2009

A Distributed Algorithm for Finding Separation Pairs in a Computer Network

Katayoon Moazzami
University of Windsor

A DISTRIBUTED ALGORITHM FOR FINDING SEPARATION PAIRS IN A COMPUTER NETWORK

by

KATAYOON MOAZZAMI

A Thesis

Submitted to the Faculty of Graduate Studies

through Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2009

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

One of the main problems in graph theory is the question of graph connectivity. Graph connectivity often arises in problems related to network reliability. Graph connectivity can be studied from two different aspects, namely, vertex-connectivity and edge-connectivity. Vertex connectivity is the smallest number of vertices whose deletion will cause a connected graph to be disconnected. The focus of this work is on finding separation pairs of a graph which is the set of pairs of vertices that deleting them would disconnect a graph. Finding separation pairs can be used in solving the problem of vertex-connectivity and finding the triconnected components of the graph. There have been different algorithms presented during the past which are non-linear or if linear, very complicated. This work is based on the work of Tarjan and Hopcroft which finds the separation pairs in linear time, that is achieved by the use of Depth-First Search (DFS). Our goal is to present an algorithm that finds the separation pairs in an asynchronous distributed computer network using distributed Depth-First search (DDFS).

Dedication

I would like to dedicate this thesis to my wonderful parents and kind friends.

Acknowledgements

I am very grateful to my supervisor, Dr. Y.H. Tsin for his valuable guidance during my thesis work and through completion of my degree requirements. Without his support, this thesis could not have been completed. I also would like to thank members of my master committee, Dr. Z. Kobti, Dr. G. Lan for their helpful advice and guidance.

I acknowledge the financial support from my supervisor, Dr. Y.H. Tsin, in the form of research assistantship through NSERC; from the School of Computer Science, in the form of graduate assistantship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Graph connectivity has always been one of the most important problems explored in graph theory. Some of the most common uses of graph connectivity are in communication networks like telephone networks where the reliability of the network can be determined. Other applications are in irreducibility analysis of Feynman diagrams in quantum physics and chemistry, scheduling problem in operational research and circuit-layout problems.

A graph is *k-vertex-connected* (*k-edge-connected*, respectively) if removing less than k vertices (edges, respectively) will not result in a disconnected graph. Clearly, we are interested in the cases where $k \geq 2$. A 2-*vertex-connected* graph is commonly called a *biconnected* graph while a 2-*edge-connected* graph is commonly called a *bridge-connected* graph. A 3-*vertex-connected* graph is also called a *triconnected* graph.

Despite the fact that a tremendous amount of effort had been put into the studies of the graph connectivity problems, efficient algorithms are known only for $k \leq 4$.

For $k = 2$, the first biconnectivity algorithm was presented by Tarjan[42] The algorithm runs on the RAM (the standard sequential computer model) and takes $O(|V| + |E|)$ time and space for a connected graph $G = (V, E)$. The algorithm also solves the brige-connectivity problem. Later, Hopcroft and Tarjan presented an optimal algorithm for triconnectivity[20]. Both algorithms are based on a

graph traversal technique called *depth-first search*. Later, Gabow revisited depth-first search from a different perspective - the path-based view - and presented new depth-first-search-based optimal algorithm for biconnectivity and bridge-connectivity[16]. Galil and Italiano[17] presented the first 3-edge-connectivity algorithm. Their method is to use reduction to transform the 3-edge-connectivity problem to the 3-vertex-connectivity problem and then use the triconnectivity algorithm of Hopcroft el al. to solve the problem. The algorithm runs in optimal time and space. Owing to the fact that the algorithm of Galil et al. is rather complicated, a few simpler optimal algorithms were then reported by Taoka et al.[41], Nagamochi and Ibaraki[35] and Tsin[44],[48].

Efficient algorithms for other computer models have also been proposed. For the PRAM (the parallel RAM) model, for the biconnectivity problem, Tsin and Chin [49] presented an algorithm that runs on the CREW (concurrent-read-exclusive-write) PRAM. The algorithm takes $O(\log^2(|V|))$ time using $O(|V|^2/\log^2(|V|))$ processors (Note: log denotes $\log_2$) which is optimal for dense graphs. Tarjan and Vishkin[43] developed an algorithm for the CRCW (concurrent-read-concurrent-write) PRAM that takes $O(\log(|V|))$ time using $O(|V|+|E|)$ processors. For triconnectivity, a number of algorithms had been reported[37, 33,14] and all are very complicated. Each of them gives rise to a complicated non-optimal sequential algorithm. For 4-vertex-connectivity, Kanevsky and Ramachandran presented an algorithm for finding all the separating triples (sets of *three* vertices whose removal results in a disconnected graph)[22]. The algorithm runs on the *Arbitrary* CRCW PRAM in $O(\log^2 n)$ time using $O(n^2)$ processors. It also gives rise to a sequential algorithm that runs in $O(n^2)$ time and space.

On the External-Memory model, Chiang et.al. presented the first I/O-efficient biconnected component algorithm[10]. The algorithm is an adaption of the biconnected component PRAM algorithm of Tarjan et.al[49] based on simulation. It performs $O(min\{sort(V^2), \log(V/M)sort(E)\})$ I/Os. Using an External-Memory connected component algorithm of Munagala et al[34], Meyer showed that the I/O bound of the biconnected-component algorithm of Chiang et. al. can be improved to $O((E/V)sort(V)\cdot\max\{1,\log\log(|V|DB/|E|)\})$. Note that $(|E|/|V|\cdot sort(|V|)) = \Theta(sort(|E|))$.

On the distributed computer model, a number of algorithms that run in $O(|V|)$ time and transmits $O(|E|)$ messages of $O(\log|V|)$ length had been proposed[4,12,29,39]. In the fault-tolerance setting, with the assumption that a breadth-first search or depth-first search spanning tree is available, Karaata[26] presented a self-stabilizing algorithm that finds all the biconnected components in $O(d)$

rounds (d is the diameter of the graph) using $O(|V|\triangle \log \triangle)$ bits per processor; Devismes [13] improved the bounds to $O(H)$ moves (H is the height of the spanning tree) and $O(|V| \log \triangle)$ bits per processor, and Tsin[44] further improved the result to $O(dn \log \triangle)$ rounds and $O(|V| \log \triangle)$ bits per processor *without* assuming the existence of any spanning tree.For 3-edge-connectivity, Saifullah and Tsin[46] presented a self-stabilizing algorithm that can identify all the 3-edge-connected components in $O(H)$ rounds or $O(|V|H)$ moves using $O(|V|H \log \Delta)$ bits per processor, where H is the height of the depth-first search tree constructed by the algorithm and Δ is the largest degree of a node in the network.

In wireless sensor network, Turau[50] presented an algorithm that takes $O(|V|)$ time and transmits at most $4m$ messages for the biconnectivity problem. No algorithm has been for proposed for 3-connectivity or higher.

1.1.1 Motivation

A distributed computer/communication network can be modeled as a graph $G = (V, E)$ such that V represents the set of notes (which are computers) and E represents the communication links between the nodes. The vertex- (edge-,respectively) connectivity of G represents the reliability of the network in the presence of node or link failures. Specifically, a k-vertex-connected (k-edge-connected, respectively) network will remain connected in the presence of $k-1$ or less node (link, respectively) failures.

As pointed out earlier, the 2-vertex- and 2-edge-connectivity problems have been studied extensively in the distributed setting. A number of efficient distributed algorithms have been proposed. For 3-edge-connectivity, Jennings et al.[21] presented a distributed algorithm that requires $O(|V|^3)$ time and messages. Tsin[45] improved the time and message bounds by presenting a distributed algorithm that takes $O(|V| + H_T^2)$ time and transmitting $O(|E| + |V|H_T)$ messages, where $H_T (< |V|)$ is the height of a depth-first spanning tree of the network. In the worst case, the algorithm requires only $O(|V|^2)$ time and messages.

For 3-vertex-connectivity, no distributed algorithm has been reported so far. Our objective is to develop an efficient distributed algorithm for triconnectivity to fill the void. We shall develop our algorithm using the ideas underlying the sequential algorithm of Hopcroft et al.

1.1.2 Previous related works

For the triconnectivity problem, the only known algorithm with linear-time complexity is the aforementioned depth-first-search-based algorithm of Hopcroft and Tarjan. Unfortunately, this algorithm is rather complicated and its presentation contains quite a number of serious, although amendable, errors which make the paper quite difficult to comprehend. In 2001, Gutwenger and Mutzel[18] wrote a paper on SPQR-trees in which they revealed the errors in the algorithm of Hopcroft el al. and fixed them. Gutwenger and Mutzel were motivated by the fact that the SPQR-tree is a data structure that can be used to represent the decomposition of a biconnected graph into its triconnected components. They had also implemented the algorithm and claimed that it was the only implementation that correctly implemented the triconnectivity algorithm of Hopcroft et al. in linear time.

In the triconnectivity algorithm of Hopcroft and Tarjan, depth-first search plays a central role. Therefore, if we were to develop a distributed algorithm for triconnectivity based on their algorithm, we would have to had an algorithm to traverse the network in a depth-first search manner. Fortunately, there have been quite a number of distributed depth-first search algorithms published in the literature.

Cheung[9] reported the first distributed depth-first search algorithm. The algorithm runs in $2|E|$ time units and transmit $2|E|$ messages. Awerbuch[4] presented an algorithm that has an order of magnitude improvement in time complexity over that of Cheung but transmits twice as many messages as Cheung. Specifically, his algorithm runs in at most $4|V|$ time units (note that $|V|-1 \leq |E| \leq O(|V|^2)$) and transmits $4|E|$ messages. Moreover, his algorithm does not require the messages received at each node to be processed in the order they arrived. Lakshmanan et al.[29] presented an algorithm that improved both the time and message bounds to $2|V|-2$ time units and at most $4|E|-(|V|-1)$ messages, respectively. However, their algorithm requires that the messages received at each node to be processed in the order they arrived. Soon after that, Cidon[12] presented an algorithm that takes $2|V|-2$ time units and transmits at most $3|E|$ messages. Moreover, his algorithm also does not require the messages received at each node to be processed in the order they arrived. Tsin[47] showed that Cidon's algorithm does not always work correctly by presenting two counterexamples. He then showed that the errors in Cidon's algorithm can be fixed but the resulting algorithm will

have time and message bounds same as those of Lakshmanan et al. Tsin also showed that by using the dynamic backtracking technique of Makki et al.[31], the time bound of the corrected Cidon's algorithm can be improved to $n(1+r)$, where $0 \leq r < 1$.

Sharma et al[39] showed that if the message length is allowed to be $O(|V|)$ rather than $O(1)$, both the time and messages bounds of distributed depth-first search can be reduced to $2|V|-2$.

1.1.3 Contribution of this thesis

To the best of our knowledge, there has been no distributed algorithm presented for triconnectivity. We shall present an efficient distributed algorithm that identifies a subsets of separation pairs based on which the network can be decomposed into split components (essentially triconnected sub-networks of the given network). For a network $G = (V,E)$, our algorithm runs in $(|V|+H^2)$ time and transmits $O(|E|+|V|\cdot H)$ messages, where $H(<|V|)$ is the height of the depth-first search tree constructed by the algorithm. In the worst case (i.e. when $H = O(|V|)$), our algorithm takes $O(|V|^2)$ time and transmit $O(|V|^2)$ messages. This algorithm is the first distributed algorithm for triconnectivity.

Considering the example of a telephone or computer network (where phones or computers are the nodes of the graph and links are the edges), our algorithm can be used to determine how reliable the network is should there be one or two node-failures occur simultaneously in the network.

1.2 Organization of this thesis

In Chapter 2, some basic definitions in graph theory are given. A brief description of the depth-first search technique is also given. In Chapter 3, the triconnectivity algorithm of Hopcroft et al. is explained. In Chapter 4, the distributed model for which our algorithm is designed is described. In Chapter 5, our distributed algorithm for finding separation pairs is presented. The correctness proof of the algorithm as well as time and message complexities analysis are given. In Chapter 6, we make some concluding remarks and discuss some possible future work.

Chapter 2

Definitions

In order to be able to fully understand the algorithm we will present, we shall first familiarize the reader with graph theory terminologies and definitions. The following are some of these definitions that would be used in the following chapters.

2.1 Basic Definitions

A graph $G = (V, E)$ will be represented by its vertex set V and its edge set E.

Edge

Every element in E is an edge in G. Every edge is determined by two vertices. If the two vertices are v and w, then the edge is denoted by (v, w).

Adjacent

If there is an edge (v, w) in the graph G, then v and w are adjacent to each other.

Incident to

Edge (v, w) is incident to vertices v and w. Vertices v and w are incident to the edge (v, w).

Degree

A vertex v in the graph G has the degree k (denoted by $deg(v)$) if there are k edges incident to it in G.

Directed Edges

If the (v,w) in graph G is an ordered pair, then the edge is a directed edge and is said to be directed from v to w.

Undirected Edges

If the (v,w) in graph G is an unordered pair, then the edge is an undirected edge.

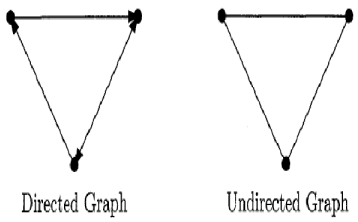

Figure 2.1: Directed and undirected graphs

Directed Graph

A directed graph is a graph in which every edge is a directed edge.

Undirected Graph

A undirected graph is a graph in which every edge is an undirected edge.

Multigraph

A multigraph is a graph in which there exists two vertices having two or more edges incident upon both of them.

Head And Tail Of The Edge

Let (v,w) is a directed edge, v is called the tail and w is called the head of the edge.

Path

Let G is a multigraph, a path in G $v \rightarrow^* w$ is a sequence of vertices and edges leading from v to w. A path is simple if all its vertices are distinct. Vertices v and w are called the terminating vertices of the path.

Cycle

A path $v \rightarrow^* w$ is a cycle if all its edges are distinct and the only vertex that occurs exactly twice on the path is v.

Connected Graph

An undirected multigraph is connected, if every pair of vertices v and w in G is connected by a path.

Subgraph

If $G(V,E)$ and $G\prime = (V',E')$ are two multigraphs such that $V \subseteq V'$ and $E \subseteq E'$ then G' is a subgraph of G.

Bond

A multigraph having exactly two vertices v,w and one or more edges (v,w) is called a bond.

Directed Tree

A directed tree is a directed graph whose undirected version is connected and has a vertex, called the root, which is the head of no edges and all vertices except the root are the head of exactly one edge.

Spanning Tree

If G is a multigraph, a tree T is a spanning tree of G, if T is a subgraph of G and T contains all the vertices of G.

Biconnected Graph

A connected multigraph G is biconnected if for each triple of distinct vertices v, w and a in V, there is a path $P : v \to w$ such that a is not on the path P.

Separation Point

If there is a distinct triple v, w, a in graph G such that a is on every path $v \to w$, a is called a Separation point (or an articulation point) of G.

Separation Pair

Let a, b be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1, E_2, ..., E_n$ such that two edges which lie on a common path do not contain any vertex of a, b except as an endpoint are in the same class. These classes are called the separation classes of G with respect to a, b. If there are at least two separation classes, then a, b is a separation pair of G unless:

(i) there are exactly two separation classes, and one class consists of a single edge, or

(ii) there are exactly three separation classes, each consisting of a single edge. In much simpler words, a separation pair is a pair of vertices that their elimination will disconnect a biconneted graph.

Triconnected Graph

If G is a biconnected multigraph such that it does not contain a separation pair, then G is triconnected.

Adjacency List

In order to show the relation between the edges and vertices in a graph we can use an adjacency list. This list shows which vertices are adjacent to a given vertex v.

A graph and its adjacency list are shown as an example In Figure below.

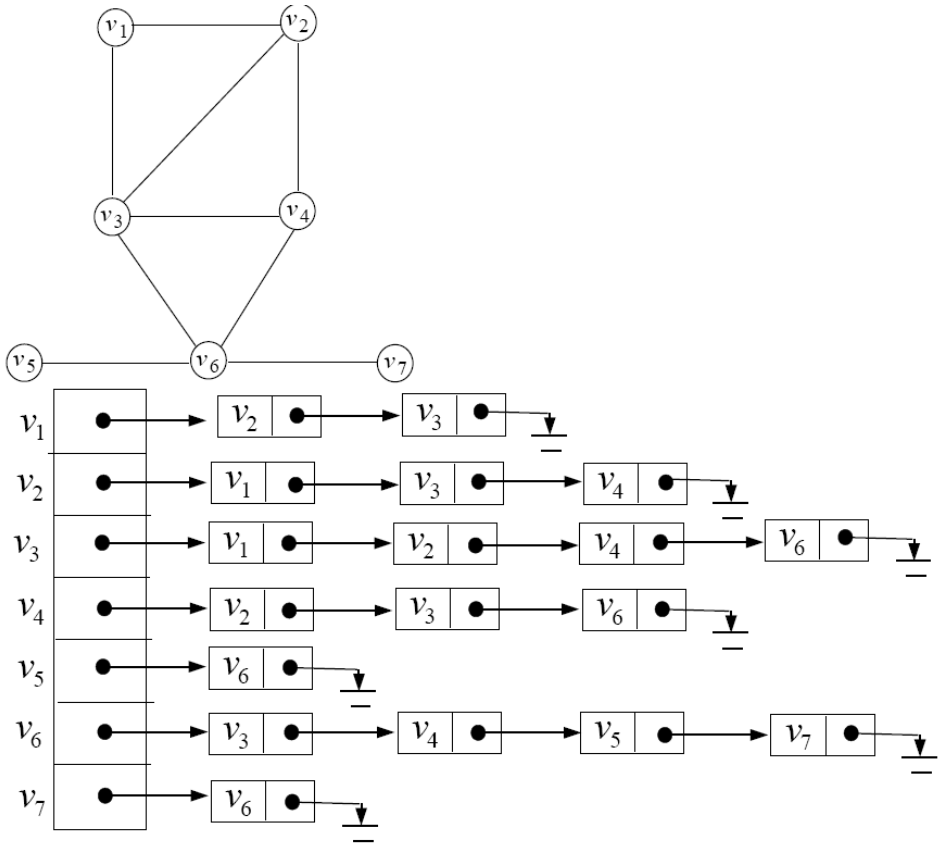

Figure 2.2: A graph and its adjacency list

2.2 Depth-first Search

Depth-first search is a systematic method for exploring a multigraph[42]. This search can be used for numbering the vertices in the graph and also determining a spanning tree of a graph.

In this search, first we assume that all the vertices are in the *unvisited* state. We start from an arbitrary vertex, call it the root, and look at its adjacency list for unvisited vertices. If there are unvisited vertices, we will arbitrarily choose one and move to that vertex. We continue the same process till the time there are no unvisited vertices, then the search will backtrack. In depth-first search, each edge is examined exactly twice and each vertex is visited once. If the graph is an undirected graph, then by using depth-first search we can impose an order on its vertices (according to the depth-first

search numbers) and hence transform it to a directed graph.

The search can be presented as a recursive function as shown below:

DFS(v)

begin

visit vertex v and mark it as *visited*

choose an *unvisited* vertex w in $A[v]$ (the adjacency list of v)

invoke DFS(w)

end

It is well-known that depth-first search runs in $O(|V|+|E|)$ time, where E is the edge set and V is the vertex set of the graph[42].

The following figures shows how a graph is traversed by DFS.

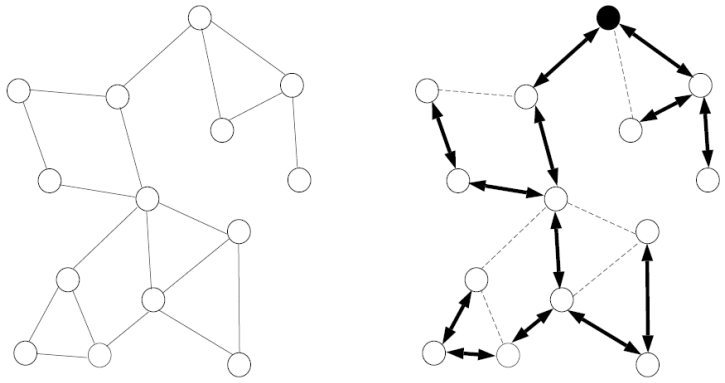

Figure 2.3: DFS traversal

As noted before, DFS can be used for constructing a spanning tree of a graph. After DFS is applied, the edges that belong to the spanning tree are called the *tree-edges* and the rest of the edges are called the *fronds*. The difference between the tree edges and the fronds is that, if during the search we visit a vertex w in $A[v]$ and w is not visited the edge (v,w) will become a tree-edge and if w is already visited, the edge (v,w) will become a frond.

2.2.1 Some Definitions Related To DFS

DFS Tree

The spanning tree constructed by the depth-first search traversal of the graph is called a DFS tree of the graph.

DFS Number

The number assigned to a vertex during a depth-first search is called the *depth-first search number* of that vertex and is denoted by *dfs(v)*.

Child

In the DFS tree, if (v,w) theis a tree edge then v is the parent of the w and w is a child of v.

Leaf

If a vertex in the DFS tree of graph G does not have any children, it is a leaf.

Ancestor and descendant

In a DFS tree, if $dfs(v) < dfs(w)$ then v is an ancestor of w and w is a decendant of the v. Each node is its own descendant and ancestor. The number of descendants of the node v is denoted by $nd(v)$.

Subtree

In the DFS tree T, a subtree rooted in w, denoted by $T(w)$, is the largest subgraph of T (which is a tree) whose root is w.

Incoming and Outgoing frond

If (v,w) is a frond (also called back-edge) then it is an outgoing frond of v and an incoming frond of w.

Chapter 3

Sequential Algorithm

3.1 The Sequential Algorithm Of Hopcroft And Tarjan

In 1973, Tarjan and Hopcoft presented a sequential algorithm for finding the triconnected componenets of a biconnected graph. The algorithm is based on depth-first search and makes several passes over the input graph. Since we are focusing our algorithm on finding separation pairs, there will be certain parts of their algorithm that we will not discuss in this thesis. However, we will explain the sequential algorithm of Tarjan and Hopcroft to facilitate the understanding of our distributed algorithm in the next chapters.
The algorithm basically consists of the following steps (these steps will be discussed in more detail in the following sections):

1. The input graph G is split into a set of triple bonds and a graph G'.

2. If G' is already biconnected, then continue at Step 3. Otherwise, decompose G' into biconnected components using the biconnectivity algorithm of Tarjan[20]. Repeat Step 3 and 4 for each of the biconnected components.

3. Perform a depth-first search over G' (or a biconnected component of G') to create an adjacency structure of the graph that complies with certain conditions (to be discussed in the following sections).

4. Based on the adjacency structure, perform a depth-first search over G' (or a biconnected component of G') to decompose the graph into a collection of paths.

5. For each of the paths determined in Step 4, determine a subset of separation pairs lying on it. The separation pairs are identified by keeping track of two stacks one of which contains the potential separation pairs and the other contains the edges of the split components. The split components are determined along with the separation pairs.

6. The set of triple bonds and triangles are merged to form a set of bonds and polygons. These bonds, polygons together with the split components form the triconnected components of the input graph G.

3.2 A Characterization of Separation Pairs

Hopcroft et al. classify separation pairs into two types: Type-1 and Type-2. Furthermore, they give the following characterization theorem of separation pairs (note that the vertices are presented by their depth-first search numbers):

Theorem 13[20] Let $G = (V,E)$ be a biconnected graph and a,b be two vertices in G with $a < b$. Then $\{a,b\}$ is a separation pair if and only if one of the following conditions holds:

• Type-1: There are distinct vertices $r \neq a,b$ and $s \neq a,b$ such that $b \rightarrow r$, $lowpt1(r) = a$, $lowpt2(r) \geq b$ and s is not a decendant of r.

• Type-2: There is a vertex $r \neq b$ such that $a \rightarrow r \rightsquigarrow b$, b is a first decendant of r,$a \neq 1$, every frond $x \dashrightarrow y$ with $r \leq x < b$ has $a \leq y$, and every frond $x \dashrightarrow y$ with $a < y < b$ and $b \rightarrow w \dashrightarrow x$ has $lowpt1(w) \geq a$.

We shall use the graph given in Hopcroft et al. to illustrate various concepts and ideas. The graph is shown in figure 3.1.

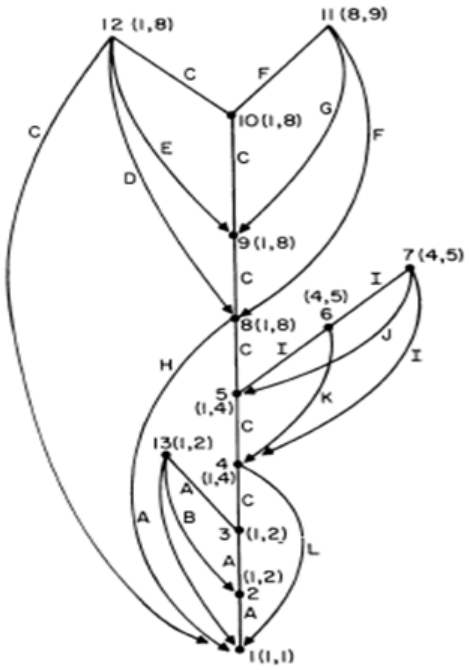

Figure 3.1: Separation pairs[20]

3.3 A High-level Description of the Algorithm

In the algorithm of Hopcroft et al., in order to find the triconnected components, the graph is split into smaller graphs which are called the split graphs. For each separation pair (a,b), the graph G is split into two graphs G_1 and G_2 and an virtual edge of (a,b) is added to the split graphs. If the split graphs are split until no more splits are possible, the graphs remaining are called the split components. These split components are non-unique.

There are three types of split components: triple bonds such as $(\{a,b\},\{(a,b),(a,b),(a,b)\})$, triangles such as $(\{a,b,c\},\{(a,b),(a,c),(b,c)\})$ and a triconnected graphs.

Some of the split components are then reassembled (merged). Specifically, the triple bonds are merged to form a set of bonds B and the triangles are merged to form polygons P. The set of $B \bigcup P \bigcup \{triconnecgtedgraphs\}$ is the set of triconnected components of the input graph G. Note that the triconneted components of a graph are unique.

Basically, the algorithm of Hopcroft et al. can be summarized into the following steps:

- The multiple edges in the graph G are split off to form a set of triple bonds and a graph G'
- The biconnected components of G' are determined and for each of them are processed as follows:
  1. The split components are determined;
  2. Connected components are determined by combining B, P and the triconnecgted graphs.

The main part of the algorithm of Hopcroft et al. is to find the split components of the biconnected components of G'. For this purpose, he had to determine the separation pairs of the graph. In order to find the separation pairs, several passes are made over the graph and a few values are calculated as follows:

- The graph is traversed with DFS and $lowpt1(v)$, $lowpt2(v)$, $nd(v)$ and $parent(v)$ are calculated;
- An acceptable adjacency list is constructed for each vertex v;
- A DFS is performed based on the acceptable adjacency list structure to partition the edges into disjoint simple paths, $lowpt1(v)$ and $lowpt2(v)$ are recalculated and $A1[v]$, $degree(v)$ and $Highpt$ are calculated for each node.

During the first depth-first search, the following parameters are calculated for each vertex v:] $nd(v)$ (the number of descendants of v), $lowpt1(v)$ and $lowpt2(v)$.
The definition for $lowpt1(v)$ and $lowpt2(v)$ are as follows:
$lowpt1(v) = \min(\{v\} \bigcup \{lowpt1(w) | v \rightarrow w\} \bigcup \{w | v - \rightarrow w\})$

and

$lowpt2(v) = min(\{v\} \bigcup ((\{lowpt1(w) | v \rightarrow w\} \bigcup \{lowpt2(w) | v \rightarrow w\} \bigcup \{w | v - \rightarrow w\}) - \{lowpt1(v)\}))$

In simple words, $lowpt1(v)$ is the lowest vertex accessible to v by traversing zero or more tree-edges and one frond and $lowpt2(v)$ would be the second lowest vertex accessible to v in the graph G in the same way.
Next, the order of the vertices adjacent to v in $A[v]$ is changed. The resulting adjacency list, $A(v)$ is called an *acceptable* adjacency lists. $A(v)$ is created based on the function ϕ. It is defined as below:

if $e = (v,w)$ is a tree edge and $lowpt2(w) < v$, then $\phi(e) = 2lowpt1(w)$
if $e = (v,w)$ is a tree edge and $lowpt2(w) \geq v$, then $\phi(e) = 2lowpt1(w) + 1$
if $e = (v,w)$ is an outgoing frond, then $\phi(e) = 2w + 1$.

Next, another DFS is performed over the graph, this search will proceed according to the new $A(v)$ to generate a collection of paths. Specifically, whenever a tree-edge is encountered, it is added to the path and whenever a frond is encountered, it will be the last edge in the path. The first path is a cycle and the remaining paths are simple paths. Each path has its first and last vertex in common with other paths. Becaue these paths are generated according to the *acceptable* adjacency list every vertex will end in the lowest possible vertex. $A1(v)$ is defined as the first vertex encountered in the new $A(v)$ and *Highpt* is the first frond encountered during the path finding search. Since the paths are found based on DFS they can be found in $O(|V| + |E|)$ time.
Next, the separation pairs are determined based on the theorem stated at the beginning of this chapter (which we restate below for convenience) and the values calculated above.

Theorem 13[20] Let $G = (V,E)$ be a biconnected graph and a,b be two vertices in G with $a < b$. Then $\{a,b\}$ is a separation pair if and only if one of the following conditions holds:
• Type-1: There are distinct vertices $r \neq a,b$ and $s \neq a,b$ such that $b \rightarrow r$, $lowpt1(r) = a$, $lowpt2(r) \geq b$ and s is not a decendant of r.
• Type-2: There is a vertex $r \neq b$ such that $a \rightarrow r \rightsquigarrow b$, b is a first decendant of r,$a \neq 1$, every frond $x \dashrightarrow y$ with $r \leq x < b$ has $a \leq y$, and every frond $x \dashrightarrow y$ with $a < y < b$ and $b \rightarrow w \dashrightarrow x$ has $lowpt1(w) \geq a$.
Two stacks are used, namely *ESTACK* and *TSTACK*.
The *TSTACK* contains triples (h,a,b) where a,b is a potential separation pair and h is the largest

vertex in the triconnected component. The algorithm checks the top entry on the *TSTACK* to see if the condition given in Theorem 13 is satisfied. The **ESTACK** is used to keep the edges of split components. The output of the algorithm is the separation pairs and the triconnected components of the graph.

Chapter 4

The Distributed Model

A distributed network can be represented by an undirected graph $G = (V; E)$ where V is the set of network sites and E is the set of communication links. The sites communicate with each other by sending messages over the communication links and performing some computations locally. Every site has an identity and is able to keep data locally. Each site knows which other sites in the network it is connected to. Without loss of generality, we assume that a message if sent will be delivered and no message will be lost in the network. We also assume that it will take finite time for every message to be delivered. Since communication cost is larger than local computation time by many order of magnitude, we shall consider the time for local computations as negligible.

The message complexity of a distributed algorithm is the total number of messages sent when the algorithm is executed. The time complexity of a distributed algorithm is the maximum time passed between the beginning and end of the algorithm. For simplicity, it is commonly assumed that each message will take at most one time unit to be delivered.

Since our distributed algorithm is bsed on an algorithm that uses depth-first search, we will need to study distributed depth-fist search. There have been a number of DDFS algorithms presented during the last two decades. These algorithms have different message lengths and different rules imposed on their networks. Some of the most important DDFS algorithms are presented by Awerbuch[4], Cidon[12],Cheung[9], Lakshmanan et al.[29], Makki et al.[31] and Sharma et al.[39]. The following table compares the time and message complexity of these algorithms:

Table 4.1: Different DDFS algorithms and their complexities[47]

Author	Message length	Time	Mesage
Cheung	$O(1)$	$2m$	$2m$
Awerbuch	$O(1)$	$< 4n$	$4m$
Lakhamanan et al.	$O(1)$	$2n-2$	$\leq 2m, < 4m-(n-1)$
Cidon(corrected)	$O(1)$	$2n-2$	$\leq 2m, < 4m-(n-1)$
Tsin	$O(\lg n)$	$n(1+r)$	$\leq n+1, < 4m-(n-1)$
Sharma et al.	$O(n)$	$2n-2$	$2n-2$
Makki et al.	$O(n)$	$n(1+r)$	$n(1+r)$

Some of the algorithms assume that every node in the network has access to global information about all the other nodes in the network. This assumption will lower the number of messages passed along the network substantially. The algorithms presented by Sharma et al. and Makki et al. use such an assumption and use messages of length $O(n)$. They show improvement in time or message complexity compared to the other algorithms.

Other algorithms such as Cidon's use messages of length $O(1)$. Cidon's algorithm also adopt a weaker computer model in that it does not require the FIFO (First-in-first-out) rule, which means that the messages are not processed in the order they arrived and are not delivered in the same order they were received. In 2002, Tsin[47] presented two counterexamples to show that Cidon's algorithm does not always work correctly. He also showed how to correct the errors in Cidon's algorithm. We will base our distributed depth-fist search on the corrected Cidon's algorithm according to Tsin.

In summary, the following are the assumptions we make for our network model:

- All the messages sent over the links are delivered correctly in one time unit; each message deliverance takes finite time and the FIFO rule is not required.
- Every node knows only its incident links and has no global knowledge of the network.
- the message length is $O(\log|V|)$.
- the nodes in the network run asynchronously (i.e. there is no clock synchronizing them).

Each node can be either in the *idle* state or *discovered* state. The edges incident to every node can be marked in one of the four possible ways: *unvisited*, *visited*, *parent* and *child*. Initially, all the links

are marked *unvisited*. There are two types of messages, namely **Visited** and **Token**, for performing the *basic* depth-first search. The **Token** is sent in sequential order from a node to one of its adjacent nodes as the graph is being traversed by a depth-first search.

Initially, all the nodes are in *idle* state. The distributed algorithm is invoked by one of the nodes. After entering the *discovered* state, the node chooses one of its *unvisited* links and sends the **Token** over the link. It then marks the link as *child*. The node also sends a **Visited** message over every incident link other than the *parent* link to let the node at the end knows that it has been visited by the search.

If a node receives a **Token** message through an *unvisited* link, it will mark the link as *parent* and enter the *discovered* state, then it will pass the **Token** along one of its *unvisited* links. If there are no *unvisited* links left, then the search has to backtrack so the **Token** is sent over the *parent* link. This process will continue until the search backtracks to the first node that initiated the DFS (which has become the root of the DFS tree).

If the Token is sent through a *visited* or *unvisited* link but not for the first time, the link is marked as *visited* and no other action is taken. If a **Visited** message is received through an *unvisited* or *child* link, the link will be marked as *visited*.

Chapter 5

The Distributed Algoithm

5.1 The Distributed Algorithm

We shall assume without loss of generality that the network is biconnected. If it is not, we can use the existing distributed algorithm for biconnectivty to decompose the network into a collection of biconnected networks and then apply our algorithm to each of them.

The algorithm is based on the sequential algorithm of Hopcroft and Tarjan. To carry the depth-first search, we shall use the distributed depth-first search algorithm of Cidon with corrections made by Tsin.

As with the algorithm of Hopcroft et. al., our algorithm makes three passes over the network $G = (V, E)$. During the first pass, a depth-first search is performed over G to generated a Palm tree $P = (V, E_P)$ of G and $lowpt1(v), lowpt2(v), nd(v), v \in V$, are computed. Using $lowpt1(v)$, $lowpt2(v), v \in V$, the values $\phi(e), \forall e \in E$, is computed and the incident links of the Palm tree at each node v are rearranged to produce the acceptable adjacency structure, $A(v), v \in V$.

During this pass, the following types of messages are used:

Token($flag, count, lowpt1, lowpt2, nd$): when $flag = 1$, the message is sent to a child node; when $flag = 0$, the message is sent to the parent node; *count* is the next depth-first search number available for assigning to a node; $lowpt1$ and $lowpt2$ are the $lowpt1$ and $lowpt2$ values of the sender; nd is

the number of descendants of the sender.

Visited(dfs): dfs is the depth-first search number of the sender.

During the second pass, a depth-first search is made over the Palm tree P using the acceptable adjacency structure and $lowpt1(v)$, $lowpt2(v), v \in V$ are recomputed. The network G is also implicitly decomposed into a collection of paths, $P_i, 1 \leq i \leq k$, such that P_1 is a cycle while each $P_i, 2 \leq i \leq k$ is a simple path whose terminating nodes lie on some $P_j, j < i$.

During this pass, the following types of messages are used:

Token1(m): m is the parameter used in determining the new depth-first number during the second pass (see Procedure 5 of[20]).

Visited: same as above.

During the third pass, a subset of separation pairs that are sufficient to decompose G into the split components are determined. The separation pairs are determined in parallel on the paths $P_i, 1 \leq i \leq k$.

During this pass, the following types of messages are used:

Vlink: to inform an ancestor of the sender to create a virtual link.

DecreaseDegree: to inform a descendant of the sender (which forms a Type-2 pair with the sender) to update its degrees as well as creating a virtual link.

UpdateDegree: to inform an ancestor of the sender (which forms a Type-1 pair with the sender) to update its degrees as well as creating a virtual link.

Deg_of_child_is_2($flag, firstchild$) with $TSTACK$: $flag = true$ if and only if the sender, say v, satisfies the condition: "$deg(v) = 2 \wedge firstchild(v) > v$". When $flag = true$, $fisrtchild$ is the first child of the sender, it is irrelevant otherwise.

Deg_of_child_is_2($flag, firstchild, d$): $flag = true$ if and only if the sender, say v, satisfies the con-

dition: "$deg(v) = 2 \wedge firstchild(v) > v$". When $flag = true$, $fisrtchild$ is the first child of the sender, it is irrelevant otherwise. The receiver of the message is d which is an ancestor of the sender.

5.1.1 Determining *lowpt*1, *lowpt*2 and *nd*(v)

Initially, all nodes are in the *idle* state and all links are marked *unvisited*. One predesignated node r then enters the *discovered* state and initiates the algorithm by starting a depth-first search as follows:

$dfs(r) \leftarrow 1$; $lowpt1(r) \leftarrow lowpt2(r) \leftarrow 1$; $nd(r) \leftarrow 1$;

call *DFS*(2);

send a message **visit**(1) over every *unvisited* incident link.

Procedure DFS(*count*)

if ($\exists$ an incident link e that is marked *unvisited*) **then**

send **Token**$(1, count, -, -, -)$ over link e;

mark link e as *child*;

else if ($v = r$) **then** /* v is the node executing algorithm DFS */

STOP

else send **Token**$(0, count, lowpt1(v), lowpt2(v), nd(v))$ over the *parent* link;

At each node $v \in V$, the following operations for computing the values of $lowpt1(v)$,$lowpt2(v)$, and $nd(v)$ are carried out:

• On receiving a **Token**$(1, count, -, -, -)$ message from an incident link e:

if ($state(v) = idle$) **then**

$state(v) \leftarrow discovered$;

mark link e as the *parent* link;

$dfs(v) \leftarrow count$;

if (there is no saved **visited** message) **then**

$lowpt1(v) \leftarrow lowpt2(v) \leftarrow dfs(v)$;

else

$lowpt1(v) \leftarrow \min(\{dfs(w) | (\textbf{visit}(dfs(w)) \text{ is a saved message}) \wedge (v,w) \neq e\}$
$\cup \{dfs(v)\})$;

$lowpt2(v) \leftarrow \min(\{dfs(w) | (\textbf{visit}(dfs(w)) \text{ is a saved message}) \wedge (v,w) \neq e\}$
$- \{lowpt1(v)\} \cup \{dfs(v)\})$;

$nd(v) \leftarrow 1$;

call DFS($count + 1$);

send a **visit**($dfs(v)$) message over every *visited* or *unvisited* incident link.

else if (link e is marked *unvisited*) **then**

mark link e as *visited*

else if (link e is marked *child*) **then**

mark link e as *visited*;

/* a **Token** message must have been sent over link e earlier.

Let it be **Token**$(1, cnt, -, -, -)$. Then, re-transmit the message */

call DFS(cnt);

• Upon receiving a **visited**(w) message from an incident link e:

if ($state(v) = idle$) **then** save the message

else if (e is marked '*unvisited*') /* i.e. e is a back-edge */

then

mark link e as *visited*;

if ($w < lowpt1(v)$) **then**

$lowpt2(v) \leftarrow lowpt1(v)$;

$lowpt1(v) \leftarrow w$;

else if ($lowpt1(v) < w < lowpt2(v)$) **then**

$lowpt2(v) \leftarrow w$

else if (link e is marked *child*) **then**

mark link e as *visited*;

/* a **Token** message must have been sent over link e earlier.

Let it be **Token**$(1, cnt, -, -, -)$. Then, re-transmit the message */

call DFS(cnt);

if ($w < lowpt1(v)$) **then**

$lowpt2(v) \leftarrow lowpt1(v)$;

$lowpt1(v) \leftarrow w$;

else if ($lowpt1(v) < w < lowpt2(v)$) **then**

$lowpt2(v) \leftarrow w$;

• On receiving a **Token**$(0, cnt, L1, L2, nd)$ message from a link e:

/* Link e must be a *child* link */

if ($L1 < lowpt1(v)$) **then** /* Update $lowpt1(v), lowpt2(v)$ */

$lowpt2(v) \leftarrow \min\{lowpt1(v), L2\}$;

$lowpt1(v) \leftarrow L1$;

else if ($L1 = lowpt1(v)$) **then**

$lowpt2(v) \leftarrow \min\{lowpt2(v), L2\}$;

else $lowpt2(v) \leftarrow \min\{lowpt2(v), L1\}$;

$nd(v) \leftarrow nd(v) + nd$;

call DFS(cnt)

Theorem 5.1 When the depth-first search terminates at the root r, lowpt1(v);lowpt2(v);nd(v);$\forall v \in V$, are correctly computed.

proof: (By induction on the level of the nodes.)

Base case: Consider a leaf-node v. Since v has no child nodes, all of the links incident on it are either outgoing fronds or the parent link. Therefore, $lowpt1(v)$ and $lowpt2(v)$ are computed purely based on the **Visited** messages. As a result, $lowpt1(v)$ and $lowpt2(v)$ are computed as:

$lowpt1(v)$

$= \min(\{dfs(v)\} \cup \{dfs(w) | (v \hookrightarrow w) \in E_P\})$

$= \min(\{dfs(v)\} \cup \{lowpt1(w) | (v \rightarrow w) \in E_P\} \cup \{dfs(w) | (v \hookrightarrow w) \in E_P\})$ and

$lowpt2(v)$

$= \min(\{dfs(v)\} \cup (\{dfs(w) | (v \hookrightarrow w) \in E_P\} - \{lowpt1(v)\}))$

$= \min(\{dfs(v)\} \cup (((\{dfs(w) | (v \hookrightarrow w) \in E_P\} \cup \{lowpt1(w) | (v \rightarrow w) \in E_P\}$

$\cup \{lowpt2(w) | (v \rightarrow w) \in E_P\}) - \{lowpt1(v)\}))$.

Hence, both $lowpt1(v)$ and $lowpt2(v)$ are computed correctly at every leaf-node. Furthermore, $nd(v)$ is correctly set to 1.

Induction hypothesis: Suppose for every node w on $level > k$, $lowpt1(w)$,$lowpt2(w)$, $nd(w)$ are correctly computed.

Induction step: Consider a node v on level k. For each frond $v \hookrightarrow w$, node v receives $dfs(w)$ through the incident link $e = (v \hookrightarrow w)$. Furthermore, by the induction hypothesis, $lowpt1(w)$,$lowpt2(w)$,$nd(w)$ are correctly computed for every child node w. Since $lowpt1(w)$,$lowpt2(w)$ are sent to node v through a **Token** messages, it is easily verified that the algorithm correctly computes $lowpt1(v)$ as:

$\min(\{dfs(v)\} \cup (\{lowpt1(w) | v \rightarrow w\} \cup \{dfs(w) | (v \hookrightarrow w) = e \in E_P\}))$

and $lowpt2(v)$ as:

$\min(\{dfs(v)\} \cup ((\{lowpt1(w) | v \rightarrow w\} \cup \{lowpt2(w) | v \rightarrow w\} \cup \{dfs(w) | (v \hookrightarrow w) = e \in E_P\}) - \{lowpt1(v)\}))$.

Furthermore, it is easily verified that $nd(v)$ is correctly computed as $\sum\{nd(w) | v \rightarrow w\} + 1$. ■

During the third pass, the algorithm finds a subset of separation pairs that are sufficient to generate

all the split components of the network. The separation pairs are determined based on Lemma 3 of Gutwenger et. al., or originally, Lemma 13 of Hopcroft et. al.

Theorem 5.2 *The distributed algorithm takes $O(|V|)$ time and transmits $O(|E|)$ messages to compute $lowpt1(v), lowpt2(v), nd(v), \forall v \in V$.*

Proof: The algorithm performs a depth-first search over the network G. Since the computation of $lowpt1(v), lowpt2(v), nd(v), v \in V$, are done locally at each node v, the computation thus takes *zero* time. The total time and message complexities are thus that same as those of the distributed depth-first search algorithm which are $O(|V|)$ and $O(|E|)$, respectively[47]. ■

5.1.2 Constructing the acceptable adjacency structure

At the end of the depth-first search during the first pass, the adjacency list for the palm tree at every node is rearranged and the nodes are renumbered with new depth-first search numbers so that the following properties are satisfied:

(i) the root of the depth-first tree T is numbered 1;

(ii) let $v \in V$ and $w_i, 1 \leq i \leq n$, be its children in T. Then

$dfs(w_i) = dfs(v) + \sum_{j=i+1}^{n} nd(w_j) + 1;$

(iii) For each $v \in V$, the edges e in the adjacency list of v, $A(v)$, are in ascending order according to $lowpt1(w)$ if $e = v \rightarrow w$, or w if $e = v \hookrightarrow w$, respectively. Moreover, let $w_i, 1 \leq i \leq n$, be the children of v. Then $\exists i_0$ such that $lowpt2(w) < dfs(v), 1 \leq i \leq i_0$ and $lowpt2(w) \geq dfs(v), i_0 < i \leq n$. Every frond in $A(v)$ comes in between $v \rightarrow w_{i_0}$ and $v \rightarrow w_{i_0+1}$.

At each node v, the links of the palm tree incident on v are ordered according to the following value:

$\forall e \in E, \phi(e) =$

if $e = (v, w)$ is a tree edge and $lowpt2(w) < v$, then $\phi(e) = 3lowpt1(w)$

if $e = (v, w)$ is a tree edge and $lowpt2(w) \geq v$, then $\phi(e) = 3lowpt1(w) + 1$

if $e = (v, w)$ is an outgoing frond, then $\phi(e) = 3w + 2$.

Definition: The adjacency structure for the palm tree, $P = (V, E_P)$, of the network $G = (V, E)$ is ***acceptable*** if for every $v \in V$, the edges e in the adjacency list of v are ordered according to increasing value of $\phi(e)$.

Note that the acceptable adjacency structure involves the *child* links and the *outgoing non-tree* links (links $e = v \hookrightarrow w$ such that $dfs(v) > dfs(w)$) of G only.

At every node $v, v \in V$, the following operations are performed:

call Compute_phi(v);

Create the acceptable adjacency structure, $A(v), v \in V$, using $\phi(e), e \in E$.

Procedure Compute_phi(v);

---

for each ($e \in A(v)$) **do**

if ($e = v \rightarrow w$) **then** /* e is a tree link */

if ($lowpt2(w) < v$) **then**

$\phi(e) \leftarrow 3 * lowpt1(w)$

else $\phi(e) \leftarrow 3 * lowpt1(w) + 2$

else /* e is a frond */

$\phi(e) \leftarrow 3 * lowpt1(w) + 1$

The correctness is obvious. The time and message complexities are obviously *zero*.

5.1.3 Recalculating *dfs*(*v*), *lowpt1*(*v*), *lowpt2*(*v*)

During the second pass, a depth-first search is performed over the palm tree, $P = (V, E_P)$ of the network $G = (V, E)$ using the acceptable adjacency structure $A(v), v \in V$, of P and renumber the nodes with depth-first search numbers.

When the first pass was completed, all the nodes re-entered the *idle* state. The root of the depth-first search tree r initiates the second pass as follows:

$state(r) \leftarrow discovered$;
$m \leftarrow |V|$;
$next \leftarrow 0$;
$highpt(r) \leftarrow 0$;
$newdsf(r) \leftarrow 1$ /* which is $m - nd(r) + 1$ */;
call Pathsearch($next, m, r$);
send a **visited**($newdfs(r)$) message over every *outgoing* non-tree link.

Procedure Pathsearch($next, m, v$)
$next \leftarrow next + 1$;
while (($A(v)[next]$ is a non-tree link) $\wedge$ ($next \leq deg(v)$)) **do** /* skip non-tree links */
$next \leftarrow next + 1$;
if ($next \leq deg(v)$) **then**
send **Token1**(m) over the child link e, where $e = A(v)[next]$;
else $dfs(v) \leftarrow newdfs(v)$; /* update $dfs(v)$ */
if ($v = r$) **then** /* v is the node executing algorithm DFS */
STOP
else $m \leftarrow m - 1$;
send **Token1**(m) over the *parent* link;

At each node $v \in V$, the following operations for computing the values of $lowpt1(v), lowpt2(v)$, and $nd(v)$ are carried out:

• On receiving a **Token1**(m) message from the parent link:

$state(v) \leftarrow discovered$;

$next \leftarrow 0$;

$highpt(r) \leftarrow 0$;

$newdsf(v) \leftarrow m - nd(v) + 1$;

call Pathsearch($next, m, v$);

send a **visit**($newdfs(v)$) message over every *outgoing non-tree* link.

• On receiving a **visited**(w) message from an incident link e:

/* the link e must be an incoming non-tree link */

$highpt(v) \leftarrow \max\{highpt(v), w\}$;

• On receiving a **Token1**(m) message from a *child* link e:

call Pathsearch($next, m, v$)

Theorem 5.3 *When the depth-first search terminates at the root r, $dfs(v), highpt(v), \forall v \in V$, are correctly computed.*

Proof: The distributed algorithm performs a depth-first search over the network G to re-calculate $dfs(v), v \in V$. The correctness of computing $dfs(v), v \in V$, thus follows from that given in[47].
The correctness of computing $highpt(v), v \in V$ follows from that given in[20]. ■

Theorem 5.4 *The distributed algorithm takes $O(|V|)$ time and transmits $O(|E|)$ messages to compute*

$dfs(v), highpt(v), \forall v \in V$.

Proof: Same as that of Theorem 5.2. ■

After re-computing $dfs(v) \forall v \in V$, another depth-first search is performed over the network G to re-calculate $lowpt1(v), lowpt2(v), \forall v \in V$. Since the detail of the algorithm, the correctness proof and the time and message complexity analysis are essentially the same as those for the first pass, we shall omit them here. Instead, we state the following theorem.

Theorem 5.5 *The distributed algorithm takes $O(|V|)$ time and transmits $O(|E|)$ messages to re-compute $lowpt1(v), lowpt2(v), \forall v \in V$.*

Proof: Immediate. ■

5.1.4 Determining the Separation Pairs

For ease of explanation, we shall use v and $dfs(v)$ interchangeably in this section. During the third pass, the algorithm finds a subset of separation pairs that are sufficient to generate all the split components of the network. The separation pairs are determined based on Lemma 3 of Gutwenger et. al., or originally, Lemma 13 of Hopcroft et. al.

The third pass is initiated by all those nodes that have an outgoing non-tree link which is the last link of a path. It goes as follows:

if ($A(v)$[1] is an outgoing non-tree link) **then**

/* The outgoing non-tree link is the last link of a path. */

$TSTACK \leftarrow null$;

$v.link \leftarrow v.parent \leftarrow null$; /* initialize the virtual links */

$A(v) \leftarrow A(v) - A(v)$[1]; /* exclude the last link from further consideration */

call Update-TSTACK(v);

send *TSTACK* and a **deg_of_child_is_2**(*false*,*null*) message over the *parent* link;

Procedure Update-TSTACK(v)

---

for each ($e \in A(v)$) **do**

if ($e = v \rightarrow w$) **then** /* e is a tree link */

while (top entry (h,a,b) on *TSTACK* satisfies $a > lowpt1(w)$) **do**

pop *TSTACK*;

if (no entry was popped out of *TSTACK*) **then**

push $(w + nd(w) - 1, lowpt1(w), v)$ onto *TSTACK*

else

let (h',a',b') be the last entry popped;

push $(\max\{y, w + nd(w) - 1\}, lowpt1(w), b')$ onto *TSTACK*;

else let $e = v \hookrightarrow w$; /* e must be a frond */

while (top entry (h,a,b) on *TSTACK* satisfies $a > dfs(w)$) **do**

pop *TSTACK*;

if (no entry was popped out of *TSTACK*) **then**

push (v,w,v) onto *TSTACK*

else

$y \leftarrow \max\{h | (h,a,b) \textit{was popped out of} \textit{TSTACK}\}$;

let (h',a',b') be the last entry popped;

push (y,w,b') onto *TSTACK*

rof;

Procedure Type-2($flag, z$)

---

if ($flag = true$) **then** /* i.e. ($deg(w) = 2 \wedge firstchild(w) > w$) */

mark (v, z) as a type-2 separation pair; /* $z = firstchild(w)$ */

$v.child(v) \leftarrow z$; /*effectively deleted (v, w) and $(w, firstchild(w))$ */

send a **vlink**(v, z) message over the link (v, w);

else if (*top entry*(h, a, b) *on TSTACK satisfies* $a = v$) **then**

pop top-entry (h, a, b) out of *TSTACK*;

mark (a, b) as a type-2 separation pair;

for each $(e = (v, x) \in E_v)$ **do**

if $(a \leq x \leq h)$ **then**

$deg(v) \leftarrow deg(v) - 1$; $E_v \leftarrow E_v - \{e\}$;

/* Join nodes $v(= a)$ and b with a virtual link (v, b) */

$v.child(v) \leftarrow b$; $deg(v) \leftarrow deg(v) + 1$; $E_v \leftarrow E_v \cup \{(v, b)\}$;

send a **DecreaseDegree**(b, a, h) message to node b

Procedure Type-1

---

mark $(v, lowpt1(w))$ as a type-1 pair;

for each $(e = (v, x) \in E_v)$ **do**

if $(w \leq x \leq w + nd(w) - 1)$ **then**

$deg(v) \leftarrow deg(v) - 1$; $E_v \leftarrow E_v - \{e\}$;

$v.parent \leftarrow lowpt1(w)$; $E_v \leftarrow E_v \cup \{(v, lowpt1(v))\}$; $deg(v) \leftarrow deg(v) + 1$;

send an **UpdateDegree**$(lowpt1(w), w, nd(w))$ message to node $lowpt1(w)$ over

the *parent* link;

Procedure Separation-pairs

---

if $((a = v) \wedge (parent(b) = a))$ **then** pop top entry out of *TSTACK*;

if $(v \neq 1) \wedge ((flag = true) \vee$ top entry (h,a,b) on *TSTACK* satisfies $(a = v))$ **then**

call Type-2$(flag, z)$

else if $(lowpt1(w) < v \wedge lowpt2(w) \geq v)$

$\wedge(parent(v) \neq 1 \vee$ *v has a child other than w*$)$ **then**

call Type-1;

call Clean-up;

else call Clean-up;

Procedure Clean-up

---

while (top entry (h,a,b) on TSTACK satisfies $v \notin \{a,b\} \wedge (highpt(v) > h)$)

do pop *TSTACK*;

if (*e* does not start a new path) **then**

$A(v) \leftarrow A(v) - \{e\}$;

call Update-TSTACK(v);

if $(v \neq 1 \wedge (deg(v) = 2 \wedge firstchild(v) > v))$ **then**

send a **deg_of_child_is_2**(*true*, $firstchild(v)$) message with *TSTACK* over the *parent* link;

else

send a **deg_of_child_is_2**(*false*, *null*) message with *TSTACK* over the *parent* link;

For every node $v \in V$:

• On receiving a **deg_of_child_is_2**(*flag*, *z*) message and a *TSTACK* from a child link *e*:

call Separation-pairs

• On receiving a **deg_of_child_is_2**(*flag*, *z*, *d*) message from a *child* link:

if ($v \neq d$) **then**

pass the message over the *parent* link

else /* there must exist a *TSTACK* at node *v* */

call Separation-pairs

• On receiving a **DecreaseDegree**(b,a,h) message from the *parent* link:

if $(v = b)$ **then**

for each $(e = (v,x) \in E_v)$ **do**

if $(a \leq x \leq h)$ **then**

$deg(v) \leftarrow deg(v) - 1;\ E_v \leftarrow E_v - \{e\};$

/* Join nodes a and v with a virtual link (a,v) */

$v.parent(v) \leftarrow a;\ deg(v) \leftarrow deg(v) + 1;\ E_v \leftarrow E_v \cup \{(a,v)\};$

if $(deg(v) = 2 \wedge firstchild(v) > v)$ **then**

send a d**eg_of_child_is_2**$(true, firstchild(v), a)$ message over the *parent* link;

else send a **deg_of_child_is_2**(*false,null,a*) message over the *parent* link;

else pass the message over the *firstchild* link;

• On receiving a **vlink**(z,z') message from the *parent* link:

if $(v = z')$ **then**

$v.parent \leftarrow z; E_v \leftarrow E_v \cup \{(v,z)\} - \{parent\ link\};$

send a **deg_of_child_is_2**$(false, null, z)$ message over the *parent* link

else send the message over the *firstchild* link;

• On receiving a **UpdateDegree**(z,w,nd) message from a child link e:

if $(v = z)$ **then**

for each $(e = (v,x) \in E_v)$ **do**

if $(w \leq x \leq w + nd - 1)$ **then**

$deg(v) \leftarrow deg(v) - 1; E_v \leftarrow E_v - \{e\};$

$v.child(v) \leftarrow w; E_v \leftarrow E_v \cup \{(v,w)\};\ deg(v) \leftarrow deg(v) + 1;$

else if (e does not start a new path) **then**

send the message over the *parent* link;

Since the distributed algorithm finds a subset of separation pairs, we must show that the subset is sufficient to decompose the network G into its splits components.

Lemma 5.6 *Let* (a,b) *be a Type-1 separation pair. The distributed algorithm correctly identifies* (a,b) *as a separation pair.*

Proof: By Lemma 3 of Gutwenger et. al., (or Lemma 13 of Hopcroft et. al.), $\exists r,s \in V$ such that $r,s \notin \{a,b\}, b \rightarrow r$, $lowpt1(r) = a$, $lowpt2(r) \geq b$ and s is not a descendant of r. It follows that if $parent(b) = 1$, then $a = 1$ and s is not a descendant of r implies that s must be a descendant of a child x of a or a child y of b such that $x \neq b$ and $y \neq r$. But G is biconnected implies that the former case is impossible. The latter case implies that b has a child other than r.

Therefore, the condition $(lowpt(r) < b \wedge lowpt2(r) \geq b) \wedge (parent(b) \neq 1 \vee b \text{ has a child other than } r)$ holds true. As a result, after node b invokes Procedure Separation-pairs, the aforementioned condition will eventually be evaluated which results in (a,b) being marked as a Type-1 separation pair at nodes b. ■

Lemma 5.7 *Let* w *be a node of degree 2 in* G *such that* v *and* u *are the two nodes adjacent to it. The distributed algorithm correctly identifies* (v,u) *as a separation pair.*

Proof: The links (v,w) and (w,u) must lie on a common path P_i. Furthermore, one of them, say (v,w), must be a tree link while the other, say (w,u), is either a tree link or a frond.

If (w,u) is a frond, then (v,u) is a Type-1 separation pair and by Lemma 5.1, the algorithm correctly identifies it.

Suppose (w,u) is a tree link. When node w sends its *TSTACK* over its parent link to v, it also send a **deg_of_child_is_2**$(true,u)$ message over the parent link. Upon receiving the *TSTACK* and the **deg_of_child_is_2**$(true,u)$ message, node v will invoke Procedure Type-2 which results in marking

(v,u) as a Type-2 separation pair. ■

Lemma 5.8 *Let G be a biconnected network with no vertices of degree two or type-1 separation pairs. Suppose on returning from node w to node v along the tree link $v \rightarrow w$ on a path P_i, the top entry on TSTACK (h,a,b) is found to satisfy the Type-2 test, Then (a,b) is a Type-2 separation pair.*

Proof: The Type-2 test tests for the the condition:

$(v \neq 1) \wedge$ (top entry (h,a,b) on $TSTACK$ satisfies $(a = v)) \wedge (parent(b) \neq a)$

Since the condition is satisfied, we have $(parent(b) \neq a)$ which implies that $\exists r$ such that $a \rightarrow r$ and r is an ancestor of b.

If there a frond (x,y) such that $r \leq x < b$ and $y < a$, then the triple on $TSTACK$ corresponding to (h,a,b) would have been deleted when $TSTACK$ was examined at node u where u is the lowest common ancestor of x and b. But then (h,a,b)

Similarly, if there a frond (x,y) such that x is a descendant of b while y lies on the tree-path connecting nodes a and b, then the triple on $TSTACK$ corresponding to (h,a,b) would have been deleted when the frond (x,y) was examined at node y owing to $highpt(y)$. But then (h,a,b) cannot be the top entry on $TSTACK$ at node v, a contradiction!

It then follows from Lemma 3 of Gutwenger et. al., (or Lemma 13 of Hopcroft et. al.) that (a,b) is a Type-2 separation pair. ■

Lemma 5.9 *Let G be a biconnected network with no vertices of degree two or type-1 separation pairs. Suppose G has a type-2 separation pair. Then the distributed algorithm will detect a separation pair.*

Proof: Let (a,b) be a Type-2 separation pair and $b_1, b_2, \ldots, b_n$ be the children of b in the order they occur in A(b).

Let $i_0 = \min\{i \mid lowpt1(b_i) \geq a\}$. If i_0 exists, the the triple $(b_{i_0} + nd(b_{i_0}) - 1, lowpt1(b_{i_0}), b)$ will be pushed onto $TSTACK$ at node b when the tree link $b \rightarrow b_{i_0}$ is being examined. The triple may be replaced, but only with triple of the form (h,x,b) such that $a \leq x \leq lowpt1(b_{i_0})$. Eventually, such a triple will satisfy the Type-2 test unless another Type-2 pair is found first.

If i_0 does not exist, let (x, y) be the first link examined after node b is reached such that $a \leq y$ and $x \leq b$. If (x, y) is a frond, then the triple (x, y, x) will be pushed onto *TSTACK*. If (x, y) is a tree link, then the triple $(y + nd(y) - 1, lowpt1(y), x)$ will be pushed onto *TSTACK*. In either case, as with the above case, the triple may be replaced, but eventually, the triple will satisfy the Type-2 test unless another Type-2 pair is found first.

Hence, if G has a Type-2 separation pair, then at least one of them is detected by the algorithm. ■

Lemma 5.10 *The distributed algorithm detects a separation pair if and only if there exists one in the graph.*

Proof: A direct consequence of Lemmas 5.6, 5.7, 5.8 and 5.9. ■

Theorem 5.11 *The distributed algorithm correctly identifies a subset of separation pairs that decompose the network G into split components.*

Proof: (By induction on the number of links in G)

Base case: Suppose G has exactly one link. By Lemma 5.10, the algorithms correctly reports no separation pair.

Induction hypothesis: Suppose for any network with less than m links, the distributed algorithm correctly identifies a subset of separation pairs that decompose the network G into split components.

Induction step: Let G be a network with m links. If G has no separation pair, then by Lemma 6.5, the algorithm correctly identifies no separation pair. Otherwise, by Lemma 5.10, the algorithm correctly identifies one separation pair of G. Let G_1 and G_2 be the two split components corresponding to the separation pair.

Since both G_1 and G_2 has less than $m - 1$ links, by the induction hypothesis, the algorithm correctly identifies a subset of separation pairs of G_1 (G_2, respectively) that decompose G_1 (G_2, respectively) into split components.

The separation pair that splits G and the two subsets of separation pairs for G_1 and G_2 form a subset of separation pairs that decompose the network G into split components. ■

Theorem 5.12 *During the third pass, the distributed algorithm takes $O(H^2)$ time and transmits a total of $O(|V| \cdot H)$ messages to identify a subset of separation pairs that decomposes G into split components, where $H(< |V|)$ is the height of the depth-first tree of G.*

Proof: At the beginning of the third pass, a decomposition of G into a collection of paths $P_i, 1 \leq i \leq k$, has been achieved. Let $P_i : v_1, v_2, ..., v_m$ be any of the paths such that $(v_i, v_{i+1}), 1 \leq i < m-1$, are tree links while (v_{m-1}, v_m) is a frond.

The algorithm identifies the separation pairs lying on P_i starting from v_{m-1}. Since *TSTACK* can grow from one entry to at most $\frac{H}{2}$ entries and transmitting the stack over a parent link is accomplished by transmitting one entry at a time, the total time spent on transmitting *TSTACK*, in the worst case, is thus $\sum_{j=1}^{\frac{H}{2}} j = O(H^2)$.

Since each **vlink**, **UpdateDegree**, **Deg_of_child_is_2** and **DecreaseDegree** message is transmitted from an ancestor (descendant, respectively) to a descendant (ancestor, respectively) along a path, the time required to transmit each of them is thus $O(H)$.

The total time required by the third pass is thus $O(H^2) + O(H) = O(H^2)$.

The size of *TSTACK* at any node is at most $\frac{H}{2}$. Therefore, there are a total of at most $O(\frac{H}{2} \cdot |V|) = O(|V| \cdot H)$ stack entries transmitted.

As for the **vlink**, **UpdateDegree**, **Deg_of_child_is_2** and **DecreaseDegree** messages, since each node transmits a constant number of each of them, the total number of such messages transmitted is $O(|V|)$.

The total number of messages transmitted in the third pass is thus $O(|V| \cdot H) + O(|V|) = O(|V| \cdot H)$. ■

Theorem 5.13 *The distributed algorithm correctly identifies a subset of separation pairs that decompose the network G into split components.*

Proof: By Theorems 5.1, $lowpt1(v), lowpt2(v), nd(v), \forall v \in V$ are correctly computed by the first pass. By Theorems 5.3 and 5.5, the acceptable adjacency structure is correctly constructed, $dfs(v), highpt(v), \forall v \in$

V are correctly computed and $lowpt1(v), lowpt2(v), \forall v \in V$ are correctly re-computed by the second pass. By Theorem 5.11, a subset of separation pairs that are sufficient to decompose the network into split components are correctly determined by the third pass. The theorem thus follows. ■

Theorem 5.14 *The distributed algorithm takes $O(|V| + H^2)$ time and transmits a total of $O(|E| + |V| \cdot H)$ messages, where H is the height of the depth-first tree of G.*

Proof: By Theorems 5.2, $lowpt1(v), lowpt2(v), nd(v), \forall v \in V$ can be computed in $O(|V|)$ time by transmitting $O(|E|)$ messages during the first pass. The acceptable adjacency structure can clearly be constructed in $O(1)$ time using no message.

By Theorems 5.4 and 5.5, $dfs(v), highpt(v), lowpt1(v)$, and $lowpt2(v), \forall v \in V$ can be correctly computed in $O(|V|)$ time by transmitting $O(|E|)$ messages.

By Theorem 5.12, a subset of separation pairs that is sufficient to decompose the network G into split components can be determined in $O(H^2)$ time using $O(|V| \cdot H)$ messages during the third pass. The distributed algorithm thus takes a total of $O(|V|) + O(|V|) + O(H^2) = O(|V| + H^2)$ time transmitting a total of $(|E|) + O(|E|) + O(|V| \cdot H) = O(|E| + |V| \cdot H)$ messages. ■

Chapter 6

Conclusion

We have presented a distributed algorithm for finding separation pairs of a network. The algorithm runs on an asynchronous computer model in $O(|V| + H^2)$ time and transmits a total of $O(|E| + H|V|)$ messages, where $H(< |V|)$ is the height of the depth-first search tree of G. In the worst case, $H = |V|$ which implies that our algorithm takes $O(|V|^2)$ time and transmits $O(|V|^2)$ messages. This is the first and only distributed algorithm known for triconnectivity.

The following are some possible future works:

- Develop a distributed algorithm for determining the split components: Since our algorithm finds a subset of separation pairs that is sufficient to determine all the split components, This should not be a difficult task. However, the challenge is that whether we can do it within the same time and message bounds of our algorithm.

- Improve the time or message complexity of the distributed algorithm: In the worst case, our algorithm has quadratic time and message complexities. It is worthwhile to consider whether it is possible to improve the time or message complexity to a sub-quadratic bound.

- Develop new method that could lead to more efficient distributed algorithms without using depth-first search: The sequential algorithm of Hopcroft el at. based on which we develop our distributed algorithm is depth-first search based. By contrast, parallel algorithms for triconnectivity on the various PRAM models are not depth-first search based. Since a network,

to a certain extent, is a parallel computer, it is thus worthwhile to investigate if the parallel algorithms for the PRAMS would lead to more efficient distributed algorithms.

- Present a distributed algorithm for 4-vertex- or 4-edge-connectivity: it is worthwhile to investigate whether our method for solving triconnectivity can be extended to solve 4-vertex-connectivity.

Bibliography

[1] James Abello and Adam L. Buchsbaum and Jeffery R. Westbrook ,A functional approach to external graph algorithms,*Algorithmica*, (32) 3 (2002) 437-458.

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman,The design and analysis of computer algorithms *Addison-Wesley*,(1974).

[3] Lars Arge and Norbert Zeh,I/O-Efficient Strong Connectivity and Depth-First Search for Directed Planar Graphs,*FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*,(2003),261.

[4] B. Awerbuch,A new distributed depth-first search algorithm,*Information Processing Letters*, (20) (1985), 147-150.

[5] M. Ahuja and Y. Zhu,An efficient distributed algorithm for finding articulation points, bridges and biconnected components in asynchronous networks,*9th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS 405*,(1989),99-108.

[6] Adam L. Buchsbaum et al.,On external memory graph traversal,*SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*,(2000),859-860.

[7] P. Chaudhuri,A note on self-stabilizing articulation point detection,*Journal of System Architecture*,(45) 14 (1999),305-329.

[8] P. Chaudhuri,An $O(n^2)$ self-stabilizing algorithm for computing bridge-connected components, *Computing*,(62) (1999), 55-67

[9] T. Cheung,Graph traversal technique and the maximum flow problem in distributed computation, *IEEE Trans. Software Engineering*,(9) (1983), 504-512.

[10] Yi-Jen Chiang and Michael T. Goodrich et al.,External-memory graph algorithms,*SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, (1995),139-149.

[11] Francis Y. Chin and John Lam and I-Ngo Chen,Efficient parallel algorithms for some graph problems, *Commun. ACM*,(25) 9 (1982),659-665.

[12] I. Cidon, Yet another distributed depth-first search algorithm,*Information Processing Letters*,(26) (1988),183-198.

[13] S. Devismes,A silent self-stabilizing for finding cut-nodes and bridges,*Parallel Processing Letters*,(15) (2005),183-198.

[14] D. Fussell, V. Ramachandran, R. Thurimella,Finding triconnected components by local replacement, *SIAM J. Computing*,(22) (1993),587-616.

[15] S.Even, Graph Algorithms,*Computer Science Press, Potomac, MD*,(1979).

[16] Harold N. Gabow,Path-based depth-first search for strong and biconnected components,*Inf. Process. Lett.*,(74) 3-4 (2000),107-114.

[17] Z. Galil and G. F. Italiano,Reducing edge-connectivity to vertex-connectivity,*SIGPLAN Notices* ,(22) (1991), 57-61.

[18] Carsten Gutwenger and Petra Mutzel,A linear time implementation of SPQR trees,*In Proceedings of the 8th International Symposium on Graph Drawing (GD00)*,(2001),70-90.

[19] W. Hohberg,How to find biconnected components in distributed networks,*Journal of Parallel and Distributed Computing*,(9) (1990),374-386.

[20] J E. Hopcroft and and R. E. Tarjan,Dividing a Graph into Triconnected Components,*SIAM J. Computing*,(2)3 (1973),135-158.

[21] E. Jennings and L. Motyckova., Distributed algorithms for sparse k-connectivity certificates.*In Proceedings of the Symposium on Principles of Distributed Computing (PODC)*,(1996),180. Jennings

and L. Motyckova. Distributed algorithms for sparse k-connectivity certificates. In Proceedings of the Symposium on Principles of Distributed Computing (PODC), page 180, 1996.

[22] A. Kanevsky and V. Ramachandran,Improved algorithm for graph four-connectivity,*JCSS*, (42) (1991),288-306.

[23] M. Karaata,A self-stabilizing algorithm for finding articulation points,*International Journal of Foundations of Computer Science*,(10) 1 (1999,33-46.

[24] M. Karaata,A stabilizing algorithm for finding biconnected components,*Journal of Parallel and Distribued Computing*,(62) (2002),982-999.

[25] A. Kazmierczak and S. Radhakrishnan,An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanar testing,*IEEE Transactions on Parallel and Distributed Systems*,(11) (2000),110-118.

[26] M. Karaata and P. Chaudhuri,A self-stabilizing algorithm for bridge finding,*Distribted Computing*,(2) (1999),47-53.

[27] Donald Ervin Knuth,The Art of Computer Programming, *2nd Ed. (Addison-Wesley Series in Computer Science and Information)*,(1978),

[28] ,V. Kumar and E. J. Schwabe,Improved Algorithms and Data Structures for Solving Graph Problems in External Memory,169-177.

[29] K. Lakshmanan and N. Meenakshi and K. Thulasiraman,A time-optimal message-efficient distributed algorithm for depth-first search,*Information Processing Letters*,(25) (1987),103-109.

[30] [28]S. MacLane,A structural characterization of planar combinatorial graphs,*Duke Math. J.*, (3) (1937),466-472.

[31] S. Makki and G. Havas,Distributed algorithm for depth-first search,*Information Processing Letters*,(60) (1996),7-12.

[32] Ulrich Meyer,Exernal memory BFS on undirected graphs with bounded degree,*SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*,(2001),87-88.

[33] Gary L. Miller and Vijaya Ramachandran,A new graph triconnectivity algorithm and its parallelization,*Combinatorica*,(12) (1992),53-76.

[34] Kameshwar Munagala and Abhiram Ranade,I/O-complexity of graph algorithms, *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, (1999),687-694.

[35] Hroshi Nagamochi and Toshihide Ibaraki, A Linear Time Algorithm for Computing 3-Edge-Connected Components in a Multigraph,*Japan J. Indust. Appl. Math.*,(8) (1992),163-180.

[36] J. Park and N. Tokura and T. Masuzawa and K. Hagihara,Efficient distributed algorithms solving problems about the connectivity of network,*Systems and Computers in Japan*,(22) (1991), 1-16.

[37] V. Ramachandran and U. Vishkin,Efficient parallel triconnectivity in logarithmic time,*VLSI Algorithms and Architectures, LNCS 319*,(1988),33-42.

[38] Jeffrey Scott ,Elizabeth A. M. Shriver.,Algorithms for Parallel Memory I: Two-Level Memories, *Technical report DUKE-TR-993-01*,(1993).

[39] M. Sharma and S. Iyengar and N. Mandyam,An efficient distributed depth-first search algorithm, *Information Processing Letters*,(32) (1989),183-186.

[40] B. Swaminathan and K. J. Goldman, Anincremental distributed algorithm for computing biconnected components in dynamic graphs,*Algorithmica*,(22) (1998),305-329.

[41] S. Taoka and T. Watanabe and K. Onaga,A Linear Time Algorithm for Computing all 3-Edge-Connected Components of a Multigraph,*IEICE Trans. Fundamentals*,(75) 3 (1992),410-424.

[42] Robert Endre Tarjan,Depth-First Search and Linear Graph Algorithms,*SIAM J. Comput.*, (1) 2 (1972),146-160.

[43] Robert Endre Tarjan and Uzi Vishkin,An Efficient Parallel Biconnectivity Algorithm, *SIAM J. Comput.*,(14) 4 (1985),862-874.

[44] Y. H. Tsin,A Simple 3-edge-connected Component Algorithm,*Theory of Computing Systems*, (40) 2 (2007),125-142.

[45] Y. H. Tsin,An Efficient Distributed Algorithm for 3-edge-connectivity,*International Journal of Foundations of Computer Science*,(17) 3 (2006),677-701.

[46] Yung H. Tsin,An improved self-stabilizing algorithm for biconnectivity and bridge-connectivity, *Information Processing Letters*,(102) 1 (2007),27-34.

[47] Y.H.Tsin, Some remarks on distributed depth-fist search, *Information Pocessing Letters*,(82) (2002),173-178.

[48] Y. H. Tsin,Yet another optimal Algorithm for 3-edge-connectivity,*Journal of Discrete Algorithms*,(7)1 (2009),130-146.

[49] Yung H. Tsin and Francis Y. Chin,Efficient parallel algorithms for a class of graph theoretic problems

[50] Volker Turau,Computing bridges, articulations and 2-connected components in wireless sensor networks,*Algorithmic Aspects of Wireless Sensor Networks, Second International Workshop ALGOSENSORS 2006, LNCS 4240*,(1989),164-175.

[51] Jeffrey D. Ullman and Mihalis Yannakakis,The input/output complexity of transitive closure, *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*,(1990),44-53.

[52] Robin J Wilson,Introduction to graph theory,(1986).

[53] A functional approach to external graph algorithms,James Abello and Adam L. Buchsbaum and Jeffery R. Westbrook,*Algorithmica*,(32) 3 (2002), 437-458.

Vita Auctoris

Katayoon Moazzami obtained her bachelor degree in pure mathematics from Iran University of Science and Technology in 2005. She joined the master program in computer science at university of Windsor in 2006 and graduated in 2009.