University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

2010

Finding 3-edge-connected components in parallel

Yuan Ru
University of Windsor

Recommended Citation

Ru, Yuan, "Finding 3-edge-connected components in parallel" (2010). *Electronic Theses and Dissertations*. 7917.
https://scholar.uwindsor.ca/etd/7917

FINDING 3-EDGE-CONNECTED COMPONENTS IN PARALLEL

by
Yuan Ru

A Thesis
Submitted to the Faculty of Graduate Studies
through School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2010

Your file Votre référence
ISBN: 978-0-494-62727-3
Our file Notre référence
ISBN: 978-0-494-62727-3

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

A parallel algorithm for finding 3-edge-connected components of an undirected graph on a CRCW PRAM is presented. The time and work complexity of this algorithm is $O(\log n)$ and $O((m+n)\log\log n)$, respectively, where n is the number of vertices and m is the number of edges in the input graph. The algorithm is based on ear decomposition and reduction of 3-edge-connectivity to 1-vertex-connectivity. This is the first 3-edge-connected component algorithm of a parallel model.

Acknowledgements

I greatly appreciate the help from my supervisor, Dr. Yung H. Tsin. During the whole process of this writing thesis, he gave me countless valuable suggestions and insights. Without his help, I could not have been able to achieve my goal of completing this thesis. Also I would like to thank Dr. R. M. Barron and Dr. Dan Wu for their support and encouragement.

I acknowledge the financial support by my supervisor, Dr. Yung H. Tsin, in the form of research assistantship through **NSERC**, the School of Computer Science in the form of graduate assistantship. A few words cannot express my appreciation of these people and entities.

I also need to thank my wife Jenny Liu for her understanding during the whole writing period.

Contents

List of Figures

Chapter 1

Introduction

Graph connectivity (vertex-connectivity and edge-connectivity) is a fundamental subject in graph theory and has been studied extensively. A connected graph $G = (V,E)$ is k-edge (k-vertex) connected if removing $(k-1)$ or fewer edges (vertices) will leave the graph connected. Graph connectivity has applications in a wide variety of areas such as network reliability, DNA construction, DNA computation, quantum physics and chemistry where the Feynman diagram is used [31]. Because of its importance, graph connectivity has been explored extensively in the last few decades on different computational models, especially on the sequential and parallel models.

In this thesis, an efficient algorithm for finding 3-edge-connected components of an undirected graph is presented. The algorithm runs on the *Arbitrary-CRCW-PRAM* model. For any graph with m edges and n vertices, the algorithm runs in $O(\log n)$-time using $O(((m+n)\log\log n)/\log n)$ processors and performing $O((m+n)\log\log n)$ work. This is the first algorithm for finding 3-edge-connected components on the *Arbitrary-CRCW-PRAM* model.

In addition to its applications in the traditional areas such as network reliability, 3-

edge-connectivity also has important applications in quantum physics and chemistry as is explained below.

In quantum physics and chemistry, a Feynman diagram consists of a set of vertices and a set of edges. The edges can be partitioned into two types: V-edges and G-edges. The V-edges are undirected edges while the G-edges are directed edges. Every vertex in the diagram is incident with exactly three edges: one V-edge, one G-edge of which the vertex is the tail and one G-edge of which the vertex is the head. A Feynman diagram is irreducible if it cannot be disconnected by removing fewer than three G-edges. Let F be a Feynman diagram and L$\widetilde{F}$ be the undirected graph obtained from F by contracting the V-edges into a vertex and treating the G-edges as undirected edges. Then F is irreducible if and only if $\widetilde{F}$ is 3-edge-connected. In quantum Monte Carlo simulation, it is necessary to determine if a Feynman diagram is irreducible [31].

Recently, in bioinformatics, a data structure, called a *cactus graph*, had been introduced to capture the nested structure of genome comparisons [33]. The cactus graph is built from an adjacency graph G_0 in a series of steps. First, the connected components of G_0 formed by the adjacency edges are determined. Then pseudo adjacency edges are added to produced a graph G_1 representing a decomposition of G_0. In G_1, two vertices x and y are equivalent if it takes the removal of three or more edges to disconnect them. The equivalence classes of vertices are thus the *3-edge-connected components*. A graph G_2 is constructed to represent this decomposition. It has one vertex for each 3-edge connected component. The theory of graph decomposition into 3-edge-connected components shows that G_2 is a cactus graph in the combinatorial sense. Finally, to construct the cactus graph, the tree-like structures in G_2 are folded to obtain an Eulerian cactus graph.

Chapter 2

Related Works

On the sequential model, for undirected graphs, linear-time algorithms are known only for $k = 2, 3$. Tarjan [38] presented a very simple and elegant linear-time sequential algorithm for finding 2-vertex-connectivity. This technique is based on a powerful graph traversal technique, called depth-first search, devised by Hopcroft and Tarjan [19]. The depth-first search technique was also used by Tarjan [39] to solve the st-numbering problem. Gabow [11] revisited depth-first search from a path-view perspective and designed a new elegant linear-time algorithm for 2-vertex-connectivity and 2-edge-connectivity. For 3-vertex-connectivity, the problem was first studied by Hopcroft and Tarjan [19] in 1973. They presented a rather complicated linear-time algorithm. In 2001, Gutwenger and Mutzel [15] presented a list of errors in the algorithm of Hopcroft and Tarjan and showed how to correct them. Unfortunately, their explanation was brief and incomplete. For 3-edge-connectivity, the first linear-time algorithm was presented by Galil and Italiano [12]. Their method is to reduce 3-edge-connectivity to 3-vertex-connectivity in linear time and then use Hopcroft and Tarjan's 3-vertex-connectivity algorithm to solve the problem. This algorithm is rather complicated and difficult to implement. Two simpler linear-time algorithms were then reported by Taoka et al [37] and Nagamochi and Ibaraki [30]. Both algorithms are based on depth-first search. The algorithm of Nagamochi and Ibaraki [30] uses graph transformation technique. However, they use three different types of graph transformations and perform multiple depth-first search over the input graph. The algorithm is complicated and hard to implement. The algorithm of Taoka et al [37] computes the 3-edge-connected components in three phrases with four depth-first search. However, the algorithm is simpler than the previous two algorithms and is easier to implement. Both Nagamochi et al [30] and Taoka et al [37] classify the cut-pairs into two types, type-1 and type-2, and determine them separately. Tsin [43] presented a very simple and elegant linear-time algorithm for finding 3-edge-connected components. This algorithms does not distinguish between type 1 and type 2 cut-pairs. It use a novel graph transformation technique, called absorb-eject, to transform the given graph so that every 3-edge-connected component is transformed into a single vertex, called a super-vertex, which is then released to generate the 3-edge connected component. The algorithm is conceptually simple and is easy to implement. Tsin [44] also presented another linear-time algorithm for finding a set of cut-pairs whose removal leads to the 3-edge-connected components. This algorithm is also simple and easy to implement. An empirical study [44] shows that this algorithm outperforms all the other algorithms in finding cut-pairs and in determining if a graph is 3-edge-connected whereas the algorithm of Tsin [43] outperforms the rest in determining 3-edge-connected components. For $k = 4$, no linear-time algorithm has been reported so far. Only an $O(n^2)$-time algorithm (n is the number of vertices in graph G) has been reported. Moreover, this algorithm only determines if a graph is 4-vertex-connected.

On the parallel computer models, the k-edge (k-vertex) connectivity problems have also received great attention. The most popular parallel computer model is the **PRAM (Parallel**

Random Access Machine). A PRAM is a parallel computer in which n processors have access to a common memory, called the shared memory. The PRAM is a SIMD (Single Instruction Multiple Data) machine. Specifically, at any point of time during the execution of a program, every processor in the PRAM executes the same instruction but on different data stored in the shared memory. The processors are synchronized. Depending on whether more than one processor is allowed to read from or write into the same memory location in the shared memory, the PRAM can be classified into the following types:

EREW (Exclusive-Read-Exclusive-Write): At any time, only one processor is allowed to read from a memory location and only one processor is allowed to write into a memory location.

CREW (Concurrent-Read-Exclusive-Write): More than one processor is allowed to read from the same memory location at the same time but only one processor is allowed to write into a memory location at any time.

ERCW (Exclusive-Read-Concurrent-Write): Only one processor is allowed to read from a memory location at any time but more than one processor is allowed to write into a memory location at any time.

CRCW (Concurrent-Read-Concurrent-Write): More than one processor is allowed to read from the same memory location at the same time and more than one processor is allowed to write into a memory location at the same time.

Depending on how write-conflicts are handled, this PRAM model has been further classified as follows:

CRCW-common: all processors writing into the same memory location must write the same thing; only one processor will succeed and we don't know which one.

CRCW-arbitrary: the processors writing into the same memory location may write different things. However, only one processor succeeds and we don't know which one.

CRCW-priority: the processors in the PRAM are given different priorities. When more than one processor attempt to write into the same memory location, only the one that has the highest priority succeeds.

In the following, m and n are the number of edges and vertices in the input graph, respectively.

While depth-first search had been successfully used in designing optimal (linear-time) algorithms for various graph connectivity problems, it has not been successful in designing efficient algorithms, let alone optimal algorithms, for parallel computers. This is due to the fact that the technique is inherently sequential.

A parallel algorithm that use $O(n^2)$ processors to find the connected components (1-vertex-connected) of an undirected graph in $O(\log^2 n)$-time on an CREW-PRAM was first reported in [17]. Later, Hirschberg et al [17] showed that the $O(\log^2 n)$-time bound can also be achieved using only $n\lceil n/logn \rceil$ processors. Chin et al [4] further improved the bounds to $O(n^2/K + \log^2 n)$-time using $K(> 0)$ processors. Note that when $K = O(n^2/\log^2 n)$, this algorithm achieves the $O(\log^2 n)$-time bound using only $O(n^2/\log^2 n)$ processors. An almost optimal algorithm for finding connected components has been developed by Cole and Vishkin [6]. It runs in $O(\log n)$-time using $O((m+n)\alpha(m,n)/\log n)$ processors, where α is the inverse Ackermann function. However, this algorithm runs on the stronger CRCW-PRAM model.

Biconnectivity (2-vertex-connected) and bridge-connectivity (2-edge-connected) were first studied by Savage and Ja'Ja' [35]. The parallel algorithms they presented uses $O(\log^2 n)$-

time and $O(n^3/\log n)$ processors and runs on a CREW-PRAM. Later, Tsin and Chin [45] presented optimal algorithms that run in $O(n/K + \log^2 n)$-time with $nK(K \geq 1)$ processors on the same model. When $K = O(n/\log^2 n)$, these algorithms achieve the $O(\log^2 n)$-time bound using only $O(n^2/\log^2 n)$ processors. As the processor-time product, also called *work*, is $O(n^2)$, the algorithms do optimal work for dense graphs. Tarjan and Vishkin [40] developed a parallel implementation for finding biconnected components that runs in $O(\log n)$-time using $O(m+n)$ processors. However, the algorithm runs on the stronger CRCW-PRAM model.

For triconnectivity (3-vertex-connectivity), several algorithms have been developed. The algorithms reported in [21] and [14] use a parallel algorithm for matrix multiplication as subroutine; hence their algorithms are far from optimal. Major progresses were made by Miller and Ramachandran [13] and Ramachandran and Vishkin [34]. The former presented an algorithm that runs in $O(\log^2 n)$-time on the CREW-PRAM while the later presented an algorithm that runs in $O(\log n)$-time on a CRCW-PRAM. Later, Fussell, Thurimella and Ramachandran [9] came up with a parallel algorithm for finding separation pairs whose time and processor complexity are $O(\log n)$ and $O(m+n)$, respectively, on the CRCW-PRAM. Fussell et al [9] used a local replacement technique to successfully improve the processor bound. Specifically, their algorithm runs in $O(\log n)$-time using $O((m+n)\log\log n/\log n)$ processors. No parallel algorithm has been reported for 3-edge-connectivity. For 4-vertex-connectivity, a parallel algorithm for the Arbitrary-CRCW-PRAM was reported by Kanevsky and Ramachandran [23]. This algorithm runs in $O(\log^2 n)$-time using $O(n^2)$ processors.

Graph connectivity is a natural way of measuring the robustness and reliability of a computer or communication network. The subject has thus been extensively studied on the distributed computer model. For biconnectivity and bridge-connectivity, a number of algorithms that run in $O(n)$-time and transmit $O(m)$ messages of $O(\log n)$ length have been proposed [1, 18, 27, 32, 36]. For 3-edge-connectivity, Jennings et al [22] presented the first algorithm that runs in $O(n^3)$-time transmitting $O(n^3)$ messages [22]. Tsin [41] improved both the time and message bounds to $O(n^2)$. No 3-vertex-connectivity algorithm has been reported so far.

Fault-tolerance is a very important issue in computer network. The concept of self-stabilization is a concept introduced by Dijkstra [8] to handle transient faults on distributed computer. For bridge-connectivity, Karaata and Chaudhuri [26] presented the first self-stabilizing algorithm. However, their algorithm must run concurrently with a self-stabilizing breadth-first spanning tree algorithm. The algorithm runs in $O(mn^2)$ steps and $O(dm)$ rounds(for the definition of *rounds*, please see [26] for details), where $d(< n)$ is the diameter of the network. Chaudhuri [2] presented another algorithm that must run concurrently with a self-stabilizing depth-first spanning tree algorithm and requires only $O(n^2)$ steps and $O(d)$ rounds. For biconnectivity, Karaata [24] presented the first self-stabilizing algorithm for finding cut-vertices. His algorithm must run concurrently with a self-stabilizing breadth-first spanning tree algorithm and a self-stabilizing bridge finding algorithm. The algorithm requires $O(mn^2)$ steps and $O(dm)$ rounds. Chaudhuri [3] improved the bounds to $O(n^2)$ steps and $O(d)$ rounds by presenting an algorithm that runs concurrently with a self-stabilizing depth-first spanning tree algorithm. Karaata [25] presented a self-stabilizing algorithm that finds all the biconnected components in $O(d)$ rounds using $O(n\triangle \log \triangle)$ bits per processor, where $\Delta(< n)$ is the largest degree of a vertex in the network. His algorithm must run concurrently with a self-stabilizing breadth-first spanning tree algorithm and a self-stabilizing bridge finding algorithm. Devismes [7] improved the bounds to $O(H)$

moves(for the definition of *moves*, please see [7] for details) ($H(< n)$ is the height of the spanning tree) and $O(n \log \triangle)$ bits per processor, with the assumption that a breadth-first search or depth-first search spanning tree is available. Tsin [42] improved all of the above results by presenting an algorithm that finds all the bridges, cut-vertices, bridge-connected components and biconnected components in $O(dn \log \triangle)$ rounds using $O(n \log \triangle)$ bits per processor. Moreover, in contrast with the above algorithms, this algorithm does not assume the existence of any spanning tree.

In the wireless sensor network setting, Turau [46] presented an algorithm that takes $O(n)$-time and transmits at most $4m$ messages.

Chapter 3

Definitions

A ***graph*** is a triple $G = (V,E)$ in which V and E are two disjoint finite sets such that $V \neq \phi$. Each element of V is called a ***vertex*** of G, and each element of E is called an ***edge*** of G.

Let $v_1 e_1 v_2 ... v_{k-1} e_{k-1} v_k$ be a sequence of alternating vertices and edges such that $v_i \in V$, $1 \leq i \leq k$, and $e_i = (v_i, v_{i+1}) \in E$, $1 \leq i < k$. We shall also denote the edge e_i by $v_i \dashrightarrow v_{i+1}$. The sequence is a ***path*** P in G if the vertices v_i, $1 \leq i \leq k$, are distinct; the sequence is a ***cycle*** in G if $v_1 = v_k$ and the vertices v_i, $1 \leq i < k$, are distinct, where $k > 1$. The subscript i is called the ***index*** of vertex v_i on the path P. We assume that on each path P, the index of every vertex is distinct. When the sequence is a path, then v_1, v_k are the ***end vertices***, and each $v_i, 1 < i < k$, is an ***internal vertex***. We shall also denote the path by $v_1 \rightsquigarrow v_k$. The path is a ***null*** path if $k = 1$. A ***self-loop*** is an edge that connects a vertex to itself. A ***u-v walk*** is a sequence of vertices starting at u and ending v, where every two consecutive vertices in the sequence are adjacent in the graph. A ***closed walk*** is a walk when the first and last vertices are the same. The set of edges (vertices, respectively) on a path P is denoted by $E(P)$ ($V(P)$, respectively). $G - E'$, where $E' \subseteq E$, is the graph resulting from G after the edges in E' are removed. A graph G is ***connected*** if there is a path between every two vertices;

it is ***disconnected*** otherwise. A ***connected component*** is a maximal connected subgraph of G. An edge e is called a ***bridge*** of G if $G-e$ is disconnected. If a connected graph G contains no bridge, then G is called a ***bridgeless*** graph or a ***2-edge-connected*** graph. A vertex $v \in V$ is called ***cut-vertex*** if $G-v$ is disconnected. A graph without a cut-vertex is ***biconnected*** and is also called ***block***. A pair of edges $\{e,e'\} \in E$ is called a ***cut-pair*** of a bridgeless graph G if $G-\{e,e'\}$ is disconnected. A bridgeless graph without cut-pairs is a ***3-edge-connected*** graph. Let $G=(V,E)$ and $G'=(V',E')$ be two simple disjoint graphs, then the ***union*** of G and G' is the graph $G \cup G' = (V \cup V',\ E \cup E')$. G' is called a ***spanning subgraph*** of G if $V=V'$ and G' is a subgraph of G.

An ***ear decomposition*** D of an undirected graph $G=(V,E)$ is a partition of E into a set of edge-disjoint paths $D_0,D_1,\ldots D_i,\ldots D_n$ such that D_0 is a cycle on which a vertex, r, that has the smallest index value, is designated as the ***root***, and for every $D_k, k \geq 1$, each end vertex of D_k is a vertex on some $D_j, j<k$. We say that ear D_i is ***smaller*** than D_j if $i<j$. If there is only one edge in D_i, then D_i is a ***trivial*** ear; otherwise, it is a ***non-trivial*** ear. Two ears are ***parallel*** if they have the same end vertices. The ***distance*** between two vertices on an ear is the number of edges between them on the ear. The following defintion is from [10]. Starting with the end vertex p of D_i with the smaller index, define $\boldsymbol{pos}(p,D_i)$ to be zero. $\forall v \in V(D_i)-\{p\}$, $\boldsymbol{pos}(v,D_i)$ is the distance from p to v on D_i. The value of $pos(w,D_i)$, for $w \notin V(D_i)$ is undefined. For the sake of consistency, we label the vertices of an ear in D by their *pos* values. Specifically, if v_l, v_j are two vertices on the ear D_i, then l and j are the *pos* value of v_l and v_j, respectively. We use $D_i(e,e')$ ($D_i[e,e']$, respectively) to denote the portion of ear D_i between edges e and e', exclusive (inclusive, respectively) of these edges. $D_i(v,v')$ ($D_i[v,v']$, respectively) denotes the portion of ear D_i between vertex v and v', exclusive (inclusive, respectively) of these vertices. $D_i - D_i[e,e']$ refers to the segment

of D_i whose edge set is $E(D_i) - E(D_i[e, e'])$. ***Remark***: In this thesis, we assume without loss of generality that the input graph G is 2-edge-connected.

Chapter 4

Some Properties of Cut-Pairs

Lemma 1. *Let $G = (V,E)$ be a connected graph. An edge e of G is a bridge if and only if e does not lie on any cycle in G.*

Proof. See [43]. $\square$

Lemma 2. *An undirected graph G has an ear decomposition if and only if it is 2-edge connected.*

Proof. See [47]. $\square$

Lemma 3. *Let $G = (V,E)$ be a connected graph and let $e, e' \in E$. Then $\{e, e'\}$ is a cut-pair if and only if e $(e', respectively)$ is a bridge in $G - e'$ $(G - e, respectively)$.*

Proof. See [43] $\square$

Lemma 4. *Let $G = (V,E)$ and D be an ear decomposition of G. Then for any edge $e \in \bigcup_{j=1}^{i-1} D_j$, there exists a cycle containing e without passing through any edge in ear D_i.*

Proof. (Proof by induction)

Base case: when $i = 1$, according to the definition of ear decomposition, D_0 is a cycle C containing every edge in it.

Suppose the Lemma holds true for $i = k-1$.

Consider $i = k$. Let $e \in \bigcup_{j=0}^{k-1} D_j$. If $e \in \bigcup_{j=0}^{k-2} D_j$, then by the induction hypothesis, e lies on a cycle in $\bigcup_{j=0}^{k-2} D_j$ without passing through any edge in ear D_{k-1}. Clearly, the cycle is also a cycle in $\bigcup_{j=0}^{k-1} D_j$, without passing through any edge on D_k.

If e lies on D_{k-1}, let the two end-vertices of ear D_{k-1} be lying on the ears D_u and D_v, where $u, v < k-1$. By a simple induction, it is easily verified that $\bigcup_{j=0}^{k-2} D_j$ is a connected graph. As a result, there is a path P in it connecting the two end-vertices of D_{k-1}. It follows that the path P and ear D_{k-1} forms a cycle containing e in $\bigcup_{j=0}^{k-1} D_j$ and this cycle does not use any edge on ear D_k. $\square$

Lemma 5. *Let $G = (V, E)$ and D be an ear decomposition of G. If $\{e, e'\}$ is a cut-pair of G, then there exists a non-trivial ear $D_i \in D$ that contains both e and e'.*

Proof.

Since D is a partition of E, every edge belongs to one and only one ear.

Let $\{e, e'\}$ be a cut-pair such that $e \in D_i$ and $e' \in D_j$. We want to prove that $i = j$.

Suppose to the contrary that $i \neq j$, without loss of generality, we assume $i < j$. By Lemma 4, there exists a cycle in $\bigcup_{k=0}^{j-1} D_j$ containing e but not e'. But then e is not a bridge in $G - e'$, which contradicts Lemma 3. $\square$

Theorem 1. *Let $G = (V, E)$ be an undirected graph and r be the root of an ear decomposition D of G. Let $D_i \in D$ and $e, e' \in E(D_i)$ such that $e = (a, b)$, $e' = (a', b')$. Furthermore, $pos(a, D_i) < pos(b, D_i) < pos(a', D_i) < pos(b', D_i)$. Then $\{e, e'\}$ is a cut-pair of G if and only if it satisfies both the following conditions:*

1. *There does not exist a path $P : m \rightsquigarrow r$, where $m \in V(D_i[b,a'])$, and $V(P) \cap V(D_i) = \{m\}$.*

2. *Let $Q = \{w \in D_i | \exists$ path P': $V(P') \cap V(D_i) = \{w,m\}$, where $m \in V(D_i[b,a'])\}$. If Q is not empty, let $c \in Q$ such that $pos(c,D_i) \leq pos(u,D_i)$ and $d \in Q$ such that $pos(d,D_i) \geq pos(u,D_i), \forall u \in Q$. Then $pos(b,D_i) \leq pos(c,D_i) \leq pos(d,D_i) \leq pos(a',D_i)$.*

Proof.

1. **(Only if)** Since $\{e,e'\}$ is a cut-pair of G, $G - \{e,e'\}$ contains at least two connected components, namely C_1 and C_2. Furthermore, $D_i - D_i[e,e']$ and $D_i(e,e')$ cannot be in the same connected component, otherwise, there would exist a path P_1 from a vertex $x \in V(D_i - D_i[e,e'])$ to some vertex $y \in V(D_i(e,e'))$ in $G - \{e,e'\}$. This path and the portion of D_i between x and y, called it P_2, form a closed walk W. Owing to the fact that P_1 does not contain e or e' and P_2 contains only one of e and e', without loss of generality, we assume P_2 contains e. As W is a closed walk, there must exist a cycle, C, in W that contains e. Since C does not contain e', therefore C is a cycle in $G - \{e'\}$. By Lemma 1, e is not a bridge in $G - \{e'\}$ which contradicts Lemma 3.

   Now, let $D_i - D_i[e,e']$ be in C_1 and $D_i(e,e')$ be in C_2.

   For condition 1, suppose to the contrary that there exists a path $m \rightsquigarrow r$, where $m \in [b,a']$. Let $G' = \bigcup_{j=0}^{i-1} D_j$ (i.e. G' is a subgraph of G composing of the ears $D_j, 0 \leq j < i$). It is easily verified that G' is a connected graph. Let v' be one of the end vertices of D_i, then v' is a vertex in G'. As r is also a vertex in G' and G' is connected, therefore, there is a $r \rightsquigarrow v'$ path connecting r and v' in G'. The paths $m \rightsquigarrow r$ and $r \rightsquigarrow v'$ form an $m \rightsquigarrow v'$ path. Since $m \in V(C_1)$ while $v' \in V(C_2)$, we thus have $C_1 = C_2$ which contradicts $C_1 \neq C_2$.

For condition 2, suppose to the contrary that $pos_{v'}(c,D_i) < pos_{v'}(b,D_i)$, then $c \in Q$ implies that there is a path P' from m to c without passing through e and e'. It follows that P' connects m and c in $G - \{e,e'\}$. But $c \in V(C_1)$ and $m \in V(C_2)$, therefore $C_1 = C_2$ which contradicts $C_1 \neq C_2$. By a similar argument, $pos_{v'}(d,D_i) \geq pos_{v'}(a',D_i)$.

2. **(If)** Suppose $\{e,e'\}$ is not a cut-pair. Then by Lemma 3, e is not a bridge in $G - \{e'\}$. By Lemma 1, e lies on a cycle C. If C completely lies within $\bigcup_{j=0}^{i} D_j$, let m be a vertex such that $m \in V(D_i[b,a'])$. Then there must exist a path $P : m \rightsquigarrow r$ in $\bigcup_{j=0}^{i} D_i$, hence in G, such that $V(P) \cap V(D_i) = \{m\}$. Condition 1 is thus violated.

   On the other hand, if the cycle C contain an edge outside $\bigcup_{j=0}^{i} D_j$, then $\exists$ a path P' : $V(P') \cap V(D_i) = \{w,m\}$, where $m \in V(D_i[b,a'])\}$ such that $pos(w,D_i) < pos(b,D_i)$. Condition 2 is thus violated.

   The case in which e' is not a bridge in $G - \{e\}$ can be proved in a similar way.

$\square$

Chapter 5

Finding 3-edge-connected Components

Given $G = (V, E)$, let D be an ear decomposition of G. We define $G' = (V', E')$ as follows: $V' = \{v'_i \mid D_i \in D\}$, vertex v'_i, $1 \leq i \leq n$, is called the ***image*** of D_i; $E' = \{e' = (v'_i, v'_j) \mid \exists e = (u_h, w_l) \in E$ such that $u_h \in V(\widetilde{D}_i) \wedge w_l \in V(\widetilde{D}_j) \wedge i \neq j\}$; edge e is called the ***corresponding edge*** of e' in G. Moreover, for every edge $e' = (v'_i, v'_j) \in E'$, where $i < j$, a 2-tuple ***λ-value*** is associated with e', denoted by $\lambda_{e'}$, such that $\lambda_{e'}[1] = h$ and $\lambda_{e'}[2] = l$. Let $B_1, B_2, \ldots, B_l$ be the blocks of G'.

```
**Input**: A bridgeless graph $G = (V,E)$
**Output**: A graph $G' = (V',E')$

Find an ear decomposition $D$ of $G$.
for each ear $D_i \in D$ do in parallel
    create a new vertex $v'_i \in V(G')$.
Endpar
for each edge $e = (x,y) \in E(G)$ do in parallel
    if ($x = u_h \in V(\tilde{D}_i)$ and $y = w_l \in V(\tilde{D}_j)$, where $i < j$ ) then
        create a new edge $e' = (v'_i, v'_j) \in E(G')$.
        $\lambda_{e'}[1] = h$.
        $\lambda_{e'}[2] = l$.
Endpar
```

Algorithm 1: Building G' for graph G

Lemma 6. *There does not exist a bridge in the graph $G' = (V',E')$.*

Proof.

In order to prove that there is no bridge in the graph G', we shall prove that for each edge $e' \in E(G')$, there exists a cycle containing e'. Let $e' = (v'_i, v'_j)$, where v'_i (v'_j) is the image of ear D_i (D_j, respectively), and edge $e = (u,v)$ be the corresponding edge in graph G, where $u \in \tilde{D}_i \wedge v \in \tilde{D}_j$. Since G is a bridgeless graph, by Lemma 1, we know that e lies on a cycle C in G.

Let $P^k_{x,y}$ denote the ear portion between vertices x and y, exclusive, on the ear D_k, where $x \in V(D_k)$, $y \in V(D_k)$ and $D_k \in D$. Then the cycle C can be represented as follows:

$u, P^i_{u,w_1}, w_1, P^d_{w_u,w_2}, w_2, \ldots P^a_{w_{h-1},w_h}, w_h, P^b_{w_h,w_{h+1}}, \ldots, w_v P^j_{w_v,v}, v, e, u$, where $i \neq d \neq a \neq b \neq j$.

For the segment, $P^a_{w_{h-1},w_h}, w_h, P^b_{w_h,w_{h+1}}$, of cycle C, we want to show that v'_a, v'_b are connected in G', if D_a and D_b are non-trivial ears. [**Remark**: If D_a or D_b is a trivial ear, since the end vertices of D_a or D_b are the internal vertices of some other ears, say D_c and D_d respectively. So, instead of proving v'_a and v'_b are connected, we can prove v'_c and v'_d are connected.]

The following cases are to be considered.

- If $w_h \in \widetilde{D}_a$, then by the definition of G', an edge $e' = (v'_a, v'_b)$ is in G'.

- If w_h is one of the end vertices of ear D_a, by the definition of ear decomposition, w_h is an internal vertex of another ear D_k. If $k \neq b$, then edges $e_1 = (v'_a, v'_k)$ and $e_2 = (v'_k, v'_b)$ exist in G'. It follows that e_1 and e_2 form a path $v'_a \rightsquigarrow v'_b$ in G'. On the other hand, if $k = b$, then an edge $e' = (v'_a, v'_b)$ exists in G'.

From the above argument, it is easily verified that for a cycle C in G, if it passes through two non-trivial ears, then there exists a path in G' connecting the images of these two ears; if it passes through at least one trivial ear, then there exists a path in G' between the images of the ears that contain those end vertices as internal vertices. As a result, if $e = (u, v)$, where $u \in \widetilde{D}_i \wedge v \in \widetilde{D}_j$ lie on a cycle in G, then there corresponds a cycle C' in G' containing $e' = (v'_i, v'_j)$. Hence, e' lies on a cycle in G'. □

Lemma 7. *Let v'_i, v'_j be the images of ear D_i and D_j, respectively. If $i < j$, then there exists a path between root r and v'_i without passing through v'_j.*

Proof. Immediate from Lemma 4. □

Theorem 2. *Let $D_i, D_j \in D$ such that $i < j$ and $v'_i, v'_j \in V(B_k)$, where B_k is a block in G', $1 \leq k \leq l$. Then any path P starting from a vertex $v \in V(D_j)$ must pass through some* *internal vertex $u \in V(D_i)$ before reaching some vertex $x \in V(D_z)$, $z < i$ if and only if v'_i has the smallest index among all the vertices in the block B_k.*

Proof.

1. **(If)**

   Since $v \in V(D_j)$, the assertion is equivalent to proving that any path P starting from v'_j must pass through vertex v'_i before reaching some vertex v'_z, $z < i$. Suppose to the contrary that there exist a path P starting from v'_j which does not pass through v'_i before reaching some vertex v'_z, $z < i$. We assume that v'_z lies on a block B_w. Owing to the fact that v'_i has the smallest index value in block B_k and the image of any ear that is smaller than D_i cannot be in B_k, so v'_z cannot be in the block B_k. Since $v'_i, v'_j \in V(B_k)$, there exists a cycle C containing v'_i and v'_j.

   Let v'_h be the common vertex of P and C such that no vertex following v'_h on P lies on C. Let v'_s be the first vertex on P which lies on the section $v'_h \rightsquigarrow v'_z$ such that $v'_s \notin V(B_k)$ while the vertex preceding it on P does belong to $V(B_k)$. Since $\bigcup_{j=0}^{i} D_j$ is a connected graph containing the ears D_z and D_i, there must be a path, P_1, connecting v'_i with v'_z outside B_k. Let v'_t be a common vertex of P_1 and the section $v'_s \rightsquigarrow v'_z$ that is closest to v'_s. Then, the section $v'_i \rightsquigarrow v'_h$ on C, the section $v'_h \rightsquigarrow v'_s$ on path P, the section $v'_t \rightsquigarrow v'_i$ on P_1 form a cycle containing v'_i and v'_s in G'. As a result, $v'_s \in V(B_k)$ which contradicts the above assumption made on v'_s.

2. **(Only if)** Suppose to the contrary that v'_i does not have the smallest index among all the vertices in the block B_k. Let us assume that v'_k has the smallest index in B_k. Since $k < i$, by Lemma 4, there exists a cycle in $\bigcup_{j=0}^{i-1} D_j$, hence in G, that contains D_k and not D_i. Since $v'_j, v'_k \in V(B_k)$, that means v'_j and v'_k lie on a common cycle. It follows

that there are at least two paths connecting v'_j and v'_k in B_k. Therefore, there is a path P_1 connecting them without going through v'_i. Moreover, since $k, z < i$, $\bigcup_{j=0}^{i-1} D_j$ is a connected graph containing the vertices v'_k and v'_z. As a result, there is a path P_2 connecting v'_k and v'_z without passing through v'_i. The paths P_1 and P_2 form a walk, hence a path connecting v'_j and v'_z but bypassing v'_i. This contradicts the assumption.

□

Corollary 1. *Let $G = (V, E)$ and D be an ear decomposition of G. If $B_i, 1 \leq i \leq l$, is a biconnected component of $G' = (V', E')$, then the vertex of B_i with the smallest index value is either the root r or a cut-vertex of G'.*

Proof.

Let B_i be a block of G' and v'_k be the vertex in it that has the smallest index value. Suppose v'_k is not the root r. Then $r \notin V(B_i)$. Suppose to the contrary that v'_k is not a cut vertex. Then $G' - v'_k$ is a connected graph which implies that for any other vertex in block B_i, there is path connecting it with the root r without passing through v'_k. But $r = v'_0$ and $0 < k$. This contradicts Theorem 2.

□

Lemma 8. *Let D_i and D_j, $j \neq i$, be any two ears in an ear decomposition D of graph $G = (V, E)$. For an edge $e = (u, v) \in E(G)$ such that $u \in V(\widetilde{D}_i)$, $v \in V(\widetilde{D}_j)$ and $i \neq j$, if $v'_j \in V'$ belongs to a block whose smallest index is smaller than i, then there exists a path P connecting v to the root r in G without passing through any internal vertex $z \in D_i$.*

Proof.

Since vertex $v \in V(\widetilde{D}_j)$, this lemma is equivalent to proving that there exists a path P between v'_j and the root r without passing through v'_i. Let us assume that v'_j belongs to the block B_k and v'_i belongs to the block B_h. Two cases are to be considered separately.

(a) If $k = h$, let v'_p has the smallest index value among all the vertices in $B_k (= B_h)$. By Theorem 2, if there exists a path from v'_j to the root r, it must pass through v'_p. Since v'_i, v'_j, v'_p belong to the same block, they lie on a cycle. By Lemma 1, there exists a path P_1 connecting v'_j and v'_p without passing through v'_i. By Lemma 7, there is a path P_2 from v'_p to root r without using v'_i. It follows that paths P_1 and P_2 form a path connecting v'_j with r in G' without passing through v'_i. Consequently, there is a path P connecting v to the root r in G without passing through any internal vertex $z \in D_i$.

(b) If $k \neq h$, then v'_j and v'_i belong to two different blocks. Let us assume that v'_w has the smallest index in block B_k. Since $w < i$, by Lemma 7, there exists a path P_1 connecting the root r with vertex v'_w bypassing v'_i. Moreover, as v'_j lies in the block B_k, there exists a path P_2 connecting v'_w and v'_j without passing through v'_i. It follows that paths P_1 and P_2 form a walk that contains a path connecting v'_j with r without passing through v'_i. Hence, there exists a path P connecting v to the root r in G without passing through any vertex $z \in D_i$.

$\square$

Let B_i, $1 \leq i \leq l$, be a block in G' and v'_k be the vertex in block B_i that has the smallest index. Let $l_{B_i} = \min\{h \mid \exists e = (u_h, w_q), u_h \in V(D_k) \text{ and } w_q \in V(D_j), i \neq j\}$ and $r_{B_i} = \max\{h \mid \exists e = (u_h, w_q), u_h \in V(D_k) \text{ and } w_q \in V(D_j), i \neq j\}$.

Then, $\forall z \in V$, let z be on an ear D_w such that v'_w and v'_k belong to the same block. If z is directly connected to some vertex in D_k, then let $\beta_z[1..3]$ such that $\beta_z[1] = k$, $\beta_z[2] = l_k$, $\beta_z[3] = r_k$. Otherwise, $\beta_z[1] = k$, $\beta_z[2] = $ null , $\beta_z[3] = $ null .

```
Input: A bridgeless graph $G = (V,E)$, an ear decomposition $D$, $G' = (V',E')$ of $G$
Output: $\beta_z[1..3]$, $\forall z \in V(D_w)$, $0 < w \leq n$
Determine the blocks, $B_1, B_2, \ldots, B_l$, of $G'$.
for $B_i$, $1 \leq i \leq l$ do in parallel
    Compute $v'_k$ such that $k = \min\{j \mid v'_j$ is a cut-vertex of $G$ and $v'_j \in V(B_i)\}$.
    Compute $Q = \{e' \in E(B_i) \mid v'_k$ is an end-vertex of $e'\}$.
    Compute $min_{B_i} = \min\{\lambda_{e'}[1] \mid e' \in Q\}$.
    Compute $max_{B_i} = \max\{\lambda_{e'}[1] \mid e' \in Q\}$.
    Compute $P = \{\lambda_{e'}[2] \mid e' \in Q\}$.
    for $(v'_w \in V(B_i) - \{v'_k\})$ do in parallel
        for $(z \in V(D_w))$ do in parallel
            if $z \in P$ then
                $\beta_z[1] = k$.
                $\beta_z[2] = min_{B_i}$.
                $\beta_z[3] = max_{B_i}$.
            end
            else
                $\beta_z[1] = k$.
                $\beta_z[2] =$ null.
                $\beta_z[3] =$ null.
            end
        Endpar
    Endpar
Endpar
```

Algorithm 2: Finding β value

Lemma 9. *Algorithm 2 correctly computes* $\beta_z[1..3], z \in V$.

Proof. Immediate from the definition of $\beta_z[1..3], z \in V$. □

Let D_i be a non-trivial ear $v_0 e_0 v_1 e_1 \ldots e_{k-1} v_k$ in an ear decomposition D of graph G. For each vertex $v \in D_i$, $0 < i \leq n$, after computing β_v, we then build a graph, Ω_i, for ear D_i to help us find the cut-pairs of the given graph G. The vertex set of Ω_i consists of the vertices of ear D_i. For an internal vertex v on the ear D_i, if v can reach the root r without using any internal vertex in the D_i, then $\tau_v =$ true.

For a vertex w on the ear D_i, if w is one of the end-vertices of an edge $e \in E(G)$ such as $e = (w, z)$, where $z \in D_j$ and $j \neq i$, then an edge $e_1 = (w, v_k) \in E(\Omega_i)$, where v_k is the vertex with the smallest *pos* value on the ear D_i that can be reached from w without using any internal vertex in the ear D_i. If $v_k = w$, then no edge is created in $E(\Omega_i)$. By symmetry, an edge $e_2 = (w, v_h) \in E(\Omega_i)$, where v_h is the vertex with the largest *pos* value on the ear D_i that can be reached from w without using any internal vertex in the ear D_i. An ***Ω-graph*** of G is the union of the graphs Ω_i, $1 \leq i \leq n$.

Input: A graph G, its associated graph G' and an ear decomposition D

Output: Ω-graph for G

```
for each ear $D_i \in D$ do in parallel
    for $v_h \in V(\widetilde{D}_i)$ do in parallel
        $\tau_{v_h}$ = false.
        for $e = (u, v_h)$, $u \in V(D_j), e \notin E(D_i)$ do in parallel
            if $\beta_u[1] < i$ then
                $\tau_{v_h}$ = true.
            end
        Endpar
        if $\tau_{v_h} = false$ then
            $a_{v_h} = \min\{\beta_u[2] \mid \exists e = (u, v_h), u \in V(D_j), i \neq j\}$.
            $b_{v_h} = \max\{\beta_u[2] \mid \exists e = (u, v_h), u \in V(D_j), i \neq j\}$.
        end
        $c_{v_h} = \min\{pos(u, D_i) \mid e = (u, v_h), u \in V(D_i), e \notin E(D_i)\}$.
        $d_{v_h} = \max\{pos(u, D_i) \mid e = (u, v_h), u \in V(D_i), e \notin E(D_i)\}$.
    Endpar
Endpar

if $h \neq \min\{a_{v_h}, c_{v_h}\}$ then
    add an edge $e = (v_h, w)$ such that $pos(w, D_i) = \min\{a_{v_h}, c_{v_h}\}$.
end
if $h \neq \max\{b_{v_h}, d_{v_h}\}$ then
    add an edge $e = (v_h, w)$ such that $pos(w, D_i) = \max\{b_{v_h}, d_{v_h}\}$.
end
```

Algorithm 3: Build Ω-graph for G

Lemma 10. *Algorithm 3 correctly builds the Ω-graph for the graph G.*

Proof.

Since Ω-graph is the union of Ω_i graph for each ear $D_i, 0 \leq i \leq n$, so this statement is equivalent to proving that Algorithm 3 correctly builds the Ω_i graph for each ear D_i.

- **(i)**: Since $v_h \in V(\widetilde{D_i})$, if $\beta_u[1] < i$, it means that v'_i, the image of D_i, belongs to a block whose vertex with the smallest index is smaller than i. Then, by Lemma 8, there exists a path P from u to the root r without using any vertex of ear D_i. By definition, τ_u is correctly set to *true*.

  [**Remark:** Notice that for any edge $e = (u, v_h)$, $u \in V(\widetilde{D_j}), v_h \in V(\widetilde{D_i}), j \neq i$, it is not possible that $\beta_u[1] > i$. Suppose to the contrary, $\beta_u[1] > i$, then by the definition of the β value, v'_i and v'_j cannot be in the same block. Let us assume that $v'_j \in V(B_w)$ and $v'_i \in V(B_t)$, $w \neq t$, where B_w, B_t are blocks of G'. Two separate cases are to be considered:

  Case 1: If vertex v'_j is not the cut-vertex of block B_w, then by Theorem 2, edge $e = (u, v_h)$ must contain a vertex of ear D_l, where v'_l is the cut-vertex of block B_w. Since $u \in V(D_j)$, $j \neq l$, that means $v_h \in V(D_l)$. Furthermore, $v_h \in V(D_i)$ by assumption. So, v_h belongs to both D_i and D_l. But v'_i and v'_l are in two different blocks. We thus have $i = l$. As a result, $\beta_u[1] = i$ which contradicts our assumption that $\beta_u[1] > i$.

  Case 2: If vertex v'_j is the cut-vertex of block B_w, since $e = (u, v_h), v_h \in D_i$ and $v'_i \in V(B_t)$, that means $\beta_u[1]$ is the index of the cut-vertex of B_t. As a result, $\beta_u[1] \leq i$ which contradicts the assumption that $\beta_u[1] > i$.]

- **(ii)**: For edge $e \in E(G)$ such that $e = (u, v_h), u \in V(D_j), v_h \in V(D_i), j \neq i$, if $\beta_u[1] = i$, it means that v'_i and v'_j are in the same block B_k, $1 \leq k \leq l$, where v'_i is the cut-vertex of B_i. Owing to the fact that the cut-vertex v'_i can be a vertex in several blocks, and for each such block B_h, there corresponds two values l_{B_h} and r_{B_h}, it can be easily seen that the variable a_{v_h} represents the minimum l_{B_h} values, and the variable b_{v_h} represents the maximum r_{B_h} values. Furthermore, from the vertex v_h, there can be an edge $e_1 = (u, v_h)$, where $u \in V(D_i)$. It is easily seen that e_1 does not use any internal vertex of D_i except u and v_h.

The variable c_{v_h} represents the minimum u value and the variable d_{v_h} represents the maximum u value, where $u \in V(D_i)$. It is obvious that the minimum of a_{v_h} and c_{v_h} is the index of the vertex with the smallest *pos* value on D_i which can be reached from v_h. Similarly, the maximum of b_{v_h} and d_{v_h} is the index of the vertex with the largest *pos* value on D_i which can be reached from v_h.

□

The following definitions are similar to those defined in [10]. We refer to the edges in $E(\Omega_i) - E(D_i)$ as ***arcs***. It is easily verified that the arcs form a collection of paths in Ω_i. We shall denote each of the paths by $v_1 a_1 v_2 a_2 \ldots v_{q-1} a_{q-1} v_q$, where a_j is the arc connecting vertices v_j and v_{j+1}.

Define an equivalence relation $\mathcal{R}$ over $V(D_i)$ as follows: $\forall v_a, v_b \in V(D_i)$, $v_a \mathcal{R} v_b$ if there is an arc between them or there exists a pair of arcs (v_a, v_c) and (v_b, v_d) such that $pos(v_a, D_i) < pos(v_b, D_i) < pos(v_c, D_i) < pos(v_d, D_i)$.

Definition: An Ω'_i-graph of an ear D_i is a spanning subgraph of the Ω_i-graph such that $E(\Omega'_i) = E(\Omega_i) - E(D_i)$. An Ω'-graph of a graph G is a spanning subgraph of the Ω-graph such that $E(\Omega') = E(\Omega) - E(G)$.

Input: The Ω'-graph of graph G

Output: The connected components $C_1, C_2, \ldots, C_n$ of the Ω'-graph

```
for $D_i \in D$ do in parallel
    for $e_1 = (v_a, v_c) \in E(\Omega'), pos(v_a) < pos(v_c), v_a, v_c \in V(D_i)$ do in parallel
        Find an edge $e_2 = (v_b, v_d)$, $pos(v_b) < pos(v_d)$, such that
        $(i)\ pos(v_a) < pos(v_b) < pos(v_c)$.
        $(ii)\ pos(v_d) = max\{pos(v') \mid \exists e = (v_b, v'), v' \in V(D_i) \land pos(v') > pos(v_c)\}$.
        Find an edge $e_3 = (v_{b'}, v_{d'})$ , $pos(v_{b'}) < pos(v_{d'})$, such that
        $(i)\ pos(v_a) < pos(v_{d'}) < pos(v_c)$.
        $(ii)\ pos(v_{b'}) = min\{pos(v') \mid \exists e = (v', v_{d'}), v' \in V(D_i) \land pos(v') < pos(v_a)\}$.
        if $e_2$ exists then
            add an edge $(v_b, v_c)$.
        end
        if $e_3$ exists then
            add an edge $(v_a, v_{d'})$.
        end
    Endpar
    for $v \in V(D_i)$, where $\tau_v = true$ do in parallel
        add an edge between $v$ and one end vertex of $D_i$.
    Endpar
Endpar
Find the connected components $C_1, C_2, \ldots, C_n$.
```

Algorithm 4: Finding connected components of Ω'-graph

Lemma 11. *Let $u, v \in V(\Omega_i')$. If there exists a path P connecting u and v in Ω_i', then exists two edge-disjoint paths connecting u and v in G.*

Proof. (Proof by induction on the number of interlaces κ in P)

Base case: when $\kappa = 0$, the path P consists of a sequences of arcs:

$$(u =)v_1 a_1 v_2 a_2 \ldots v_{q-1} a_{q-1} v_q (= v)$$

Since every arc corresponds to a path in G that does not use any edge on D_i, therefore the path P gives rise to a path in G that does not use any edge on D_i. Since $D_i[u, v]$ is a path connecting u and v in G that uses only edge on D_i, we thus have two edge-disjoint paths connecting u and v in G.

Suppose the Lemma holds true for $\kappa = k - 1$.

Let $P : (u =)v_1 e_1 v_2 e_2 \ldots v_{q-1} e_{q-1} v_q (= v)$ have $k(> 1)$ interlaces.

Let e_h be the first edge in P that is not an arc (note: this is where the first interlace in P occurs). Consider $P' : v_1 e_1 v_2 e_2 \ldots v_h e_h v_{h+1}$. Then $e_j = a_j, 1 \leq j < h$. It follows that $v_1 a_1 v_2 a_2 \ldots e_{h-1} v_h$ is a path in G that does not use any edge on D_i. As $D_i[u, v]$ is a path connecting u and v in G that uses only edge on D_i, we thus have two edge-disjoint paths connecting u and v_{h+1} in G which give rise to a cycle containing the edge $e = (v_{h+1}, v_h)$.

Now, consider $P'' : v_{h+1} e_{h+1} v_{h+2} \ldots v_{q-1} e_{q-1} v_q$. Clearly, P'' has $k - 1$ interlaces. By the induction hypothesis, there exists two edge-disjoint paths connecting v_{h+1} with v. These two paths also give rise to a cycle containing the edge $e = (v_{h+1}, v_h)$.

Clearly, the edge e is the only common edge of P' and P''. Therefore, by removing e from both paths and joining them at the vertices v_h and v_{h+1}, we obtain a cycle containing u and v in G. Hence, then exists two edge-disjoint paths connecting u and v in G. □

Theorem 3. *Let vertices $u, v \in V(G)$. u, v are in the same connected component of the Ω'-graph if and only if u, v are in the same 3-edge-connected component of G.*

Proof.

1. **(Only If)**: Suppose u and v belong to the same connected component in the Ω'-graph. Then there is a path P connecting u and v.

   (i) u and v belong to the same Ω'_w-graph:

   If P is a path in the Ω'_w-graph, then by Lemma 11, there are two edge-disjoint paths connecting u and v in G. Since there is yet another path connecting u and v using only edge outside $D_i - D_i[u,v]$, we thus have three edge-disjoint paths connecting u and v in G.

   If P is not a path in the Ω'_w-graph, then there must exist a path P' in the Ω'_w-graph that connects u with one end-vertex a of D_i and another path P'' that connects v with the other end-vertex b of D_i. By Lemma 11, there are two edge-disjoint paths connecting u and a in G and two edge-disjoint paths connecting v and b in G. But a and b are connected by two edge disjoint paths using only edges outside the Ω'_w-graph. Hence there are two edge-disjoint paths connecting u and v in G. However, the path $D_i[u,v]$ is a path connecting u and v. This path and the other two paths are clearly disjoint. Hence, there are three edge-disjoint paths connecting u and v in G.

   (ii) u and v belong to different Ω'_w-graph:

   The path P can be partitioned into a collection of sub-paths so that each sub-path is in some Ω'_j-graph.

   By applying Lemma 11 and a simple induction, it is easily verified that there are three edge-disjoint paths connecting u and v in G.

2. **(If)**: Suppose to the contrary that vertex u and v are in the same 3-edge-connected component of G, but u and v are not in the same connected component in Ω'-graph.

Let $u \in V(D_i)$ and $v \in V(D_j)$.

(i) If $i = j$, without loss of generality, we assume that $pos(i, D_i) < pos(j, D_i)$ and b, c be the two end-vertices of D_i. Let

$$P_1 : v_l a_l v_{l+1} a_{l+1} \dots v_{u-1} a_{u-1} v_u (= u) a_u \dots a_q v_{q+1}$$
$$P_2 : v_t a_t v_{t+1} a_{t+1} \dots v_{u-1} a_{v-1} v_v (= v) a_v \dots a_s v_{s+1}$$

then following cases are to be considered: (1) both u and v do not have paths connecting them to b or c, then there is no path in Ω'_i connecting v_{q+1} and v_t, otherwise, u and v are connected. It follows that (v_{l-1}, v_l) and (v_{q+1}, v_{q+2}) forms a cut-pair for 3-edge-connected component that contains u. Similarly, (v_{t-1}, v_t) and (v_{s+1}, v_{s+2}) forms a cut-pair for 3-edge-connected component that contains v. As a result, there exists a cut-pair which can separate vertices u and v. This contradicts our assumption that u and v are 3-edge-connected. (2) If u or v, but not both, is connecting to b(or c), otherwise, u and v is connected in Ω'-graph, then without loss of generality, let u has a path to one of the end-vertices of D_i, b. It follows that (v_{s+1}, v_{s+2}) forms a cut-pair for 3-edge-connected component that contains v, which contradicts our assumption. In conclusion, u and v are in the same connected component in Ω'-graph

(ii) If $i \neq j$, since u and v are 3-edge-connected, there exists ξ number of edge-disjoint paths between u and v, $\xi \geq 3$. Furthermore, these ξ paths can be partitioned into a collection of sub-paths so that each sub-path is in some Ω'_j-graph. By applying a simple induction, it is easily verified that u and v are in the same connected component in Ω'-graph.

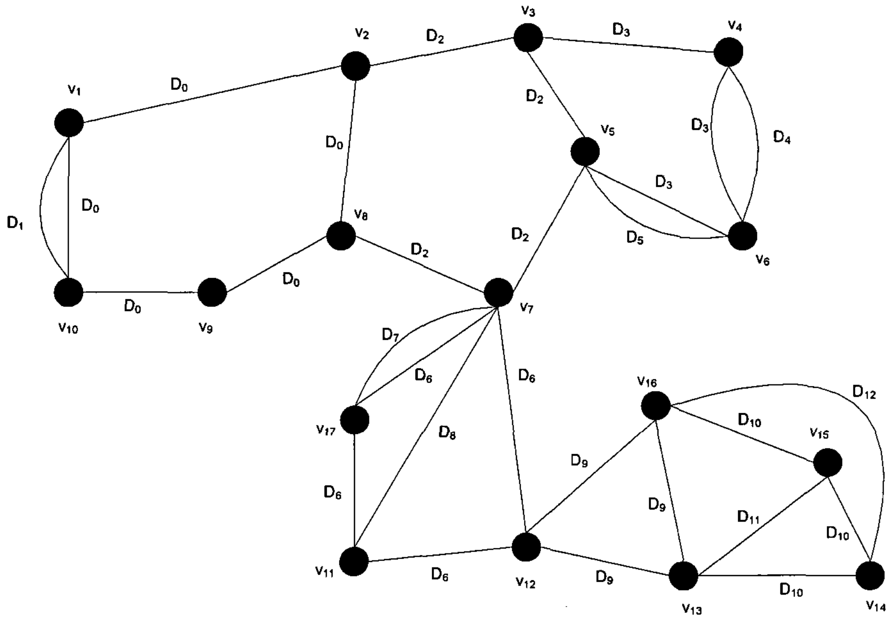
□

```
**Input**: A bridgeless graph $G = (V, E)$
**Output**: The 3-edge-connected components of $G$
Algorithm 1.
Algorithm 2.
Algorithm 3.
Algorithm 4.
```

Algorithm 5: Finding 3-edge-connected components of graph G

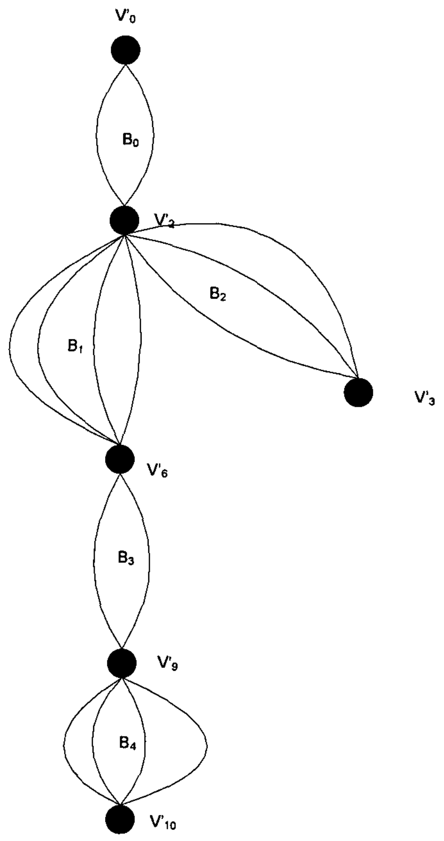
Lemma 12. *Algorithm 5 correctly find the 3-edge-connected components of graph G.*

Proof. Immediate from the correctness of Algorithms 1 to 4. □



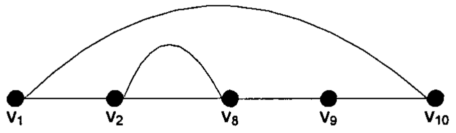
$D_0 = \{ v_1, v_2, v_8, v_9, v_{10} \}$
$D_1 = \{ v_1, v_{10} \}$
$D_2 = \{ v_2, v_3, v_5, v_7, v_8 \}$
$D_3 = \{ v_3, v_4, v_6, v_5 \}$
$D_4 = \{ v_4, v_6 \}$
$D_5 = \{ v_6, v_5 \}$
$D_6 = \{ v_7, v_{17}, v_{11}, v_{12} , v_7 \}$
$D_7 = \{ v_7, v_{17} \}$
$D_8 = \{ v_{11}, v_7 \}$
$D_9 = \{ v_{12}, v_{16}, v_{13}, v_{12} \}$
$D_{10} = \{v_{16}, v_{15}, v_{14}, v_{13}\}$
$D_{11} = \{v_{15}, v_{13}\}$
$D_{12} = \{v_{16}, v_{14}\}$

Figure1: Graph *G* and an open ear decomposition *D*

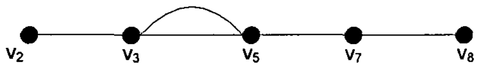


$B_0 : l_{v'0} = 2; r_{v'0} = 8;$ $B_1 : l_{v'2} = 7; r_{v'2} = 7;$
$B_2 : l_{v'2} = 3; r_{v'2} = 5;$ $B_3 : l_{v'6} = 12; r_{v'6} = 12;$
$B_4 : l_{v'9} = 13; r_{v'9} = 16;$

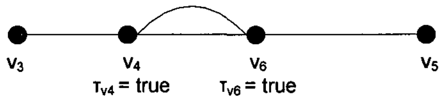
Figure 2: *G'* graph of *G*



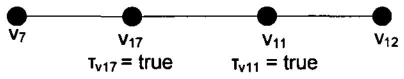
Ω graph for ear D_0



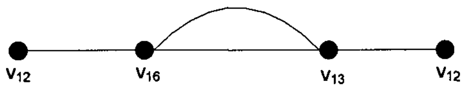
Ω graph for ear D_2



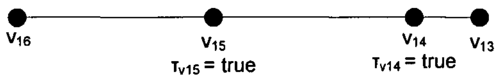
Ω graph for ear D_3



Ω graph for ear D_6

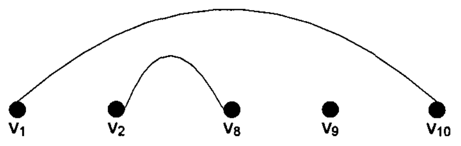


Ω graph for ear D_9

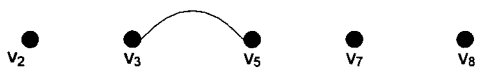


Ω graph for ear D_{10}

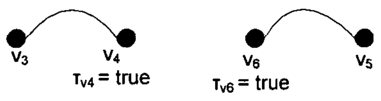
Figure 3: *Ω* for the ears in *D*



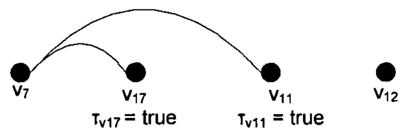
Ω' graph for ear D_0



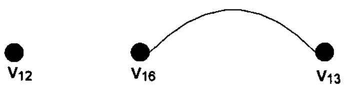
Ω' graph for ear D_2



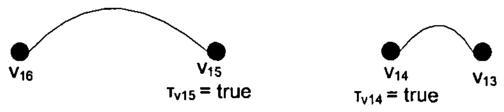
Ω' graph for ear D_3



Ω' graph for ear D_6



Ω' graph for ear D_9



Ω' graph for ear D_{10}

3-edge-connected components:
$\{V_9\},\{V_1,V_{10}\},\{V_2,V_8\},\{V_3,V_4,V_5,V_6\},\{V_{12}\},\{V_7,V_{17},V_{11}\},\{V_{13},V_{14},V_{15},V_{16}\}$

Figure 4: 3-edge-connected components of *G*

Chapter 6

Complexity on PRAM CRCW Model

Fussell et al [9] showed that the time complexity and work for finding triconnected components of a graph G is $O(\log n)$ and $O((m+n)\log\log n)$, respectively. We shall show that finding the 3-edge-connected components can be done within the same bounds.

Definition: ***st-numbering*** [39]: Let $G = (V,E)$ be an undirected graph with $|V| = n$ and $|E| = m$. Let s and t be two distinct vertices of G. Then an st-numbering of G is a numbering of vertices of G by the integers 1 through n such that s is numbered 1, t is numbered n, and any other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex.

Lemma 13. *An st-numbering exists if and only if the graph G is biconnected.*

Proof. See [28] □

Lemma 14. *The Euler-tour technique on trees can be implemented optimally in $O(\log n)$-time with $O(n)$ work on an EREW PRAM.*

Proof. See [40]. □

Lemma 15. *The connected components and a spanning tree of an n-node, m-edge graph can be determined in $O(\log n)$-time with $O((m+n)\log\log n)$ work on an arbitrary-CRCW PRAM provided that the input is presented as an adjacency list.*

Proof. See [5]. □

Lemma 16. *List Ranking on n elements can be performed optimally in $O(\log n)$-time with $O(n)$ work on an EREW PRAM.*

Proof. See [5]. □

The input representation for Algorithm 5 (*Finding 3-edge-connected components of graph G*) is an adjacency list for each individual vertex. The adjacency list is represented by a 1-dimensional array. In this adjacency lists structure of G, for every edge (u,v), vertices u and v appear in each other's adjacency list. The adjacency lists structure can be constructed as follows: given the list of edges, L, of G in which every edge is represented by an unordered pair of the end-vertices of the edge, a parallel bucket sort in the range $[1,\ldots,n]$ is performed over L to produce the desired adjacency lists structure. This can be done in $O(\log n)$- time and $O((n+m)\log\log n)$ work [16].

In executing the parallel algorithm, every vertex u and every edge e is associated with a processor, denoted by P_u and P_e, respectively. In the following, we shall analyze the time and work complexity of Algorithms 1 to 4.

- *Algorithm 1*:

  Finding an ear decomposition D of G can be done in $O(\log n)$-time with $O((m+n)\log\log n)$ work [20]. However, as the cited algorithm does not order the edges on each ear, the following two steps are needed to impose such an order. (1): Decompose the input graph G into biconnected components $G_i, 1 \leq i \leq \omega$. By Lemma 13, we know that an st-numbering must exist for each G_i. By adding $\sum_{j=1}^{i-1} |G_j|$ to the *st*-numbers of the vertices in $G_i, 1 \leq i \leq \omega$, we obtain an *st*-numbering for G. Label each vertex $v \in V(G)$ with this st-number. This can be achieved in $O(\log n)$-time with $O((m+n)\log\log n)$ work [29], (2): Since every ear is a path, every edge (u,v) on an ear can be oriented from u to v. As a result, we can sort the edges on each ear according to the *st*-number of u by a parallel bucket sort. This can be done in $O(\log n)$-time with $O((m+n)\log\log n)$ work [16].

Create an array N of size $n \times 1$ in shared memory, where n is the total number of ears in G. Since n equals to the number of non-tree edges in G, therefore, the value of n is known after the spanning tree of G is constructed. By Lemma 15, this can be done in $O(\log n)$-time. $N[i]$, $0 \leq i \leq z$, represents ear D_i in G. After that, do the following for each ear. For an ear D_i, let us assume the number of edges in D_i is k. Allocate in memory one array M_i of size $k+1$ and create a pointer from $N[i]$ to the first element of M_i. The processor associate with the first edge of ear D_i is responsible for inserting the vertices in $M_i[0]$ and $M_i[1]$. The processors associate with the rest of edges do the following: Let us assume one of the edge is $e = (u,v)$, where pre-order of u is smaller than v, and e is in nth position in the edge list of ear D_i after sorting. Then processor P_e is responsible for inserting v into $M_i[n+1]$. This can be done in $O(1)$-time. After the above setup, the *pos* value can be easily calculated by reading its index number in array M_i associated with ear D_i.

Building G' can be constructed as follows: let the processor assigned to the first edge of ear D_i does the following: (1) Find end-vertices u, v of ear D_i. This can be easily done by examining array M_i which takes $O(1)$-time. (2) Find the ears D_p

and D_q where u and v are the internal vertices in P and Q, respectively. This can be accomplished by examining the adjacency list of u. Since each vertex in this list has been assigned an ear number for the corresponding edge and the edge with the smallest ear number is one that lies on ear D_p, determining this smallest ear number, hence the ear number p, can be done in $O(\log n)$-time. Furthermore, as the st-number of vertex u is known and the vertices in ear D_p are ordered by their st-numbers, finding $pos(u,P)$ can be done in $O(\log n)$-time with binary search. The ear D_q and $pos(v,Q)$ can be determined similarly.

Finally, the edges $e' = (v'_i, v'_p)$, $e'' = (v'_i, v'_q)$ are created while the following assignments are carried out: $\lambda_{e'}[1] \leftarrow p$, $\lambda_{e'}[2] \leftarrow i$, $\lambda_{e''}[1] \leftarrow i$ and $\lambda_{e''}[2] \leftarrow q$.

- *Algorithm 2*

  Using the list of the edges of G' created by Algorithm 1, the first step of finding the blocks in G' can be done in $O(\log n)$-time and $O((m+n)\log\log n)$ work [40]. Then for each block B_i, $1 \leq i \leq l$, the vertex v'_k with the smallest index in B_i is determined. This can be done in $O(\log n)$-time and $O(m)$ work. A spanning tree T_i of B_i is then constructed. By Lemma 15, this can be done in $O(\log n)$-time with $O((m+n)\log\log n)$ work. By applying the Euler-tour technique and using the smallest index k, the spanning tree can be rooted at v'_k. This can be done in $O(\log n)$-time with $O(m)$ work. The set Q consists of the vertices adjacent to v'_k in T_i. Computing min_{B_i} and max_{B_i} thus takes $O(\log n)$-time with $O(n)$ work. The set P can be determined similar to Q. Finally, the values of $\beta_z[i], 1 \leq i \leq 3 \forall z \in V(D_w)$ can be calculated in $O(\log n)$-time with $O(m)$ work.

- *Algorithm 3*

In determining the τ_{v_h} values, the initialization step takes $O(\log n)$-time with $O(n)$ work; determining if $\beta_u[1] < i$ can also be done within the same time and work bounds. If $\exists u$, such that $\beta_u[1] < i$, then $\tau_{v_h} \leftarrow true$. Since there can be more than one such u, there can be more than one processor writing into τ_{v_h}. A write conflict thus occurs. However, as the model allows concurrent-writes and all the processors involved are writing the same value (i.e. *true*), one of them will succeed. This step also takes $O(\log n)$-time with $O(n)$ work. Computing a_{v_h}, b_{v_h}, c_{v_h} and d_{v_h} involves computing the minimum and maximum value of the labels of the edges incident on vertex v_h, where $v_h \in \widetilde{D}_i$. We can apply list ranking [5] to the adjacency list of v_h. Briefly, we compare the β values of each vertex in the adjacency list of v_h in the recovery stage of the list ranking algorithm. By Lemma 16, the four values can be computed in $O(\log n)$-time using $O(n)$ work. Finally, adding the edges e can be easily done in $O(\log n)$-time with $O(n)$ work.

- *Algorithm 4*

  Determining e_2 and e_3 can be reduced to the range-minima problem which can be solved in $O(\log n)$-time with $O(n)$ work [5]. If range-minima returns a position number, then the edge e_2 (e_3, respectively) is added. Adding edges for the cases where $\tau_v = true$ can be trivially done in $O(\log n)$-time with $O(n)$ work. Finally, By Lemma 15, determining the connected component can be done in $O(\log n)$-time with $O((m+n)\log\log n)$ work.

In conclusion, Algorithm 5 (*Finding 3-edge-connected components of graph G*) runs in $O(\log n)$-time on an Arbitrary-CRCW PRAM while performing $O((m+n)\log\log n)$ work.

Chapter 7

Conclusions and Future work

We have presented a parallel algorithm for finding the 3-edge-connected components of an undirected graph. The time and work bounds (hence processor bound) of the algorithm match those of the best-known algorithm for 3-vertex-connectivity on the same computer model.

Our algorithm consists of a number of steps each of which solves a particular subproblem. With the exception of the step for finding connected components and the step for generating an adjacency lists structure, each step can be done in the *optimal* $O(\log n)$-time and $O(m+n)$ work. Actually, for the step for finding connected components, an "*almost optimal*" algorithm is known [5]. This algorithm takes $O(\log n)$-time and does $O((m+n)\alpha(m,n))$ work, where α is the inverse Ackermann function which grows slightly faster than a constant function. Unfortunately, for the step for generating an adjacency lists structure, the best known algorithm relies on integer sort. So far, there does not exist a parallel integer sorting algorithm that has the "almost optimal" time and work bounds.

Hence, to improve the work complexity of our algorithm to the optimal $O(m+n)$ bound, we shall try to improve the work complexity for the problem of finding connected components and for sorting integers.

Bibliography

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

[2] P. Chaudhuri. A note on self-stabilizing articulation point detection. *Journal of System Architecture*, 45(14):1249–1252, 1999.

[3] P. Chaudhuri. A self-stabilizing algorithm for detecting fundamental cycles in a graph. *J. Comput. Syst. Sci.*, 59(1):84–93, 1999.

[4] F. Y. Chin, J. Lam, and I. N. Chen. Efficient parallel algorithms for some graph problems. *Comm. ACM*, 25(9):659–665, 1982.

[5] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 478–491, Washington, DC, USA, 1986.

[6] R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Inform. and Comput.*, 92(1):1–47, 1991.

[7] S. Devismes. A silent self-stabilizing for finding cut-nodes and bridges. *Parallel Processing Letters*, 15(1-2):183–198, 2005.

[8] E.W. Dijkstra. Self-stabilizing algorithm in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[9] D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, 22(3):587–616, 1993.

[10] D. Fussell and R. Thurimella. Separation pair detection. In *VLSI Algorithms and Architectures (Corfu, 1988)*, volume 319 of *Lecture Notes in Comput. Sci.*, pages 149–159. Springer, New York, 1988.

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

[12] Z. Galil and G.F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991.

[13] L.M. Gary and V. Ramachandran. A new graphy triconnectivity algorithm and its parallelization. In *STOC '87: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 335–344, New York, NY, USA, 1987. ACM.

[14] L.M. Gary and J.H. Reif. Parallel tree contraction and its application. In *SFCS '85: Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Washington, DC, USA, 1985. IEEE Computer Society.

[15] C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, pages 77–90, London, UK, 2001. Springer-Verlag.

[16] T. Hagerup. Towards optimal parallel bucket sorting. *Inform. and Comput.*, 75(1):39–51, 1987.

[17] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *STOC '76: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 55–57, New York, NY, USA, 1976. ACM.

[18] W. Hohberg. How to find biconnected components in distributed networks. *J. Parallel Distrib. Comput.*, 9:374–386, 1990.

[19] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.

[20] J. Ja'Ja'. *An Introduction to Parallel Algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[21] J. Ja'Ja' and J. Simon. Parallel algorithms in graph theory: planarity testing. *SIAM J. Comput.*, 11(2):314–328, 1982.

[22] E. Jennings and L. Motyckova. Distributed computation and maintenance of 3-edge-connected components during edge insertions. In *Proceedings of 3rd Colloquium SIROCCO96*, pages 224–240, June 1996.

[23] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivty. *J. of Comput. and System Sci.*, 42:288–306, 1991.

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

[25] M. Karaata. A stabilizing algorithm for finding biconnected components. *J. Parallel Distrib. Comput.*, 62(5):982–999, 2002.

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

[27] 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*, 11:110–118, 2000.

[28] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs (Internat. Sympos., Rome, 1966)*, pages 215–232. Gordon and Breach, New York, 1967.

[29] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theor. Comput. Sci.*, 47(3):277–298, 1986.

[30] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math.*, 9(2):163–180, 1992.

[31] N. Nakanishi. *Graph Theory and Feynman Integrals.* Gordon and Bridge Science Publishers, 1971.

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

[33] B. Paten, M. Diekhans, J. Ma, B. Suh, and D. Haussler. Cactus graphs for genome comparisons. In *Proceedings of the 14th International Conference on Research in Computational Molecular Biology (RECOMB)*, Lisbon, Portugal. (to appear), 2010.

[34] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time (extended abstract). In *VLSI Algorithms and Architectures (Corfu, 1988)*, volume 319 of *Lecture Notes in Comput. Sci.*, pages 33–42. Springer, New York, 1988.

[35] C. Savage and J. Ja'Ja'. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput.*, 10(4):682–691, 1981.

[36] B. Swaminathan and K. J. Goldman. An incremental distributed algorithm for computing biconnected components in dynamic graphs. *Algorithmica*, 22:305–329, 1998.

[37] S. Taoka, T. Watanabe, and K. Onaga. A linear-time algorithm for computing all 3-edge-connected components of a multigraph. *IEICE Transactions on Information and Systems*, E75(3):410–424, 1992.

[38] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[39] R.E. Tarjan. Two streamlined depth-first search algorithms. *Fund. Inform.*, 9(1):85–94, 1986.

[40] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[41] Y. H. Tsin. An efficient distributed algorithm for 3-edge-connectivity. *International J. of Foundations of Computer Science*, 17(3):677–701, 2006.

[42] Y. H. Tsin. An improved self-stabilizing algorithm for biconnectivity and bridge-connectivity. *Inf. Process. Lett.*, 102(1):27–34, 2007.

[43] Y. H. Tsin. A simple 3-edge-connected component algorithm. *Theory Comput. Syst.*, 40(2):125–142, 2007.

[44] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. of Discrete Algorithms*, 7(1):130–146, 2009.

[45] Y. H. Tsin and F. Y. Chin. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Comput.*, 13(3):580–599, 1984.

[46] V. Turau. Computing bridges, articulations and 2-connected components in wireless sensor networks. In *Algorithmic Aspects of Wireless Sensor Networks, Second International Workshop ALGOSENSORS 2006, LNCS 4240*, pages 164–175, 1989.

[47] H. Whitney. Non-separable and planar graphs. *Trans. Amer. Math. Soc.*, 34(2):339–362, 1932.

Vita Auctoris

Yuan Ru was born in 1980 in Wuxi, Jiangsu, China. He graduated from Wuxi No.1 Middle School in 1999. In 2001 he went on to the University of Windsor, in Ontario, Canada, where he obtained a B.Sc. degree in Computer Science in 2004. In 2005, he joined TP Software Ltd in Shanghai as a Java Software Engineer. He is currently a candidate for the Master of Science degree in Computer Science at the University of Windsor and plans to graduate by May 2010.