) }

$BV.i := \{p.i\} \cup \{i\}$;
var v : vertex $:= i + 1$;
{ **invariant**:
$BV.i = \{p.i\} \cup \{u \mid (u < v) \wedge (i \xrightarrow{T}{}^{*}\, u) \wedge (\forall j : (j \in I) \wedge (i \xrightarrow{T}{}^{+}\, j) : \neg(j \xrightarrow{T}{}^{*}\, u))\}$ }
do $(v < i + nd.i) \rightarrow$
if $v \in I \rightarrow v := v + nd.v$;
[] $v \notin I \rightarrow BV.i := BV.i \cup \{v\}$;
$v := v + 1$;
fi
od
{ **postcondn**: Eq. (5) }

It remains to show how to find $BV.i$ for all $i \in I$.

Algorithm 7.2 (*Compute_BV*).

{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4 $\wedge$ $I = \{u \mid low.u \geqslant p.u\}$
$\wedge$ arrays *low* and p satisfy Eqs. (3) and (1) respectively }
var BV : **array**$[0 \ldots \#V - 1]$ of set of vertices;
for $i \in 0 \ldots \#V - 1$ **do** $\rightarrow BV.i := \phi$;
var v : vertex $:= 0$;
{ **invariant**: $\forall(i : (i < v) \wedge (i \in I) :$ Eq. (5)) $\wedge$ $\forall(i : (i \geqslant v) \vee (i \notin I) : BV.i = \phi)$ }
do $(v < \#V) \rightarrow$
if $v \in I \rightarrow$ *Compute_BV.v*;
[] $v \notin I \rightarrow$ *skip*;
fi
$v := v + 1$;
od
{ **postcondn**: $(\forall i : i \in I :$ Eq. (5)) $\wedge$ $(\forall i : i \notin I : BV.i = \phi)$ }

It remains to compute array *nd*. We have the recursive definition

$$des.v = \{v\} \cup \bigcup(des.u : u \in ch.v : true)$$

Hence

$$nd.v = 1 + \sum(nd.u : u \in ch.v : true) \tag{7}$$

We need to compute the *nd* values of the children before the parent. The summation over the children set is avoided by making each child contribute to its parent. Initially the *nd* value for each vertex is set to 1, as every vertex is a descendant of itself.

Algorithm 7.3 (*Compute_no._of_descendants*).
{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4 $\wedge$ array p satisfies Eq. (1) }
var nd : **array**$[0 \ldots \#V-1]$ **of** $1 \ldots \#V$;
for $(i \in 0 \ldots \#V-1)$ **do** $\rightarrow$ $nd.i := 1$;
var v : vertex $:= \#V-1$;
{ **invariant**: $(\forall u : u \in V : nd.u = 1 + \sum(nd.w : (w \in ch.u) : (w > v)))$ }
do $(v > 0)$ $\rightarrow$
  $nd.(p.v) := nd.(p.v) + nd.v$;
  $v := v - 1$;
od
{ **postcondn**: $(\forall v : v \in V :$ Eq. (7)) }

8. Finding the edge set for all the blocks

Now we refine the step 3 of Algorithm 3.1. Let $BE.i$ be the set of edges corresponding to block i. Recall that an edge can belong to only one block. $BE.i$ includes tree edges from $\overrightarrow{(p.i,i)}$ until the set of parents of independent vertices reachable from i (i.e $DPI.i$). Fronds in $BE.i$ would start at or above $DPI.i$ and end at or before $p.i$. Hence

$$BE.i = \{(u,v) \mid (\overrightarrow{(u,v)} \in T \wedge v \in BV.i - \{p.i\}) \vee (\overrightarrow{(u,v)} \in F \wedge u \in BV.i - \{p.i\})\} \quad (8)$$

The following algorithm computes $BE.i$ using the above equation.

Algorithm 8.1 (*Compute_BE*).
{ **precondn**: $P = (V,T,F)$ satisfies Property 4.4
  $\wedge$ arrays p and *low* satisfy Eqs. (1) and (2) respectively
  $\wedge$ $I = \{u \mid low.u \geqslant p.u\}$ }
var BE : **array**$[0 \ldots \#V-1]$ **of** set of edges;
for $(i \in 0 \ldots \#V-1)$ **do** $\rightarrow$ $BE.i := \phi$;
var X : set of edges $:= E$;
{ **invariant**: $(\forall i : i \in I : BE.i = \{(u,v) \in E - X \mid (\overrightarrow{(u,v)} \in T \wedge v \in BV.i - \{p.i\})$
  $\vee (\overrightarrow{(u,v)} \in F \wedge u \in BV.i - \{p.i\})\}) \wedge (\forall i : i \notin I : BE.i = \phi)$ }
do $(X \neq \phi)$ $\rightarrow$
  var (u,v) : edge $:= choose(X)$;
  $X := X - \{(u,v)\}$;
(1) "Update the edge set of the block to which (u,v) belongs"
od
{ **postcondn**: $(\forall i : i \in I :$ Eq. (8)) $\wedge$ $(\forall i : i \notin I : BE.i = \phi)$ }

We need to elaborate the statement (1) of the above algorithm. From Eq. (8), for $\overrightarrow{(u,v)} \in T$ ownership is decided by v and for $\overrightarrow{(u,v)} \in F$ ownership is decided by u.

Hence ownership is always decided by the higher vertex of an edge. This suggests us to define a vertex to vertex function $io(\cdot)$ (called *owner independence vertex function*) as follows:

$$(io.v = i) \Leftrightarrow \forall(u : (u,v) \in E \wedge v > u : (u,v) \in BE.i) \tag{9}$$

From this and Eq. (8) it follows that

$$io.v = \begin{cases} io.(p.v) & \text{if } v \text{ is not independent} \\ v & \text{if } v \text{ is independent} \end{cases} \tag{10}$$

Note that io is undefined for the root. But this is of no significance as the root can never be the higher numbered vertex of any edge. Assuming the availability of function io we can replace the statement "Update ... belongs" of Algorithm 8.1 by the statement

$$BE.(io.\max(u,v)) = BE.(io.\max(u,v)) \cup \{(u,v)\} \tag{11}$$

The function io can be computed using Eq. (10) as shown below.

Algorithm 8.2 (*Compute_Independent_Owner*).
{ **precondn**: same as precondn of Algorithm 8.1 }
var io : **array**$[0 \ldots \#V-1]$ **of** $0 \ldots \#V-1$;
for $(i \in 0 \ldots \#V-1)$ **do** $\rightarrow$ $io.i := i$;
var v : vertex := 1;
{ **invariant**: $(\forall u : 0 < u < v : io.u$ satisfies Eq. (9)) }
do $(v < \#V) \rightarrow$
  if $v \in I \rightarrow$ *skip*;
  [] $v \notin I \rightarrow$ $io.v := io.(p.v)$;
  fi
  $v := v + 1$;
od
{ **postcondn**: $(\forall u : 0 < u < \#V : io.u$ satisfies Eq. (9)) }

9. Implementation and complexity

9.1. Constructing the palm graph

Assume an adjacency list representation of $G = (V,E)$, in which $A.v$ is the sequence of vertices adjacent to a vertex v. The edge (u,v) will appear as an entry of u in $A.v$ and as v in $A.u$. The only nontrivial part of the Algorithm 5.1 is step involving Eq. (4). The first term and the **MAX** quantifier require that the adjacency list of the most recently visited vertex be given preference. This can be done by maintaining a sequence of tuples of vertices sorted in the reverse order of visiting. After a vertex u is visited for the first time all the tuples (u,v) where v is in the adjacency list of u are

concatenated to the beginning of the sequence. For each tuple (u,v) in the sequence, $u \in S$ and neither $\overrightarrow{(u,v)} \in T$ nor $\overrightarrow{(u,v)} \in F$. Hence for finding the required untraversed edge we need to delete those tuples (u,v) for which either $\overrightarrow{(v,u)} \in T$ or $\overrightarrow{(v,u)} \in F$. The former can be checked by $p.(pre.u) = pre.v$ (as $u \in S$, $p.(pre.u)$ is known). If u is the root, $\overrightarrow{(u,v)}$ already satisfies the selection clause for all v. The implementation details of Algorithm 5.1 are taken care of by the following statements.

C: **var** STK : sequence of directed edges := ϕ;
  for $u \in A.v$ **do** $\rightarrow$ $STK := \overrightarrow{(v,u)} \bowtie STK$;
D: $STK := STK.(1 \ldots \#STK - 1)$;
E: $STK := STK.(1 \ldots \#STK - 1)$;
  for $u \in A.v$ **do** $\rightarrow$ $STK := \overrightarrow{(v,u)} \bowtie STK$;

Step 1 of Algorithm 5.1 can be implemented as

do $(pre.u > pre.v) \wedge (pre.v \neq 0$ **cand** $p.pre.v = pre.u) \rightarrow$
  $STK := STK.(1 \ldots \#STK - 1)$;
  $\overrightarrow{(v,u)} := STK.0$;
od

With all these additions we can extend the invariant of Algorithm 5.1 by setting H to

$$(\forall i,j : 0 \leqslant i < j \leqslant \#STK - 1 : (pre.((STK.i).1) \geqslant pre.((STK.j).1)))$$

In all our algorithms sets can be implemented as sequences. As we perform only disjoint unions, the union operation on the sets can be translated to concatenation of sequences. The initializations in Algorithm 5.1 take $O(\#V)$ time and the loop processes each edge twice giving a time complexity of $O(\#V + \#E)$.

9.2. *Compute_low*

In the refinement of Algorithm 6.1, the check can be done by looking into $A.u$ and checking whether $w > u$ where w is in $A.u$. The initialization takes $O(\#V)$ time and the total number of frond checks take $O(\#E)$ time. Hence the time complexity is $O(\#V + \#E)$.

9.3. *Set_of_independent_vertices*

The loop in Algorithm 6.2 processes all the vertices once. Hence, assuming the availability of *low* and p, this takes $O(\#V)$ time.

9.4. *Compute_BV*

In Algorithm 7.2, the initialization takes $O(\#V)$ time. The check $v \in I$ can be translated to $low.v \geqslant p.v$. Algorithm 7.1 is called $\#I$ times. Assuming the availability of *nd* and accounting for all these calls together it can be verified that the statement

$v := v + nd.v$ in Algorithm 7.1 is executed $\#I$ times and the statements updating $BV.i$ are executed $(\#V + \#I)$ times.

In Algorithm 7.3, initialization takes $O(\#V)$ time and the loop runs over the set of vertices. Hence $nd(\cdot)$ can be computed in $O(\#V)$ time. Thus the vertex sets of all the blocks can be computed in $O(\#V + \#I)$ time.

9.5. *Compute_BE*

In Algorithm 8.1, using Eq. (11) and assuming the availability of *io* array, the initialization takes $O(\#V)$ time and the loop takes $O(\#E)$ time. For computing *io*, Algorithm 8.2 takes $O(\#V)$ time as both the initialization and the loop take $O(\#V)$ time each. Hence Algorithm 8.1 has a time complexity of $O(\#V + \#E)$.

With all these refinements our main Algorithm 3.1 takes $O(\#V + \#E)$ time.

10. Conclusions

We have presented the development of a linear algorithm for finding the biconnected components (both vertex and edge sets) of an undirected simple graph. Our development followed the well known principle of programming that proof of correctness and program should go hand in hand. We firmly believe that efforts like ours and the one in [3] will go a long way in giving a better understanding of the intricacies of algorithmic graph theory.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms* (Addison Wesley, Reading, MA, 1984).

[2] G. Dromey, *Program Derivation* (Addison Wesley, Reading, MA, 1989).

[3] D. Gries and J. Xue, The Hopcroft Tarjan Planarity Algorithm — Presentations and Improvements, Cornell University, Tech. Report, 88-906, 1988.

[4] F. Harary, *Graph Theory* (Addison Wesley, Reading, MA, 1969).