University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

1-1-2007

Improved I/O-efficient algorithms for solving graph connectivity, biconnectivity problems.

Shan Li
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
Li, Shan, "Improved I/O-efficient algorithms for solving graph connectivity, biconnectivity problems." (2007). *Electronic Theses and Dissertations*. 6994.
https://scholar.uwindsor.ca/etd/6994

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Improved I/O-Efficient Algorithms for Solving Graph Connectivity, Biconnectivity Problems

by

Shan Li

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
2007

©2007 Shan Li

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-35019-5
Our file Notre référence
ISBN: 978-0-494-35019-5

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Many large-scale applications involve data sets that are too massive to fit into the main memory. As a result, some of the data sets must be stored in external memory. Algorithms manipulating these data sets must transfer data between the internal and external memory using I/O (Input/Output) operations. Consequently, a computational model, called the external memory model, have thus been proposed for these applications. The efficiency of an algorithm in the model is measured in terms of the number of I/O operations performed.

In this thesis, we present I/O-efficient algorithms for solving the graph connectivity and biconnectivity problems. Previously best-known external-memory algorithms for the problems are based on simulation of their corresponding Parallel RAM algorithms. By contrast, our algorithms are based on depth-first search and Tarjan's sequential biconnected-component algorithm. All of our algorithms require $O(\lceil |V|/M \rceil \, \text{scan}(|E|) + |V|)$ I/Os, where V is the vertex set, E is the edge set and $|V|$ ($|E|$, respectively) denotes the cardinality of V (E, respectively) in G, M is the size of the internal memory (main memory). For the cases in which $\lceil |V|/M \rceil = \Theta(1)$ (i.e. the vertex set size is a constant factor larger than the main memory size) and $|E| > DB|V|$ (which includes dense graphs as special cases), where D is the number of disk drives and B is the block size, our external memory algorithms require only $O(\text{scan}(E))$ I/Os, whereas the previously best-known external-memory algorithms require the less efficient $O(\text{sort}(E))$ I/Os. Our algorithms are also much simpler.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Dedication

To my parents
Who contributed to enlightening the way of my leading
with endless love...

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

I would like to express my deepest appreciation to all those who have helped me to complete this research work.

I am greatly indebted to my supervisor Prof. Dr. Yung H. Tsin from the School of Computer Science, who has taught me how to do research and has given me invaluable suggestions and stimulating encouragement in all the time of research and writing of this thesis. This work could not have been completed if it were not for the constant assistance and the professional guidance provided by Prof. Dr. Yung H. Tsin.

I would like to express my great gratitude to Prof. Dr. Tim Traynor, Department of Mathematics and Statistics and Prof. Dr. Richard A. Frost, School of Computer Science for giving me corrections and constructive criticism to improve the quality of the thesis and for being in the committee, and to Prof. Dr. Xiaobu Yuan for serving as the chair of the defense.

My colleagues and all the faculty members and staff of the School of Computer Science have been extremely hospitable in providing their suggestions and their support during the mammoth research work. In particular, Mr. Aniss Zakaria has friendily supplied me with technical support and Ms. Mandy Dumouchelle has given me helpful hands in many day-to-day matters. Many thanks go to Ms. Lihua Duan and Ms. Lin Lan for their help in the successful completion of this work.

There are a few people who have reviewed the thesis from outside the School of Computer Science. Especially, I would like to give my thanks to the staff members of the Academic Writing Center for proof-reading the thesis.

Furthermore, I acknowledge the financial support of my supervisor; Prof. Dr. Yung H. Tsin, in the form of research assistantship through NSERC, the School of Computer Science in the form of graduate assistantship, and the Faculty of Graduate Studies and Research in the form of Tuition Scholarship during the entire period of my study at University of Windsor.

Finally, I would like to give my special thanks to my parents whose sustaining love enables me to complete this work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Abstract 3

Dedication 4

Acknowledgments 5

List of Figures 8

List of Algorithms 9

1 Introduction 10

1.1 Motivation 10

1.2 Existing Algorithms 13

1.3 Contribution 14

1.4 Organization of Thesis 15

2 Background Information 16

2.1 Graph Connectivity 16

2.1.1 Definitions 16

2.1.2 Depth First Search 18

2.1.3 Tarjan's Sequential Biconnectivity Algorithm 20

2.1.4 The PRAM Biconnected Component Algorithm 23

2.2 Model of Computation 27

3 Review of the Current State of the Art 29

3.1 The Existing EM Graph-Connectivity Algorithm 31

3.1.1 A Description of the Algorithm 31

3.2 The Existing EM Biconnectivity Algorithm 33

3.2.1 A Description of the Algorithm 33

3.3 An Existing EM Depth-first Search Algorithm 35

3.3.1 A Detailed Description of the Algorithm 36

3.3.2 Correctness Proof 40

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.3 Time Complexity Analysis 42

4 An External-Memory Algorithm for Graph Connectivity 44

4.1 A Detailed Description of the Algorithm 44

4.2 Correctness Proof 46

4.3 Time Complexity 47

5 An External-Memory Algorithm for Biconnectivity 49

5.1 An EM Algorithm for Detecting Cut-Vertices 49

5.1.1 Input Data Structures 49

5.1.2 Computing LOWPOINT 50

5.1.3 Correctness Proof 55

5.1.4 Time Complexity Analysis 57

5.1.5 Detecting the cut-vertices 58

5.2 An EM Algorithm for Detecting Biconnected Components 59

5.2.1 The Description of EM_BCC 60

5.2.2 Correctness Proof 63

5.2.3 Time Complexity Analysis 64

6 Comparison of Time Complexities 66

7 Conclusions 69

VITA AUCTORIS 75

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

1.1 "The memory hierarchy of a typical uniprocessor system, including registers, level 1 cache, level 2 cache, internal memory, and disks. Below each memory level is the range of size for this level. Each value of B at the top of the figure is the size of block switched between adjacent levels of this hierarchy." [45] p. 211 11

1.2 Platter of a magnetic disk driver [45] p. 214 12

2.1 An example of cut-vertex (articulation point) v_2. 17

2.2 After the cut-vertex v_2 is removed. 17

2.3 A graph on which a depth-first search is to be performed. 19

2.4 A depth-first search tree. The search starts at the vertex b, an arrow denotes a tree edge leading from parent to child; a dash line denotes a back-edge. 20

2.5 The depth-first tree produced by Tarjan's algorithm for the graph of Figure 2.3 21

2.6 An illustration of the graph G' of the relation R_c. Figure (a) presents a graph G, including a spanning tree T shown in solid lines with the remaining nontree edges shown in dashed lines. Figure (b) presents the connected components of G' that are the biconnected components of the graph G shown in (a). [23] p. 231 24

2.7 (a) A spanning tree represented by solid lines and the dashed edges are nontree edges. (b) The connected components of G' correspond to the biconnected components of G (a). The relation R'_c defined by the three conditions. Condition 1: $(e_4, e_1); (e_5, e_2)$; Condition 2: $(e_3, e_4); (e_4, e_5)$; Condition 3: (e_9, e_{10}). [23] p. 235 26

2.8 External Memory Model [46]p. 113 27

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Algorithms

1 Depth First search 19
2 Tarjan's Sequential Biconnected-Component Algorithm 22
3 The PRAM Biconnected Component Algorithm 25
4 The EM_GCC algorithm of Chiang et al. 32
5 The EM_BCC Algorithm of Chiang *et al.* 34
6 Routine Rooted-Tree 35
7 EM Evaluate_Tree 35
8 **Algorithm** EM_DFS of Chiang et al. 38
9 **Routine** Unvisited-vertex: To make an unvisited vertex the current vertex 39
10 **Routine** Compact-array A 40
11 **Algorithm EM_GC** 45
12 **Routine** EM_GCC 45
13 **Routine** Unvisited-vertex 45
14 **Routine** Compact-array A 46
15 Algorithm LOWPOINT 52
16 Encounter an unvisited vertex 53
17 Compact the array A when *InterStruct* is full 54
18 Algorithm EM_BCC 61
19 **Routine Unvisited-vertex:** Encounter an unvisited vertex 62
20 Compact the array A when *InterStruct* is full 62
21 **Routine Generate-BCC:** Generate the vertex set of a biconnected component 63

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

1.1 Motivation

Data sets in large-scale applications are often too massive to fit into the main memory. As a result, traditional RAM algorithms are unsuitable for such large applications owing to the substantial Input/Output cost. This is because in designing a RAM algorithm, one assumes that memory access and CPU operation have uniform cost. Unfortunately, this is not the case in large-scale applications. For economical reasons, the memory system of a modern computer consists of a hierarchal structure with distinct access time for the different levels. The highest level, which consists of external disks, is the slowest but has the largest storage size. The lowest level which consists of the registers in the CPU is the fastest but has the smallest storage size. Figure 1.1 shows the memory architecture. In the figure, $8KB$ is the block transfer size between internal memory and external disks.

For large-scale applications that have to store partial data sets on external disks, communication between the faster internal memory and the slower external memory is often considered to be the major bottleneck of the computation. This is because the internal memory is many order of magnitude faster than the external disks. Figure1.2 shows the characteristics of a disk. External disks consist of platters of disks and a read/write head for locating each platter surface. Each disk stores data sets on concentric circles called tracks. At any time, the read/write head has to mechanically locate the correct track to retrieve/transfer data. The location time from one random track to another is often in the order of 3 to 10 milliseconds, compared to the order of nanosecond (10^{-9} seconds) for accessing internal memory [45].

Furthermore, there is a huge gap between the growth rate of CPU speed and that of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

disk transfer speed. Recently, the gap has been getting larger: developments in technology, such as parallel computing, have increased the CPU speed at an annual growth of 40 to 60 percent while the disk transfer speed has only been improved by 7 to 10 percent annually [37]. Consequently, there has been an urgent need to design algorithms that have minimal I/O costs in large-scale computations.

A theoretical computational model, called the external memory model, had been proposed for designing I/O-efficient algorithms (also called external-memory algorithms or EM algorithms). From the perspective of the design of algorithms, minimizing the I/O costs is equivalent to exploiting maximal data locality in the main memory. At the early stage of the development of external memory algorithms, Aggarwal and Jeffrey [2] proposed a standard two-level external memory model. It consists of one logical disk and an internal (main) memory. Later, an improved computational model with multiple logical disks was introduced, which exploits accessing multiple disks in parallel to maximize data locality. It is called the Parallel Disk Model (PDM) [46].

In the PDM model, an I/O operation transfers a block of DB data units between external disks and main memory, or vice versa. At any time, D disks can be accessed in parallel, and each disk can access B records, which are stored in consecutive locations on

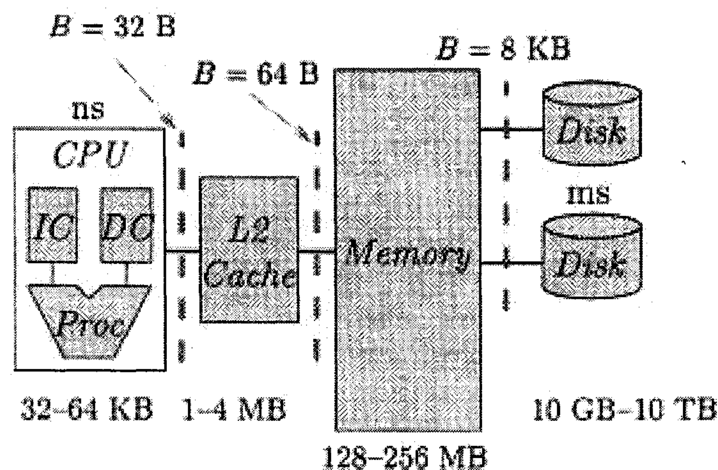


Figure 1.1: "The memory hierarchy of a typical uniprocessor system, including registers, level 1 cache, level 2 cache, internal memory, and disks. Below each memory level is the range of size for this level. Each value of B at the top of the figure is the size of block switched between adjacent levels of this hierarchy." [45] p. 211

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the disk. The performance measure for external-memory (EM) algorithms is the number of Input/Output (I/O) operations performed to transfer data between the two levels of memory without taking internal computation time into consideration. The following parameters are frequently used in analyzing the I/O complexity of EM algorithms [46]:

scan$(N) = \frac{N}{DB}$: The number of I/O operations needed to *read* N data records stored on D disks each with a buffer size B.

sort$(x) = \frac{x}{DB} \log_{\frac{M}{B}} \frac{x}{B}$: The number of I/O operations needed to *sort* N data records stored across D disks each with a buffer size B.

In the area of graph algorithms, substantial effort has been put into solving large-scale graph problems arising in geographic information systems and web modelling where the data volumes are measured in petabytes (10^{15} bytes). For example, data structures that support external-memory graph algorithms in constructing minimum spanning trees, breadth-first search, depth-first search, and finding single-source shortest paths are presented in [29]; several new techniques, such as PRAM simulation and time-forward processing for graph problems, have been developed in [13]; a large number of fundamental graph problems have been solved efficiently [2, 46, 34, 22, 1, 21], and a number of external-memory algorithms have been proposed for planar graphs [5, 6, 7, 8].

Chiang *et al.* [13] have shown that a PRAM algorithm that runs in time T using N processors and $O(N)$ space can be simulated in the EM model using $O(T * \text{sort}(N))$ I/Os. Many external-memory graph algorithms have thus been derived from the corresponding PRAM algorithms. We observed that the simulated PRAM algorithms are limited

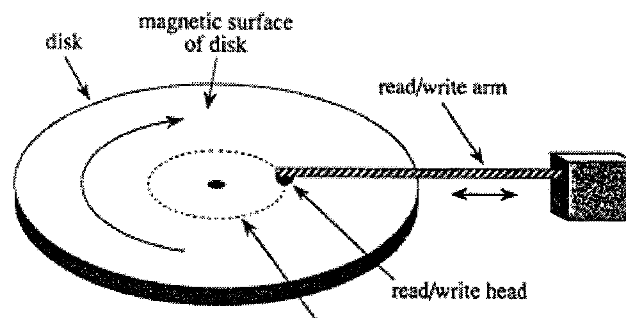


Figure 1.2: Platter of a magnetic disk driver [45] p. 214

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by sequential access to the disks, and are usually very complicated and are difficult to understand and code.

1.2 Existing Algorithms

Graph connectivity, which is one of the fundamental graph-theoretic properties, measures the extent to which a graph is connected. In real-life applications, the property represents the reliability of a telecommunication network or a transportation system. Determining biconnectivity is one of the graph connectivity problems which is related to our work. It is a problem that has been extensively studied on different computational models.

The first biconnectivity algorithm was presented by Tarjan to run on the RAM (the standard sequential computer model). It runs in optimal $O(|V|+|E|)$ time for a connected graph $G = (V, E)$ [39]. Later on, a number of biconnectivity algorithms were developed for the PRAM (the standard parallel computer model). Typically, Tsin and Chin [42] designed an optimal algorithm for dense graphs on the CREW (concurrent-read-exclusive-write) PRAM that runs in $O(\log^2(|V|))$ time using $O(|V|^2/\log^2(|V|))$[1] processors; Tarjan and Vishkin [40] developed an algorithm that takes $O(\log(|V|))$ time with $O(|V| + |E|)$ processors on the CRCW (concurrent-read-concurrent-write) PRAM. 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, 20, 27, 36, 38]. In the fault-tolerance setting, with the assumption that a breadth-first search or depth-first search spanning tree is available, Karaata [25] 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 [15] 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 [41] 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. In wireless sensor network, Turau [43] presented an algorithm that takes $O(|V|)$ time and transmits at most $4m$ messages.

In the EM model, Chiang *et al.* presented the first I/O-efficient biconnected component algorithm. The algorithm is an adaption of the biconnected component PRAM algorithm of Tarjan *et al.* [13] based on simulation. The EM algorithm performs $O(\min\{\mathrm{sort}(V^2),$ $\log(V/M)\,\mathrm{sort}(E)\})$ I/Os. Chiang *et al.* also presented an $O(\log(V/M)\,\mathrm{sort}(E))$ con-

[1] log denotes $\log_2$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

nected component algorithm [13]. This I/O bound was achieved later on by Abello *et al.* [1] using a functional approach. Furthermore, they introduced the *semi-external* model ($|V| < M < |E|$). Vitter observed that several graph problems can be solved optimally on this model. For example, finding connected component, biconnected component can be done in $O(\text{scan}(E))$ I/Os [45]. Subsequently, an I/O complexity, $O((E/V)\,\text{sort}(V) \cdot \max\{1, \log\log(|V|DB/|E|)\})$, for the EM connected component algorithm was reported by Munagala *et al.* [34]. Ulrich Meyer pointed out that the I/O bound of the EM biconnected component algorithm of Chiang *et al.* can be improved to $O((E/V)\,\text{sort}(V) \cdot \max\{1, \log\log(|V|DB/|E|)\})$ using the EM connected component algorithm of Munagala *et al.* [33]. A lower bound of $\Omega(|E|/|V| \cdot \text{sort}(|V|))$I/Os for finding connected components, biconnected components and minimum spanning forests was proved by Munagala and Ranade [34]. Note that $(|E|/|V| \cdot \text{sort}(|V|)) = \Theta(\text{sort}(|E|))$.

1.3 Contribution

In this thesis, we design I/O-efficient algorithms for the external-memory model with parallel disks (PDM [46]). We shall present I/O-efficient algorithms for both the connected component and biconnected component problems. Since detecting the cut-vertices plays a key role in determining the biconnected components, we shall first present an EM algorithm (called EM_CV) for detecting all the cut-vertices. We then present an algorithm (called EM_BCC) to generate all the biconnected components based on the cut-vertices. Our algorithms for solving the biconnectivity problem are adaptations of Tarjan's sequential algorithm. Since the sequential algorithm is developed based on depth-first search, our algorithms are therefore based on the EM depth-first search algorithm of Chiang *et al.* [13].

Our algorithms for biconnectivity are much simpler than the existing best known algorithm of Chiang *et al.* Compared to our algorithm, the algorithm of Chiang *et al.* is complicated, owing to the fact that it is an adaptation of the rather complicated PRAM algorithm of Tarjan *et al.* Our algorithm only needs to construct a *DFS* tree in G. By contrast, the PRAM algorithm uses spanning-tree and Euler-tour techniques to produce an auxiliary graph G' such that every connected component of G' corresponds to a biconnected component of G, and vice versa. The biconnected components of G are then determined by running a PRAM connectivity algorithm on G' (details will be given in Chapter 3).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In terms of I/O complexity, our algorithms make an improvement over the existing algorithm for dense graphs under certain conditions. The algorithm of Chiang *et al.* [13] performs $O(\text{sort}(|V|^2))$ I/Os for dense graphs (i.e. $|E| = O(|V|^2)$) while our algorithm performs $O((\lceil |V|/M \rceil)\,\text{scan}(|E|))$ I/Os. When $\lceil V/M \rceil = \Theta(1)$ (this includes the *semi-external* model as a special case), our algorithm performs $O(\text{scan}(|V|^2))$ I/Os on dense graphs which is better than $O(\text{sort}(|V|^2))$.

1.4 Organization of Thesis

Some background information related to graph theory and the EM computational model are described in Chapter 2. A brief review of the literature of EM algorithms and a description of the EM biconnected component algorithm of Chiang *et al.* are presented in Chapter 3. Since our biconnectivity algorithms need a depth-first search tree of the input graph, a detailed description of the EM DFS algorithm of Chiang et al. [13] is provided in Chapter 3. Chapter 4 explains the proposed algorithms in detail and presents a correctness proof and I/O complexity analysis for each of them. Finally, future work is discussed in the conclusion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Background Information

2.1 Graph Connectivity

The notion of k-vertex-connectivity and k-edge-connectivity are introduced to measure the extent to which a graph is connected. The larger the value of k, the more connected the graph. In telecommunication systems and transportation networks, these properties represent the reliability of the network in the presence of vertex or link failures. A graph is ***k-vertex-connected*** (***k-edge-connected***, respectively) if removing fewer than k vertices (edges, respectively) would not result in a disconnected graph. A ***bi-connected*** graph is a ***2-vertex-connected*** graph. A ***bridge-connected*** graph is a ***2-edge-connected*** graph.

2.1.1 Definitions

We shall denote a graph with $G = (V, E)$, where $V = \{v_1, v_2, ..., v_{|V|}\}$ is the vertex set and $E = \{e_1, e_2, ..., e_{|E|}\}$ is the edge set. Each edge e is associated with a pair of vertices $\{u, v\}$. The vertices u and v are called the ***endpoint***s of edge e which may be identical. Edge e is ***incident to*** vertex u and v.

Definition 2.1.1 ***Directed Graph.*** *A directed graph is a graph in which every edge (called an* **arc***) is associated with an ordered pair of vertices. We shall use $< u, v >$ to represent an ordered pair.*

Definition 2.1.2 *An* **undirected graph** *is a graph in which every edge is associated with an unordered pair of vertices. We shall use (u, v) to represent an unordered pair.*

Definition 2.1.3 ***Path.*** *A sequence of vertices, $v_1, v_2, \ldots, v_k$, is a path if and only if $\{v_i, v_{i+1}\} \in E, 1 \leq i < k$. It is called a $v_1 - v_k$ path. The path is a* **circuit** *if $v_1 = v_k$. The path is* **simple** *if every v_i is distinct; the path is a* **cycle** *if $v_1, v_2, \ldots, v_{k-1}$ is a simple path and $v_1 = v_k$. The vertices v_1 and v_k are called the* **endpoints** *of this path.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.1.4 *A graph $G = (V, E)$ is a **connected** graph if it has a $v - u$ path, $\forall v, u \in V$.*

Definition 2.1.5 ***Bridge (or Cut-edge).*** *In a connected graph $G = (V, E)$, a bridge is an edge $e \in E$ whose removal leads to a disconnected graph.*

Definition 2.1.6 ***Articulation Point (or Cut-vertex).*** *In a connected graph $G = (V, E)$, a vertex $v \in V$ is called an articulation point, or cut-vertex, if its removal leads to a disconnected graph. G is **biconnected** if it has no cut-vertex.*

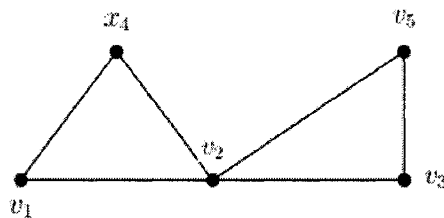


Figure 2.1: An example of cut-vertex (articulation point) v_2.

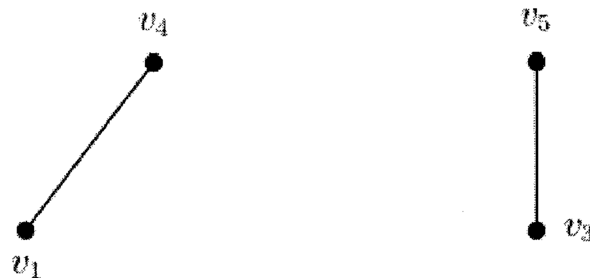


Figure 2.2: After the cut-vertex v_2 is removed.

An articulation point is illustrated in Figure 2.1. Vertex v_2 is an articulation point because its removal results in a graph having two connected components (see Figure 2.2).

Definition 2.1.7 ***Subgraph.*** *A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.*

Definition 2.1.8 ***Biconnected Component.*** *A biconnected component of G, denoted by $G_C = (V_C, E_C)$, is a maximal biconnected subgraph of G.*

Definition 2.1.9 ***Tree.*** *A graph $T = (V, E)$ is a tree if it is a connected circuit-free graph.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.1.10 ***Spanning Tree.*** *A spanning tree* $T = (V_T, E_T)$ *of a graph* $G = (V, E)$ *is a subgraph of* G *such that* $V_T = V, E_T \subseteq E$ *and* T *is a tree. An edge* e *of* G *is a* ***tree edge*** *(w.r.t.* T*) if* $e \in E_T$ *and is a* ***nontree edge*** *(w.r.t.* T*) if* $e \in E - E_T$.

Definition 2.1.11 *Let* u *be a vertex lying on the path connecting the root* r *with a vertex* v *in a tree* T*. Then* u *is called an* ***ancestor*** *of* v *while* v *is a* ***descendant*** *of* u*. If* $u \neq v$*,* u *is called a* ***proper ancestor*** *of* v*, while* v *is a* ***proper descendant*** *of* u*. If* u *and* v *are connected by a tree edge, then* u *is called the* ***parent*** *of* v *and* v *is called a* ***child*** *of* u.

2.1.2 Depth First Search

Depth-first Search (DFS) is a powerful technique for traversing a graph $G = (V, E)$. It generates a spanning tree of G called a ***depth-first search spanning tree*** of G [39] (abbreviated as the *DFS tree*) which shall be denoted by T_{DFS} in this thesis. The search starts from an arbitrary vertex of G, denoted as r, which becomes the root of the *DFS tree.* The search explores the graph deeper and deeper along unvisited vertices until it cannot explore any further, it then retreats (or backtracks) to the most recently visited vertex with an unvisited adjacent vertex and continue exploring the graph from there. The search will terminate when it backtracks to r. In implementing a depth-first search, the list of visited vertices that lie on the path connecting the root and the *current vertex* (to be defined below) in T_{DFS} is maintained on a *stack.*

Definition 2.1.12 *During a depth-first search, the* ***current vertex*** *is the most recently visited vertex that remains active.*

Definition 2.1.13 *An edge is a* ***tree edge*** *if it belongs to* T_{DFS} *and is a* ***back edge*** *otherwise.*

Lemma 2.1.1 *Every back-edge connects a vertex with a proper ancestor or a proper descendant of that vertex.*

Proof: See [39]. □

Algorithm 1 describes an implementation of depth-first search using a stack. Figure 2.4 shows a depth-first tree of the graph shown in Figure 2.3.

In Figure 2.4, the depth-first search starts from vertex b and traverses all the vertices in G in the order: a, f, d, c, h, g, i, e.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 1 Depth First search

Input: the adjacency lists A of a connected graph $G = (V, E)$.
Output: A depth-first search tree of G with vertices being assigned their *DFS number* (the ranks of the vertices in the order they are visited by the depth-first search).
Initialization: $count \leftarrow 1$; $r \leftarrow v$; $\{v$ is arbitrary$\}$
call $DFS(r)$;

Routine $DFS(v)$;
make v as "visited"; $dfs(v) \leftarrow count$; $count \leftarrow count + 1$;
for all vertex w in the adjacency list of v **do**
  if w is not visited **then**
    call $DFS(w)$;
  end if
end for

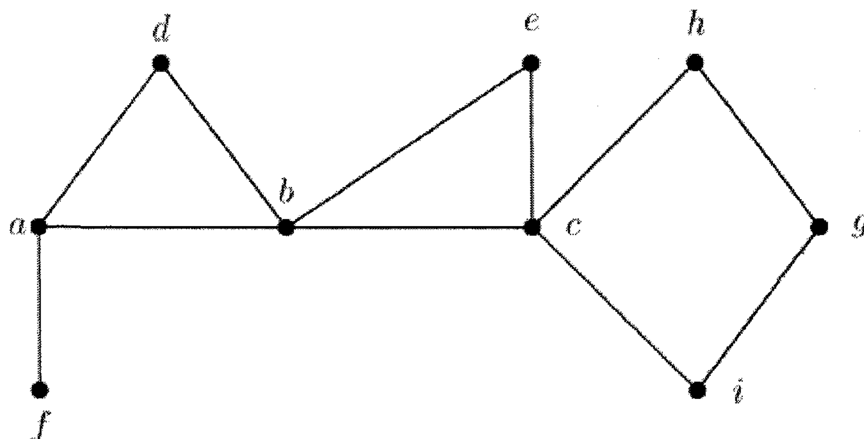


Figure 2.3: A graph on which a depth-first search is to be performed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

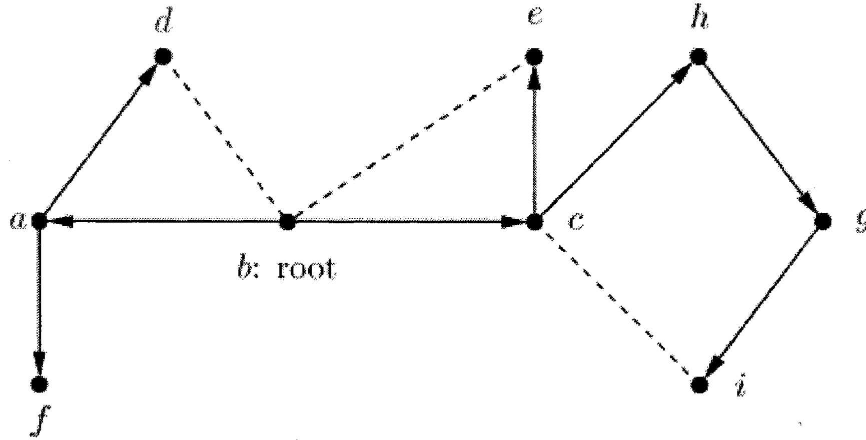


Figure 2.4: A depth-first search tree. The search starts at the vertex b, an arrow denotes a tree edge leading from parent to child; a dash line denotes a back-edge.

2.1.3 Tarjan's Sequential Biconnectivity Algorithm

Tarjan [39] presented a linear-time sequential algorithm for determining biconnected components based on cut-vertices. The algorithm defines and computes the LOWPOINT values to detect cut-vertices, which has become a crucial technique for solving the biconnecitivty problem. We give the following definition used in this algorithm for an arbitrary vertex w of a given graph $G = (V, E)$.

Definition 2.1.14 $\forall w \in V$,

- $low_{(w)} = \min(\{dfs(w)\} \cup \{dfs(u) | \{s, u\} \in B, \textit{for some descendant } s \textit{ of } w\})$, *called the LOWPOINT value of vertex w. [39]*
- $dfs(w)$*: called the* DFS number *of vertex w, which is the* rank *of w in the ordering the vertices are visited by the depth-first search.*
- B*: the set of all back-edges.*
- $C_{(w)}$*: the set of all children of vertex w.*

Lemma 2.1.2 $low_{(w)} = \min(\{dfs(w)\} \cup \{low_{(w')} | w' \textit{ is a child of } w\} \cup \{dfs(u) | (w, u) \textit{is a back-edge}\}), \forall w \in V.$

Proof: See [39]. □

Lemma 2.1.3 *A vertex v is a cut-vertex of G if and only if $v \neq r$ and $\exists w \in C_{(v)}$ such that $low_{(w)} \geq dfs(v)$ or $v = r$ and v has two or more children.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof: See [39]. □

Figure 2.5 shows a depth-first search tree of the graph given in Figure 2.3. Using Tarjan's algorithm, the cut-vertices are identified to be a, b and c. The biconnected components are subgraphs induced by the vertex sets $\{a, f\}, \{a, d, b\}, \{c, h, g, i\}$ and $\{c, b, e\}$. Note that the intersection of each biconnected component and the DFS tree is a subtree of the DFS tree whose root is a cut-vertex.

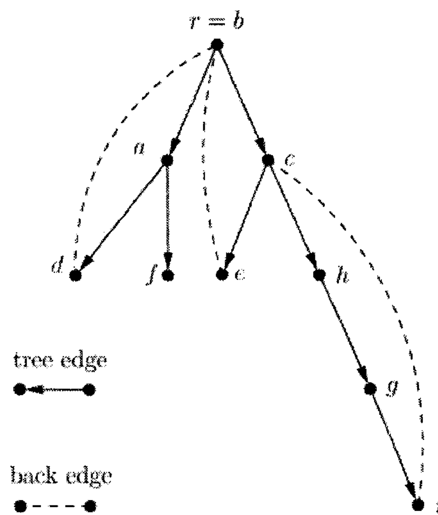


Figure 2.5: The depth-first tree produced by Tarjan's algorithm for the graph of Figure 2.3

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An Overview of Tarjan's sequential algorithm:

Algorithm 2 Tarjan's Sequential Biconnected-Component Algorithm

```
1: Input: The adjacency lists of an undirected graph G = (V, E).
2: Output: The biconnected components of G.
3: Idea: Build a depth-first search tree T of G and compute the LOWPOINT value
   of each vertex; when a cut-vertex is found, output the subtree of T rooted at that
   cut-vertex.
4: Initialization: count ← 1; r ← v; {v is arbitrary}
5: T ← ∅; {T stores the edges of the DFS tree}
6: call DFS(r, ⊥);
7:
   Routine DFS(v, u);
8: make v as "visited"; dfs(v) ← count; count ← count + 1; low_(v) ← dfs(v);
9: for all vertex w in the adjacency list of v do
10:   if edge (v, w) has not been added to T then
11:     T ← T ∪ {(v, w)};
12:   end if
13:   if w is not visited then
14:     call DFS(w, v);
15:     if (low_(w) ≥ dfs_(v)) then
16:       T' ∪ {(v, w)} form the spanning tree of a biconnected component, where T' is
          the subtree of T rooted at w;
17:       Output and delete T' ∪ {(v, w)} from T;
18:     else
19:       low_(v) ← min{low_(v); low_(w)};
20:     end if
21:   else
22:     if (w ≠ u) then
23:       low_(v) ← min{low_(v); dfs(w)};
24:     end if
25:   end if
26: end for
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.4 The PRAM Biconnected Component Algorithm

Since all of the existing EM biconnected component algorithms simulate the corresponding PRAM algorithm, we shall give a brief description of the PRAM algorithm, which is due to Tarjan and Vishkin [23, 40]. The algorithm is designed for the CRCW (concurrent-read-concurrent-write) PRAM model and takes $O(\log(|V|))$ time using $O(|V| + |E|)$ processors. Before discussing the PRAM algorithm,we shall give some definitions related to the algorithm.

Definition 2.1.15 Eulerian Graph. *A graph $G = (V, E)$ is an Eulerian graph if it contains a circuit that traverses every edge of G exactly once. The circuit is an **Euler circuit** or **Euler tour** of G.*

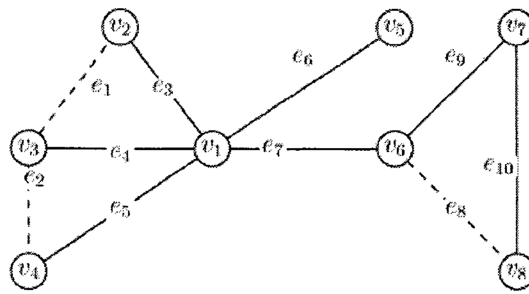
Definition 2.1.16 Transitive Closure. *The **transitive closure** of a directed graph $G = (V, E)$ is the graph $G^* = (V, E^*)$, where E^* consists of all ordered pairs $< i, j >$ such that either $i = j$ or there exists a directed path from i to j.*

Definition 2.1.17 Preorder Number. *The **preorder traversal** of a tree T rooted at r is a sequence of vertices starting with the root r, following by the preorder traversals of the subtrees of r from left to right. The **preorder number**, $pre(v)$, of vertex v is the rank of v in the preorder traversal of T.*

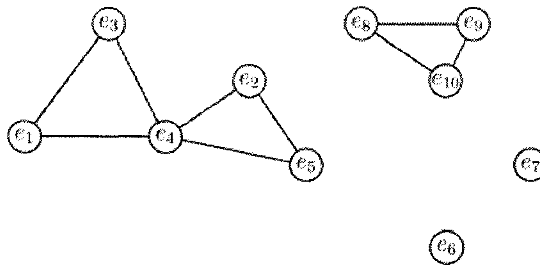
Given an undirected connected graph $G = (V, E)$ and a spanning tree T of G, each nontree edge determines a ***fundamental cycle*** which consists of the edge and the path in T connecting the two end-points of the edge. Let R_c be a relation in the set E defined by eR_cg if and only if e and g belong to a common fundamental cycle. Then the ***transitive closure of*** R_c, denoted by R_c^*, partitions the edge set E of G into a collection of edge sets each of which induces a biconnected component of G. Therefore, the biconnected components of G can be determined as follows. Let $G' = (V', E')$, where $V' = E$ and $(e, g) \in E'$ if and only if eR_cg. Then the connected components of G' correspond to the equivalence classes of R_c^* which uniquely identify the biconnected components of G. The graph G' of the relation R_c is depicted in Figure 2.6. In (a), The set of fundamental cycles consists of $C_1 = e_1, e_3, e_4$, $C_2 = e_2, e_4, e_5$ and $C_3 = e_8, e_9, e_{10}$. The graph G' of the relation R_c is shown in (b). For example, there is an edge between e_4 and e_2 because both e_2 and e_4 belong to the fundamental cycle C_2. The connected components of G' are $\{e_1, e_2, e_3, e_4, e_5\}$, $\{e_8, e_9, e_{10}\}$, $\{e_6\}$ and $\{e_7\}$, which define the biconnected components of G.

Since $|R_c| = \theta(|V|^2)$ is too time-consuming to compute. Tarjan and Vishkin defined a smaller relation R'_c which has the size $O(|E|)$ instead of $(|V|^2)$ and proved that the transitive closure of R'_c is the same as that of R_c. In their PRAM algorithm, each vertex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a)



(b)

Figure 2.6: An illustration of the graph G' of the relation R_c. Figure (a) presents a graph G, including a spanning tree T shown in solid lines with the remaining nontree edges shown in dashed lines. Figure (b) presents the connected components of G' that are the biconnected components of the graph G shown in (a). [23] p. 231

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is identified by its preorder number. For any two edges e and g, $eR'_c g$ if and only if one of the following conditions holds (the parent of a vertex u is denoted by $p(u)$ and the root of T is denoted by r): [23]

1. $e = (u, p(u))$ and $g = (u, v) \in G - T$ and $v < u$.

2. $e = (u, p(u))$ and $g = (v, p(v))$ and $(u, v) \in G - T$ such that u and v are not related (having no ancestral relationship).

3. $e = (u, p(u))$, $g = (v, p(v))$ such that $p(u) = v, v \neq r$, and some nontree edge of G joins a descendant of u to a non-descendant of v.

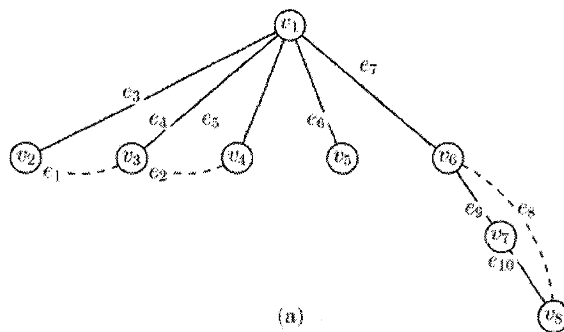
The PRAM algorithm computes R'_c instead of R_c, and for each $v \in V$, let $low(v)$ denote the *smallest* vertex that is either a descendant of v or adjacent to a descendant of v by a nontree edge. Similarly, $high(v)$ denotes the *largest* vertex that is either a descendant of v or adjacent to a descendant of v by a nontree edge. The algorithm is described below.

Algorithm 3 The PRAM Biconnected Component Algorithm

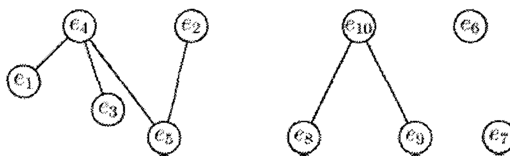
1: **Input**: A connected undirected graph G.
2: **Output**: An array C such that $C(e) = C(g)$ if and only if e and g are in the same biconnected component.
3: Construct a spanning tree T (not necessarily a DFS tree) of the input graph G.
4: Root T at an arbitrary vertex, and apply the Euler-tour technique to assign to each vertex its preorder number.
5: For each vertex v, compute two values $low(v)$ and $high(v)$.
6: Test conditions of R'_c using the low, $high$ values and build the auxiliary graph G'.
7: Find the connected components of G'. These connected components give rise to the biconnected components of G and are identified by an array C.

An illustration of the algorithm is given in Figure 2.7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a)



(b)

Figure 2.7: (a) A spanning tree represented by solid lines and the dashed edges are nontree edges. (b) The connected components of G' correspond to the biconnected components of G (a). The relation R'_c defined by the three conditions. Condition 1: $(e_4, e_1); (e_5, e_2)$; Condition 2: $(e_3, e_4); (e_4, e_5)$; Condition 3: (e_9, e_{10}). [23] p. 235

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Model of Computation

Since retrieving data from the external disks requires a substantial amount of access time, a block, instead of a datum, is transferred between the external disks and the main memory. To simulate the behavior of I/O operations, Aggarwal and Jeffrey [2] proposed a standard two-level I/O system with one logical disk.

The External memory model we shall use is the **Parallel Disk Model (PDM)** [46] (see Figure 2.8.)

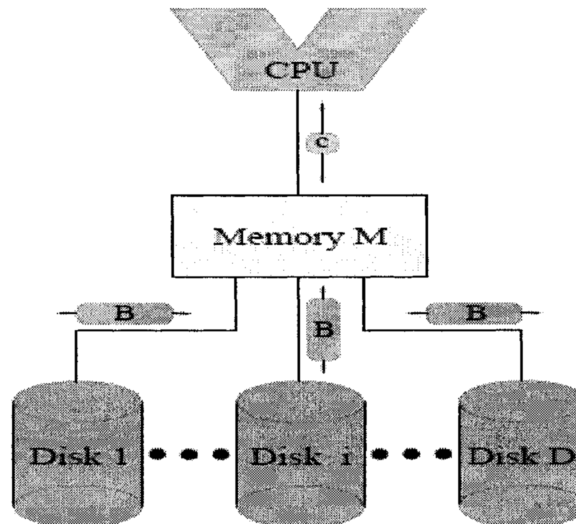


Figure 2.8: External Memory Model [46]p. 113

In this model, the external memory consists of D disks each of which is associated with a read/write head. During each ***I/O operation***, the D disks can simultaneously transfer a block of B contiguous data items. Therefore, the total number of data items transferred in each I/O is DB. The following parameters are associated with the PDM:

- N = input size (in units of data items),
- M = internal memory size (in units of data items),
- B = block transfer size (in units of data items),
- D = number of independent disk drives.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The lengths of the data items are all bounded by the same constant. The input data are stored in the external memory because they are too large to fit into the main memory. In general, $M < N$ and $1 \leq DB \leq \frac{M}{2}$. However, for graph-theoretic problems, let $G = (V, E)$ be the input graph, it is possible that $|V| \leq M < |E|$. In which case, the model is called ***semi-external memory model*** or ***semi-external model*** for short.

Since data items are transferred in blocks of B data items, it is therefore convenient to use the following parameters:

- $n = \frac{N}{B}$,
- $m = \frac{M}{B}$.

The input data are initially *striped* across the D disks in units of blocks. Specifically, the first block of B data items are stored on the first disk, the second block of B data items are stored on the second disk, ..., the Dth block of B data items are stored on the Dth disk, the $(D+1)$th block of B data items are stored on the first disk, the $(D+2)$th block of B data items are stored on the second disk, and so on. In this way, an input of N data items can be read or written with $O(N/DB) = O(n/D)$ I/Os.

The performance measure is expressed in terms of ***I/O complexity***. The I/O complexity of an EM algorithm is the total number of I/O operations it performs.

Sorting and scanning a sequence of consecutive data items are two primitive operations that are frequently used in external-memory algorithms. The I/O complexity for scanning N consecutive data items striped across the disks is:

$$\text{scan}(N) = \frac{N}{DB}$$

The I/O complexity for sorting N consecutive data items striped across the disks is:

$$\text{sort}(N) = \frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B} \text{ [35]}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Review of the Current State of the Art

In this chapter, we review works related to EM graph algorithms done in the last decade. We shall call the sequential algorithms, which were described in the previous chapter, *internal algorithms* as opposed to external algorithms.

The first publication on EM graph algorithms was due to Ullman and Yannakakis [44], in which an external-memory transitive closure algorithm on a directed graph was presented. The algorithm is based on depth-first search traversal and topological sort. It requires $O(dfs(|V|, |E|) + \text{scan}(|V|^2\sqrt{|E|/M}))$ I/Os, where $O(dfs(|V|, |E|)$ is the number of I/Os performed by the depth-first search algorithm. For the *semi-external* model (i.e. $V \leq M \leq E$), Ullman and Yannakakis proposed an EM depth-first search algorithm that performs $O(\text{scan}(|E|) + |V|)$ I/O operations.

Chiang *et al.* [13] presented a number of EM graph algorithms. The algorithms rely heavily on external sorting. As a result, The asymptotic I/O bounds of their algorithms include the parameter $\text{sort}(x)$ as a factor.

Abello *et al.* [1] developed a functional approach for solving graph problems on the external-memory model. They also showed that on the *semi-external* model, their approach could solve the connected component problem in $O(\text{scan}(|E|) \log_{M/B} C(G))$ I/O operations, where $C(G)$ is the number of connected components in G.

A substantial amount of research had been devoted to the development of EM data structures. Buchsbaum *et. al.* [9] presented an EM data structure for developing an op-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

timal EM breadth-first search. Subsequently, an EM breadth-first search algorithm for undirected graphs with bounded degree was presented by Meyer [32]. Later, Kumar and Schwabe [29] designed the EM binary heap data structure and tournament trees; as part of the result, they presented improved graph algorithms with amortized performance, for constructing a minimum spanning tree, breadth-first search and depth-first search, and single-source shortest paths.

Munagala and Ranade [34] presented improved techniques for solving the connectivity problem and undirected BFS problem. Arge [5] extended the technique to compute minimum spanning forest. In Chapter 1, we mentioned EM algorithms for solving the biconnectivity problem. These algorithms are not depth-first search based and hence do not use EM depth-first search although a large number of I/O-efficient depth-first search algorithms had been developed.

Chiang *et al.* [13] made a major contribution to the field by showing that parallel algorithms for the PRAM can be simulated in the external model. Specifically, In each step of the simulation, input data are sorted and divided into independent sets by the indices of the processors on which they are required for computations of this step. These data sets are then read into the main memory by a *scan* operation. After each simulation step, the results are written back to the disks. The results are sorted again for the next simulation step. Each simulation step needs $O(\text{sort}(N))$ I/Os. Therefore, if a PRAM algorithm runs in time T with N processors, the simulation will require $O(T \cdot \text{sort}(N))$ I/Os.

Vitter [45] pointed out that to be simulated on the EM model, a parallel algorithm must have a time complexity of $O((\log N)^c)$ (for some small constant c) when N processors are available.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 The Existing EM Graph-Connectivity Algorithm

3.1.1 A Description of the Algorithm

Chiang *et al.* [13] adapted the parallel connected component algorithm of Chin *et al.* [14] to construct an EM connected component algorithm (or GCC). The input graph $G = (V, E)$ is represented by the adjacency lists of the vertices stored in a 2-dimensional array. The adjacency lists are stored on the external disks.

Each vertex belongs to exactly one connected component. The array D identifies, for each vertex, the connected component containing that vertex. Consequently, $D(i) = D(j)$ if and only if vertices i and j belong to the same connected component.

Let $V = \{1, 2, 3, ..., |V|\}$. Chiang *et al.* use a $O(1)$ number of sorts to sort the edges of G and a list ranking method to reduce the number of vertices during the following vertex reduction step:

The idea is to find, in each iteration, sets of vertices such that the vertices in each set belong to the same connected component. This is accomplished as follows: in each set, a minimal-numbered vertex is selected as the representative of the set, called the super-vertex. All edges in each set are compressed and all vertices in the same set are merged into the super-vertex. As a result, the set is reduced to a *super-vertex*. The process is repeated on the reduced graph until the reduced graph is an edgeless graph. Each single vertex of the edgeless graph represents a connected component of G. A formal description of EM_GCC algorithm is given below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 4 The EM_GCC algorithm of Chiang et al.

1: Initialization: $\forall i \in V, D(i) \leftarrow i$;
2: **repeat**
3: $\forall i \in V, C(i) \leftarrow \min\{j | j \in Adj[i]\}$;
4: **for all** $i \in V$ **do**
5: **if** $C(i) = NULL$ **then**
6: $C(i) = i$
7: **end if**
8: **end for**
9: Label each vertex i with "*isolated*"if and only if $C(i) = i$; {vertex i is isolated}
10: $\forall i \in V$, do $D(i) \leftarrow C(i)$;
11: Apply the list ranking algorithm to do $\forall i \in V, C(i) \leftarrow C(C(i))$;
12: $\forall i \in V, D(i) = \min\{C(i), D(C(i))\}$;
13: $\forall i \in V$, do $D(i) \leftarrow D((D(i))$;
14: Replace each edge (i, j) in E by an edge $(D(i), D(j))$, where $D(k)$ is the super-vertex of the connected component vertex to which k belongs;
15: Remove parallel edges and self-loops and vertices labelled with "*isolated*"
16: **until** $|V| \leq M$

Explanation:

D: An array of length $|V|$. $D(i)$ specifies the super-vertex into which vertex i is merged.

C: An array of length $|V|$. $C(i)$ specifies the smallest-numbered vertex to which vertex i is adjacent.

On Line 1: Every vertex is a super-vertex initially. This step takes scan($|V|$) I/Os.

On Line 3: For every vertex i, the smallest-numbered vertex among all the adjacent vertices is selected and assigned to $C(i)$. In doing so, the adjacency list of i, $Adj[i]$, has to be read into the main memory. Chiang *et al.* showed that this step can be done with $O(\text{sort}(E))$ I/Os.

On Lines 4-9: The isolated vertices are eliminated. Each of these isolated vertices is a super-vertex corresponding to a connected component of G.

On Lines 10-13: Path compression is performed to merge the vertices into super-vertices based on the list-ranking technique. Chiang *et al.* [13] presented a list ranking algorithm which runs on a N-node link list in $O(\text{sort}(N))$ I/Os.

On Lines 14-15: The merged vertices are removed after all information about their adjacent vertices are transferred to the super-vertices. As a result, the size of the graph G is reduced before the next iteration begins. This step requires a constant number of *sorts* and *scans* on the edges.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The total number of I/Os performed in each iteration is sort($|E|$). During each iteration, a vertex reduction step is applied to reduce the number of vertices of G to at most $\lceil V/M \rceil$ vertices. When the number of remaining vertices is less than or equal to M during the above reduction step, the reduced graph can fit into the main memory and the problem can be solved by the internal GCC algorithm. Therefore, $\log(\lceil V/M \rceil)$ iterations are sufficient. Chiang *et al.* showed that Algorithm EM_GCC performs $O(\log(V/M)\,\text{sort}(E))$ I/O operations.

3.2 The Existing EM Biconnectivity Algorithm

3.2.1 A Description of the Algorithm

As was mentioned in Chapter 2, the existing EM Biconnectivity algorithm (or EM_BCC) simulates the PRAM algorithm of Tarjan and Vishkin. Recall that in explaining the PRAM BCC algorithm in Chapter 2, we mentioned that the central idea of the EM BCC algorithm is to transform a given graph G into a graph G' such that the connected components of G' correspond to the biconnected components of G. Each vertex of G' is an edge of G. The edge $(e1, e2)$ exists in G' if and only if $e1$ and $e2$ belong to the same cycle in G (Section 2.1.4).

The simulation involves the following phases: firstly, find an arbitrary spanning tree using Algorithm EM_GCC which can be done with $(\log(|V|/M)(\text{scan}(|E|))$ I/Os; secondly, use the Euler-tour technique and list ranking to compute the preorder number of each vertex, following by a constant number of *sorts* and *scans* on the edges to check if the conditions for R'_c described in 2.1.4 are satisfied. Thirdly, construct G' and then apply Algorithm EM_GCC to it to determine its connected components. Lastly, construct the biconnected components of G from the connected components of G'. Chiang *et al.* [13] showed that Algorithm EM_BCC performs $\min(\log(|V|/M)(\text{scan}(|E|)), \text{sort}(|V|^2))$ I/Os. An overview of the algorithm is given below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 5 The EM_BCC Algorithm of Chiang *et al.*

1: Run Algorithm **EM Generate_SpanningTree** on a connected graph $G = (V, E)$ to construct an arbitrary spanning tree of G, denoted as $T = (V, E_T)$. Algorithm Generate_SpanningTree is a slight modification of Algorithm EM_GCC.
2: Find an Eulerian circuit of $T' = (V, E')$, where T' is produced from T by duplicating every edge of the latter.
3: Use Routine **Root-Tree** to convert T' into a rooted tree, $\hat{T}(r)$, with vertex r (r is arbitrary) being the root.
4: Run Routine **EM Evaluate_Tree** on the tree $\hat{T}(r)$ to compute $low(i)$ and $high(i)$, for each vertex i.
5: Test the conditions of R_c' using the values $low(i)$ and $high(i)$ to label the edges of T. Then, construct an auxiliary graph G'.
6: Apply Algorithm **EM_GCC** to G' to determine its connected components.

Explanation:

On Line 1: In Algorithm EM_GCC, the $C(i)$ pointers induce a collection of trees. When the super-vertices are merged into larger super-vertices, the trees are also merged into larger trees. When execution of Algorithm EM_GCC terminates, the collection of trees are also merged into a spanning tree of G.

On Line 2: Let the spanning tree T of G generated on Line 1 be represented by the adjacency lists Adj_T. An Eulerian circuit of $T' = (V, E')$ can be determined by computing the ***successor*** function s. Specifically, let $Adj_T(v) = < u_0, u_1, ..., u_{d-1} >$, where d is the degree of vertex v. The successor function s is defined as: $s(< u_i, v >) = < v, u_{(i+1) \bmod d} >, 0 \leq i \leq d-1$.

On Line 3: A rooted tree $\hat{T}(r)$ with root r is constructed by applying the Euler-tour technique on T'. Routine Rooted-Tree (presented below) determines the parent of each vertex $v (\neq r)$ in $\hat{T}(r)$.

On Line 4, In the rooted tree $\hat{T}(r)$, each vertex is identified by its preorder number defined in Chapter 2. Routine EM Eavluate_Tree is presented below.

On Line 5: A list L of edges is created to determine the graph G' as follows [23]:

By condition 1 of R_c', for each edge $g = (u, v) \in G - T$ such that $v < u$, put the pair (e, g) in L, where $e = (u, p(u))$.

By condition 2 of R_c', for each edge $(u, v) \in G - T$ such that $v + size(v) \leq u$ ($size(v)$ is the number of vertices in the subtree rooted at v), put the pair (e, g) in L, where $e = (u, p(u))$ and $g = (v, p(v))$.

By condition 3 of R_c', for each edge $e = (u, p(u)), p(u) = v, v \neq r$, put the pair (e, g) into L if $low(u) < v$ or $high(u) \geq v + size(v)$, where $g = (v, p(v))$.

On Line 6: The connected components of G' are determined after it is constructed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 6 Routine Rooted-Tree

1: Let r be an arbitrary vertex (the root);
2: $s(< u, r >) \leftarrow 0$, where vertex u is the last vertex of $Adj_T(r)$;
3: Assign a weight of 1 to each arc $< u, v >$;
4: Apply the EM list ranking algorithm on the list defined by s;
5: **if** $\forall < u, v >, (rank(< u, v >)) < (rank(< v, u >))$ **then**
6: $\quad u = parent(v)$
7: **else**
8: $\quad v = parent(u)$
9: **end if**

Explanation:

On Line 4: The list L defined by s is a collection of arcs $< r, u_1 >, < u_1, u_2 >, ..., < u_{|V|}, r >$ such that each arc $< u_i, u_{i+1} >$, $1 \leq i < |V|$, except of the last one (the *tail*), stores a pointer *next* to its successor in L. The list ranking problem is to compute the distance from the head (the first element) of L to each arc $< u_i, u_{i+1} >$, denoted by $rank(< u_i, u_{i+1} >)$.

Algorithm 7 EM Evaluate_Tree

1: {Compute the preorder number of each vertex v}
2: $\forall v (\neq r) \in V$,
assign the weight $w(< parent(v), v >) = 1$ and $w(< v, parent(v) >) = 0$;
3: Apply the EM list ranking algorithm on the list defined by s;
4: $\forall v \neq r, \in V$, $pre(u) \leftarrow rank(< parent(u), u >)$; $pre(r) \leftarrow 0$;
5: {Compute the *low* values of each vertex}
6: $\forall v \in V$, $w(v) \leftarrow \min(\{v\} \cup \{u|(v, u)$ is a nontree edge$\})$
7: $\forall v \in V$, $low(v) \leftarrow \min\{w(u)|u$ is in the subtree rooted at $u\}$;
8: {Compute the *high* values of each vertex}
9: $\forall v \in V$, $w(v) \leftarrow \max(\{v\} \cup \{u|(v, u)$is a nontree edge$\})$;
10: $\forall v \in V$, $high(v) \leftarrow \max\{w(u)|u$ is in the subtree rooted at $u\}$

Theorem 3.2.1 *Given a graph $G = (V, E)$, the connected components, biconnected components of G can be computed with* $\min\{log(V/M)(scan(E)), sort(V^2)\}$ *I/Os.*

Proof: See [13]. □

3.3 An Existing EM Depth-first Search Algorithm

The original $O(V + E)$-time depth-first-search algorithm of Tarjan [39], is meant for the sequential RAM. As a result, it is not I/O-efficient.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In [13], Chiang *et al.* presented an EM depth-first search algorithm for directed graph, henceforth called EM_DFS, that requires $O((1+V/M)\,\text{scan}(E)+V)$ I/Os. Unfortunately, as the description, the correctness proof, and the time complexity analysis of the algorithm they give are extremely terse, we shall first provide a detailed description of the algorithm in this chapter.

Since our objective is to develop I/O efficient algorithms for *undirected* graphs, we shall thus present our detailed explanation of Algorithm EM DFS in the context of undirected graphs. Note that every undirected graph can be viewed as a directed graph satisfying the condition: "there is a directed edge from vertex i to vertex j if and only if there is a directed edge from vertex j to vertex i."

Lemma 3.3.1 *The EM_DFS algorithm of Chiang et.al. can be executed on an undirected graph.*

Proof: Immediate from the aforementioned condition. □

3.3.1 A Detailed Description of the Algorithm

The algorithm of Chiang *et al.* takes three arrays that are stored on external disks as input. The arrays represent the input graph $G = (V, E)$. One array A with a size of $|E|$ consists of the adjacency lists of the graph. The other two arrays, *start* and *stop*, each with size $|V|$ mark the beginning and the end, respectively, of the adjacency list of each vertex in array A. Specifically, for each vertex i, $\{A[j] | Start[i] \leq j \leq Stop[i]\}$ consists of all the vertices adjacent to vertex i in G. Without loss of generality, we assume $V = \{1, 2, 3, ..., |V|\}$

To execute EM_DFS, the main memory is divided into two parts. One part is the *input buffer* consisting of a block of DB data units. It is used to transfer data between the external memory and the main memory. The other part is used to maintain an internal search structure and to maintain or keep data for booking purposes. For instance, the offset variables for calculating the location of the block of DB data units to be read into the main memory are kept in this part.

The internal search structure is a data structure (such as a hash table or a balanced binary search tree) which is used to hold vertices that have been visited by the search. Its purpose is to avoid performing the expensive I/O operation when an edge connecting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the current vertex of the search with an already visited vertex is being explored.

The algorithm maintains a stack in the external memory, called the *DFS stack*, to store the vertices on the path that connects the root with the current vertex of the search. Initially, the stack is empty. Suppose the depth-first search starts at a vertex, say r. Then vertex r becomes the current vertex of the search. It is then inserted into the internal search structure. A section (a block of DB vertices) of the adjacency list of vertex r starting with $A[start[r]]$ is then read into the input buffer inside the main memory. Let $A[start[r]] = v$. The internal search structure is then searched for v. Since v is clearly unvisited, it cannot be in the search structure. Vertex v therefore becomes the current vertex. So, $start[r]$ is updated to $start[r] + 1$ and vertex r is pushed onto the *DFS stack*. Note that the updated $start[r]$ points at the next vertex in the adjacency list of r to be examined when the search backtracks to vertex r in a later stage. Vertex v is then inserted into the internal search structure. The vertex is then processed in a way same as that for vertex r described above.

In general, let vertex v be the current vertex of the depth-first search and a section of the adjacency list of vertex v starting from $A[start[v]]$ has just been read into the main memory. Let $A[start[v]] = u$. The internal search structure is then searched for the vertex u. If u is found in the search structure, then it is a visited vertex. So, the next vertex, $A[start[v] + 1]$, in the adjacency list is examined. If the vertex is again found in the search structure, then the next vertex in the adjacency list is examined. This is repeated until either an unvisited vertex, u, in the adjacency list is found or the section of adjacency list of v kept in the input buffer is completely examined. In the former case, the vertex u will become the current vertex. So $start[v]$ is updated so that $A[start[v] - 1] = u$, and vertex v is pushed onto the *DFS stack* while vertex u is inserted into the search structure. In the latter case, the next section of the adjacency list of vertex v is read into the input buffer and the aforementioned process is repeated. If there is no next section of the adjacency list of v that has not been examined, then the depth-first search must backtrack to the parent vertex of vertex v which is the top element on the *DFS stack*. Therefore, the stack is popped and the element popped out becomes the current vertex of the depth-first search.

Since $M < |V|$, the internal search structure may overflow. When that happens, The array A is scanned, cleaned up and compacted as follows: for each vertex v, all those vertices in the adjacency list of v that appear in the internal search structure are removed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These vertices represent edges that connect vertex v with visited vertices. Since these edges will be ignored when they are being examined in a latter stage of the search, they can thus be discarded at this point of time. The array A is then compacted so that all the (unvisited) vertices remain in the list appear in a block of consecutive locations in the external memory starting from the location $A[1]$. Furthermore, all the vertices belonging to the same adjacency list are stored in a block of consecutive locations whose beginning and end entries are marked by the updated *start* and *stop* pointers. After the array A is cleaned up, the internal search structure is then emptied and the depth-first search resumes. Note that overflow can happen at the internal search structure at most $\lceil |V|/M \rceil$ times. This is because every vertex can be inserted into the internal search structure at most once and the internal search structure can accommodate $O(M)$ vertices. The internal search structure is needed until the array A is reduced to such a size that it could fit into the main memory.

The following is a formal description of Algorithm EM_DFS.

Algorithm 8 Algorithm EM_DFS of Chiang et al.

1: Input: The arrays $A[1 :: |E|]$, $start[1 :: |V|]$ and $stop[1 :: |V|]$ representing a graph $G = (V, E)$.
2: Output: A depth-first search spanning tree of the graph G.
3: Initialization:
4: $S \leftarrow \emptyset$; $\{S$ is the DFS stack$\}$
5: Push(S, 1); { vertex 1 is the root of the DFS spanning tree }
6: $InterStruct \leftarrow \emptyset$; { $InterStruct$ is the internal search structure}
7: Store($InterStruct$, 1);
8: **while** not empty (S) **do**
9:    $i \leftarrow Pop(S)$; { i is the current vertex }
10:    read($start[i]$; $stop[i]$);
11:    **while** ($start[i] \leq stop[i]$) **do**
12:       $w \leftarrow A[start[i]]$; { get next vertex ready }
13:       $start[i] \leftarrow start[i] + 1$; { update the $start[i]$ pointer in the main memory}
14:       **if** ($w \notin InterStruct$) **then**
15:          call Routine Unvisited-vertex;
16:       **end if**
17:    **end while**
18: **end while**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 9 Routine Unvisited-vertex: To make an unvisited vertex the current vertex

1: {Insert w into the internal search structure}
2: **if** ($InterStruct$ is full) **then**
3:   call **Routine** Compact-array to clean up;
4: **end if**
5: Store($InterStruct, w$);
6: read($start[w], stop[w]$);
7: read a block of A starting from $A[start[w]]$;
   {w becomes the current vertex }
8: Push(S, i); **write**($start[i]$); {update $start[i]$}
9: $i \leftarrow w$;

Explanation:

On Lines 1 to 3, if an overflow occurs at the internal search structure, then the Routine Compact-array is called to clean up the internal search structure and to compact the array A by removing all the visited vertices.

On Line 5, insert vertex w into the internal search structure to indicate that it has become a visited vertex.

On Lines 6 and 7, a segment of the adjacency list of vertex w is read into the main memory.

On Line 8, the current vertex i is pushed onto the DFS stack and its *start* pointer is adjusted accordingly.

On Line 9, vertex w becomes the current vertex of the depth-first search.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Routine Compact-array A: when *InterStruct* is full, for each vertex i, $A[start[i]..stop[i]]$ is scanned and all the visited vertices in it are deleted. The array A is compacted and the arrays *start* and *stop* are updated accordingly.

Algorithm 10 Routine Compact-array A

1: **for** i **from** 1 **to** $|V|$ **do**
2: Determine the set $B_i = \{A[j] | (start[i] < j \leq stop[i]) \wedge (A[j] \in InterStruct)\}$;
3: Remove the vertices in B_i from $A[start[i]..stop[i]]$;
4: Compact the array A to make the remaining unvisited vertices consecutive;
5: Update $start[i]$ and $stop[i]$, accordingly;
6: **endfor**
7: $InterStruct \leftarrow \emptyset$.

Explanation:

On Lines 2 and 3, the set of visited vertices in the current adjacency list of vertex i, B_i, is determined and the vertices are removed.

Lines 4 and 5 are self-explanatory.

On Line 7, the internal search structure is emptied.

3.3.2 Correctness Proof

When a vertex is examined for the first time during the depth-first search, we must mark the vertex as *visited*. To avoid using the expensive and slow I/O operation, we shall do the marking in the internal memory. The internal search structure is used for this purpose: whenever a vertex is first visited, it is inserted into the internal search structure. When it is encountered in a later stage, The internal search structured is searched and its presence in the structure indicates that it is a visited vertex. Since a clean up is performed to the internal search structure whenever an overflow occurs, a visited vertex will no longer have an entry in the search structure after the clean up. As a result, when the vertex is encountered again in a later stage, it could be mistaken as an unvisited vertex as it does not appear in the internal search structure. Fortunately, owing to the fact the the array A is also compacted whenever an overflow occurs at the search structure, once a vertex is removed from the search structure, it will never be examined again from another vertex.

The correctness proof of Algorithm EM_DFS given in [13] is extremely brief. We shall thus give a detailed proof here as the correctness proofs of the algorithms presented in the sequel rely heavily on the correctness of Algorithm EM_DFS.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 3.3.2 *Let i be the next vertex in the adjacency list of the current vertex that is being examined. Vertex i is unvisited if and only if it does not appear in the internal search structure.*

Proof: If i is unvisited, then it has never been inserted into the search structure, therefore it will not appear in the search structure. On the other hand, it i is visited, then it was inserted into the search structure when it was first examined. Therefore, if no overflow had ever occurred at the search structure, then vertex i will remain in the search structure. Otherwise, when the overflows occurred at the search structure, it would have been removed from all the adjacency lists. Therefore, in the later case, vertex i will never be encountered in the later stage of the search; in the former case, vertex i must become visited after the most recent overflow. Hence, it must remain in the search structure. □

In Algorithm EM_DFS, vertex i always represents the current vertex while vertex w always represents the vertex in the adjacency list of i that is being examined. The current vertex is shifted from i to w if and only if the Routine Unvisited-vertex is invoked on Line 15. This happens if and only if vertex w is an unvisited vertex when it is being examined. We shall show that the edge set $E_T = \{(i, w) \in E |$Routine Unvisited-vertex *is invoked on Line 15*$\}$ induces a spanning tree of G.

Theorem 3.3.1 *Algorithm EM_DFS constructs a depth-first search spanning tree for the given connected graph $G = (V, E)$.*

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

(Induction basis) Suppose G has only one vertex.

Obviously, Algorithm EM_DFS terminates after the depth-first search visited the vertex. The lemma clearly holds.

(Induction hypothesis) Suppose the lemma holds for G having less than N vertices.

(Induction step) Suppose G has $N(> 1)$ vertices. Since Algorithm EM_DFS begins its execution from vertex 1, we shall consider the graph $G' = (V - \{1\}, E - \{e \in E | (\exists w \in V) e = (1, w)\})$. Note that G' is the graph resulting from G after vertex 1 and all its incident edges are deleted. Let $G_i, 1 \leq i \leq \omega$ be the connected components of G'. Then there

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are less than N vertices in each G_i. By the induction hypothesis, Algorithm EM_DFS constructs a depth-first search spanning tree for every G_i.

Let $r_i, 1 \leq i \leq \omega$ be the root of the depth-first search spanning tree of G_i. Since Algorithm EM_DFS is invoked at vertex 1 for each r_i, the edge $(1, r_i)$ is a tree-edge. Clearly, the edge set $\{(1, r_i) | 1 \leq i \leq \omega\}$ and the depth-first search trees of the G_i's form a spanning tree of G with vertex 1 being the root. Moreover, every edge of G that is not an edge in the spanning tree must either connect two vertices of the spanning tree of some G_i's or connect a vertex with vertex 1. In the former case, by the induction hypothesis, one of the end-vertices of the edge is an ancestor of the other. In the latter case, vertex 1 is an ancestor of the other end-vertex. Therefore, every edge in G that is not an edge in the spanning tree of G is a back-edge.

Hence, Algorithm *EM_DES* constructs a depth-first search spanning tree of the graph G. □

3.3.3 Time Complexity Analysis

Theorem 3.3.2 *Let $G = (V, E)$ be a connected undirected graph represented by the arrays A, start and stop. Algorithm EM_DFS performs $O(\lceil |V|/M \rceil \operatorname{scan}(|E|) + |V|)$ I/O operations.*[1]

Proof: The I/O costs come from three sources: accessing the adjacency lists, maintaining the *DFS stack* and handling overflow occurred at the internal search structure. We shall consider each of them separately.

When a vertex i becomes the current vertex, two I/O operations are performed to read in the pair of pointers $start[i]$ and $stop[i]$, and a segment of array A starting at $A[start[i]]$. Let $A[start[i]] = k$. If vertex k is not in the internal search structure, then by Lemma 3.3.2, it is unvisited. Vertex k thus becomes the current vertex and two I/O operations are performed, reading in the pair $start[k]$ and $stop[k]$, and a segment of array A starting at $A[start[k]]$. On the other hand, if vertex k is in the internal search structure, then by Lemma 3.3.2, it is visited. As a result, the edge (i, k) is a back-edge and is thus ignored. So the next vertex to be examined is the one following k in the internal buffer. Therefore, no I/O operation is necessary unless vertex k is at the very end of the internal buffer; in which case, the internal buffer must be refilled and an I/O operation is then performed. Hence, $O(1)$ I/O operations are performed whenever a new vertex

[1]The time complexity give in [13] is $O((1 + |V|/M) \operatorname{scan}(|E|) + |V|)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is encountered or the internal buffer is exhausted. Since the encountering of each new vertex corresponds to an edge in the depth-first search spanning tree, there are a total of $O(|V|)$ I/O operations performed for the former case. The latter case can occur at most $\sum_{i \in V} O(\lceil |E_i|/DB \rceil) = O(|E|/DB + |V|)$ times giving rise to a total of $O(\text{scan}(|E|) + |V|)$ I/O operations.

Each vertex i is pushed onto the *DFS stack* when it is first visited. It is popped out of the stack whenever it becomes the current vertex and is pushed onto the stack when a new child of it is discovered. Therefore, the number of times vertex i is pushed onto or popped out of the *DFS stack* is $O(deg_{DFS}(i))$, where $deg_{DFS}(i)$ is the degree of vertex i in the depth-first search spanning tree. Since each push or pop operation involves $O(1)$ I/O operations and $|E_{DFS}| = |V| - 1$, where E_{DFS} is the edge set of the depth-first search spanning tree, the total number of I/O operations performed is thus:

$$\sum_{i \in V} O(deg_{DFS}(i)) = O(|E_{DFS}|) = O(|V|).$$

Every vertex is inserted into the internal search structure when it is first visited. If it is ever removed from the search structure, it must be caused by an overflow which occurred at the search structure. When that happens, the array A is compacted and all occurrences of the vertex in the array A are removed. As a result, the vertex will never be encountered again during the rest of the depth-first search. Hence, every vertex is inserted into the internal search structure exactly once. Since the internal search structure has a size of $O(M)$, there can be at most $\lceil |V|/M \rceil$ overflows occurred at the search structure. As the process of scanning the array A, discarding visited vertices and compacting the rest of the adjacent lists take $O(\text{scan}(|E|))$ I/Os and updating the arrays *start* and *stop* takes $O(\lceil |V|/M \rceil) = O(\lceil |V|/DB \rceil) = O(\text{scan}(|V|)) = O(\text{scan}(|E|))$ I/Os, the total number of I/O operations performed for handling the overflows occurred at the internal search structure is thus $O(\lceil |V|/M \rceil \, \text{scan}(|E|))$.

The total number of I/O operations performed by Algorithm EM_CV is thus:

$O(|V|) + O(\text{scan}(|E|) + |V|) + O(|V|) + O(\lceil |V|/M \rceil \, \text{scan}(|E|))$
$= O(\lceil |V|/M \rceil \, \text{scan}(|E|) + |V|)$. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

An External-Memory Algorithm for Graph Connectivity

Since depth-first search traverses every vertex of a *connected* undirected graph, it can thus be used to determine the connected components of an undirected graph. The idea is to start a depth-first search from an arbitrary vertex of the given graph. When the search terminates, the vertices are scanned to see if there is any unvisited vertex. If there is no unvisited vertex, then all the connected components are determined. Otherwise, start another depth-first search from an unvisited vertex. The same procedure is repeated until all vertices are visited.

4.1 A Detailed Description of the Algorithm

The algorithm is based on Algorithm EM_DFS. What we need is to maintain a variable, called *next_comp*, in the main memory. The variable is initialized to 1. A depth-first search is then performed over the given graph. When the search terminates, the arrays *start* and *stop* are scanned, starting from the entries marked by *next_comp*, until an index k such that either $start[k] \leq stop[k]$, where $next_comp \leq k \leq |V|$, or $k > |V|$ (i.e. the arrays have been completely examined) is encountered. The former case indicates that vertex k is unvisited. So, *next_comp* is updated to k and a depth-first search is carried out starting from the vertex k to determine the vertex set of another connected component. The latter case indicates that there is no unvisited vertex left in the graph. So, all the connected components have been determined and the algorithm terminates successfully.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 11 **Algorithm EM_GC**

1: **Input**: The arrays $A[1..|E|]$, $start[1..|V|]$ and $stop[1..|V|]$ representing a graph $G = (V, E)$.
2: **Output**: The connected components of the graph G.
3: $next_comp \leftarrow 1$;
4: **while** $(next_comp \leq |V|)$ **do**
5:    **read**$(start[next_comp], stop[next_comp])$;
6:    **if** $(start[next_comp] > stop[next_comp])$ **then**
7:       $next_comp \leftarrow next_comp + 1$;
8:    **else**
9:       $i \leftarrow next_comp$; Execute Routine EM_GCC;
10:    **end if**
11: **end while**

Algorithm 12 **Routine** EM_GCC

1: **Initialization:**
2: $S \leftarrow \emptyset$; $\{S$ is the *DFS stack*$\}$
3: Push(S, i); { vertex i is the root of the DFS spanning tree}
4: $InterStruct \leftarrow \emptyset$; $\{InterStruct$ is the internal search structure$\}$
5: Store$(InterStruct, i)$;
6: **write**(i); {add vertex i to the current connected component}
7: **while** not empty (S) **do**
8:    $i \leftarrow$ Pop(S);
9:    **read**$(start[i], stop[i])$; read a block of A starting from $A[start[i]]$;
10:    **while** $(start[i] \leq stop[i])$ **do**
11:       $w \leftarrow A[start[i]]$;
12:       $start[i] \leftarrow start[i] + 1$; { update the $start[i]$ pointer }
13:       **if** $(w \notin InterStruct)$ **then**
14:          **write**(w); {add vertex w to the current connected component}
15:          **call Routine** Unvisited-vertex;
16:       **end if**
17:    **end while**
18: **end while**

Algorithm 13 **Routine** Unvisited-vertex

1: {Insert w into the internal search structure}
2: **if** $(InterStruct$ is full$)$ **then**
3:    call **Routine** Compact-array to clean up;
4: **end if**
5: Store$(InterStruct, w)$;
6: read a block of A starting from $A[start[w]]$;
   $\{w$ becomes the current vertex$\}$
7: Push(S, i); **write**(start[i]); {update $start[i]$}
8: $i \leftarrow w$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 14 Routine Compact-array A

1: **for** i **from** 1 **to** $|V|$ **do**
2: Determine the set $B_i = \{A[j] | (start[i] < j \leq stop[i]) \wedge (A[j] \in InterStruct)\}$;
3: Remove the vertices in B_i from $A[start[i]..stop[i]]$;
4: Compact the array A to make the remaining unvisited vertices consecutive;
5: Update $start[i]$ and $stop[i]$, accordingly;
6: **rof**
7: $InterStruct \leftarrow \emptyset$.

4.2 Correctness Proof

Routine EM_GCC is almost identical to Algorithm EM_DFS with the following differences: a **write**(i) statement on Line 6 and a **write**(w) statement on Line 14. These statements are included to output the vertices in the current connected component generated by the depth-first search.

Theorem 4.2.1 *Algorithm EM_GC correctly determines all the connected components of the given graph* $G = (V, E)$.

Proof: (By induction on the number of connected components in the graph G).

(Induction basis) Suppose G has only one connected component (i.e. G is connected). By Theorem 3.3.1, Algorithm EM_DFS visits every vertex of the graph G. Since Routine EM_GCC and Algorithm EM_DFS differ in only some **write** statements, therefore, the routine also visits every vertex of the graph G and the **write** statements output all the vertices of G. When control returns from the first call of Routine *EM_GCC*, every vertex in G is visited. This implies that $start[next_comp] > stop[next_comp], 1 \leq next_comp \leq |V|$. Execution of the algorithm thus terminates.

(Induction hypothesis) Suppose the theorem holds for G having less than c connected components.

(Induction step) Suppose G has $c(> 1)$ connected components. Let $G' = (V', E')$ be the connected component containing the vertex 1. Since Algorithm EM_GC starts its execution from vertex 1. It therefore performs a depth-first search over G'. By Theorem 3.3.1, Algorithm EM_DFS traverses every vertex of G'. Therefore, when the first call to Routine EM_GCC from within Algorithm EM_GC terminates successfully, the connected component G' of G is identified.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let k be the vertex in $G - G'$ that has the smallest vertex identity. Then vertex k has not been visited, therefore $start[k] \leq stop[k]$. As a result, the **while** loop in Algorithm EM_GC will iterate until $next_comp = k$. From that point onwards, Algorithm EM_GC will behave as if it is running on the input graph $G-G'$. Since $G-G'$ has $c-1$ connected components, by the induction hypothesis, Algorithm EM_GC correctly finds all the connected components of $G - G'$. □

4.3 Time Complexity

Theorem 4.3.1 *Algorithm EM_GC performs* $O(\lceil |V|/M \rceil \,\mathrm{scan}(|E|) + |V|)$ *I/O operations.*

Proof: Suppose the graph G has ω connected components. Let them be $G_j = (V_j, E_j), 1 \leq j \leq \omega$, such that G_j is traversed by the jth depth-first search during the execution of Algorithm EM_GC.

Since Routine EM_GCC and Algorithm EM_DFS differ in only the two **write** statements. The number of I/Os performed by Routine EM_GCC is thus the sum of the number of I/Os performed by Algorithm EM_DFS and the number of I/Os performed by the **write** statements. By Theorem 3.3.2, the number of I/Os performed by Algorithm EM_DFS is $O(\lceil |V_j|/M \rceil scan(|E_j|) + |V_j|), 1 \leq j \leq \omega$.

The two **write** statements will involve the external memory only if the output buffer kept in the main memory for storing vertices of the current connected component has been completely filled. This could happened at most $\lceil |V_j|/(DB) \rceil$ times. Therefore, the two **write** statements could involve at most $O(\lceil |V_j|/(DB) \rceil) = O(scan(|E_j|))$ I/O operations.

Hence, for each connected component G_j, Routine EM_GCC performs a total of: $O(\lceil |V_j|/M \rceil scan(|E_j|) + |V_j|) + O(scan(|E_j|)) = O(\lceil |V_j|/M \rceil scan(|E_j|) + |V_j|)$ I/O operations.

For Algorithm EM_GC, the **read** statement on Line 5 will involve the external memory only if the segment of *start* and *stop* kept in the main memory have been completely examined. This could happened at most $\lceil |V_j|/(DB) \rceil$ times. Therefore, Line 5 could involve at most $O(\lceil |V_j|/(DB) \rceil) = O(scan(|E_j|))$ I/O operations. Since Algorithm EM_GC invokes Routine EM_GCC ω times, once for each connected component. The total number

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of I/O operations performed by Algorithm EM_GC is thus:

$$\sum_{1\leq j\leq\omega}(O(\lceil|V_j|/M\rceil scan(|E_j|)+|V_j|)+O(scan(|E_j|)))$$
$$=\sum_{1\leq j\leq\omega}O(\lceil|V_j|/M\rceil scan(|E_j|)+|V_j|)$$
$$=O(\sum_{1\leq j\leq\omega}\lceil|V_j|/M\rceil scan(|E_j|)+\sum_{1\leq j\leq\omega}|V_j|)$$
$$=O(\sum_{1\leq j\leq\omega}\lceil|V_j|/M\rceil scan(|E_j|)+|V|)$$
$$\leq O(\sum_{1\leq j\leq\omega}\lceil|V_j|/M\rceil scan(|E|)+|V|)$$
$$\leq O((\sum_{1\leq j\leq\omega}(|V_j|/M)scan(|E|)+\sum_{1\leq j\leq\omega}1)+|V|)$$
$$=O((\sum_{1\leq j\leq\omega}(|V_j|/M)scan(|E|)+\omega)+|V|)$$
$$=O((|V|/M)scan(|E|)+\omega+|V|)$$
$$=O((|V|/M)scan(|E|)+|V|)\qquad\square$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

An External-Memory Algorithm for Biconnectivity

5.1 An EM Algorithm for Detecting Cut-Vertices

We adapt Tarjan's LOWPOINT method to design an external-memory algorithm for detecting the cut vertices of an undirected connected graph based on the algorithm EM_DFS.

5.1.1 Input Data Structures

The proposed algorithm, Algorithm EM_CV, uses the input data structure of Algorithm EM_DFS with the following modifications. The input graph $G = (V, E)$ is represented by three arrays, $A[1..(|V| + |E|)]$, $start[1..|V|]$ and $stop[1..|V|]$, stored on the external disks. Again, we assume, without loss of generality, that $V = \{1, 2, 3, \cdots, n\}$. The array A consists of the adjacency lists of all the vertices.

To reduce I/O cost in computing LOWPOINT, we modify the array A slightly so that the *last* entry in the adjacency list, $A[stop[i]]$, of every vertex i is reserved for storing a partial value of the LOWPOINT of that vertex. Specifically, only the segment of array A, $A[start[i]..stop[i] - 1]$, contains the adjacency list of vertex $i, 1 \leq i \leq |V|$. Furthermore, $stop[i] + 1 = start[i + 1], 1 \leq i < |V|$.

The internal search structure is also modified. it stores not only the vertices themselves but also their *DFS number.*

The DFS stack keeps in each of its entries a vertex whose role as the current vertex is temporarily suspended, along with the LOWPOINT value, the DFS number, and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DFS number of the *parent vertex* of that vertex.

5.1.2 Computing LOWPOINT

We shall explain how to compute the LOWPOINT value of every vertex. Recall that the LOWPOINT value of a vertex v is defined as:

$$LOWPOINT(v) = \min(\{dfs(v)\} \cup \{LOWPOINT(w) | w \text{ is a child of } v\} \cup \{dfs(w) | (v,w) \text{ is a back-edge}\})$$

For each vertex i, we shall use $low_{(i)}$ to denote its LOWPOINT value.

As was mentioned earlier, $A[stop[i]]$ is reserved to store an initialized value of $low_{(i)}$. It is initialized to ∞ (an arbitrary number larger than $|V|$, the largest depth-first search number) and is updated whenever an overflow occurs at the internal search structure.

When an overflow occurs at the internal search structure, the array A is compacted. For each vertex i, the adjacency list of vertex i is scanned and every vertex k in it that is a visited vertex (i.e. it appears in the internal search structure) is eliminated. However, as the edge (i,k) is a back-edge of i, the value $dfs(k)$ must be used to update $low_{(i)}$ if $dfs(k) < low_{(i)}$ at that point of time. Since the current value of $low_{(i)}$ is stored somewhere on the DFS stack, we cannot update $low_{(i)}$ immediately. Instead, we determine the set of visited vertices in the adjacency list of vertex i, pick the one whose depth-first-search number is the smallest and store that number in $A[stop[i]]$. Specifically, whenever array A is compacted, we determine the set $\bar{B}_i = \{k | \text{vertex } k \text{ appears in the internal search structure}\}$ using the vertices in $A[start[i]..stop[i]]$ and update $A[stop[i]]$ with $\delta_{min}(i) = \min\{dfs(k) | k \in \bar{B}_i\}$.

Let vertex i be the current vertex of the depth-first search. Let j be the vertex $A[start(i)]$. If j is in the internal search structure, then (i,j) is a back-edge and $dfs(j)$ is retrieved from the search structure. Furthermore, if $dfs(j) < low_{(i)}$ then $low_{(i)}$ is updated to $dfs(j)$. If j is not in the internal search structure, vertex j is unvisited. $low_{(i)}$ is then pushed onto the DFS stack along with vertex i, $dfs_{(i)}$ and $dfs_{(p(i))}$, where $p(i)$ is the parent vertex of i. The depth-first search then advances to vertex j making vertex j the current vertex of the search. As a result, vertex j is inserted into the internal search structure along with its depth-first search number $dfs(j)$. Furthermore, $low_{(j)}$ is initial-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ized to $dfs(j)$. When the search backtracks from vertex j to vertex i at a later stage, the value of $low_{(j)}$ is finalized. After $low_{(i)}$ is popped out of the DFS stack, it is updated to $\min\{low_{(i)}, low_{(j)}\}$. Furthermore, if $low_{(j)} \geq dfs(i)$, then vertex i is a cut-vertex (Theorem 5.1.3).

When vertex i is the current vertex and $start(i) = stop[i]$, the adjacency list of vertex i is completely examined. The value stored in $A[stop[i]]$ (which is a partial value of $low_{(i)}$) is then retrieved to update $low_{(i)}$. However, if $A[stop[i]] = dfs(parent(i))$, then it is an indication that the parent-edge of vertex i was mistakenly used as an outgoing back-edge of vertex i in computing $A[stop[i]]$ when the array A was compacted earlier. The actual value of $A[stop[i]]$ should be $dfs(i)$ which would have no impact on the final value of $low_{(i)}$. So, the value in $low_{(i)}$ remains unchanged. Otherwise, $low_{(i)}$ is updated to $\min\{low_{(i)}, A[stop[i]]\}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 15 Algorithm LOWPOINT

1: **Input**: The arrays $A[1..|V|+|E|]$, $start[1..|V|]$ and $stop[1..|V|]$ representing a graph $G=(V,E)$.
2: **Output**: $low_{(i)}, \forall i \in V$.
3: **Initialization:**
4: $i \leftarrow 1$; $dfs(i) \leftarrow 1$; $dfs \leftarrow 2$; $low_{(i)} \leftarrow 1$;
5: $S \leftarrow \emptyset$; {S is the *DFS stack*}
6: Push$(S, (i, dfs(i), low_{(i)}, \perp))$;
7: $InterStruct \leftarrow \emptyset$; {$InterStruct$ is the internal search structure}
8: Store$(InterStruct, (i, dfs(i)))$;
9: **while** not empty (S) **do**
10: $\quad (i, dfs(i), low_{(i)}, dfs(p(i))) \leftarrow$ Pop(S);
11: $\quad$ **read**$(start[i], stop[i])$; read a block of A starting from $A[start[i]]$;
12: $\quad$ **while** $start[i] < stop[i]$ **do**
13: $\quad\quad w \leftarrow A[start[i]]$;
14: $\quad\quad$ **if** $(w \notin InterStruct)$ **then**
15: $\quad\quad\quad$ **call Routine** Unvisited-vertex;
16: $\quad\quad$ **else**
17: $\quad\quad\quad low_{(i)} \leftarrow \min(low_{(i)}, dfs(w))$;
18: $\quad\quad\quad start[i] \leftarrow start[i] + 1$;
19: $\quad\quad$ **end if**
20: $\quad$ **end while**
21: $\quad$ **if** $(A[stop[i]] < dfs(p(i)))$ **then**
22: $\quad\quad low_{(i)} \leftarrow \min(low_{(i)}, A[stop[i]])$ {Finalize $low_{(i)}$}
23: $\quad$ **end if**
$\quad$ { Update LOWPOINT of parent vertex }
24: $\quad (k, dfs(k), low_{(k)}, dfs(p(k))) \leftarrow$ Pop (S);
25: $\quad low_{(k)} \leftarrow \min(low_{(k)}, low_{(i)})$;
26: $\quad$ Push $(S, (k, dfs(k), low_{(k)}, dfs(p(k))))$;
27: $\quad i \leftarrow k$; {backtrack to the parent vertex k}
28: **end while**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Routine Unvisited-vertex:

Algorithm 16 Encounter an unvisited vertex

1: {Insert w into the internal search structure}
2: **if** ($InterStruct$ is full) **then**
3: call **Routine** Compact-array to clean up;
4: **end if**
5: $dfs(w) \leftarrow dfs$; $dfs \leftarrow dfs + 1$; $dfs(p(w)) \leftarrow dfs(i)$;
6: $low_{(w)} \leftarrow dfs(w)$; {Initialize $low_{(w)}$ }
7: Store($InterStruct, (w, dfs(w), i)$);
8: **read**($start[w], stop[w]$); read a block of A starting from $A[start[w]]$;
{w becomes the current vertex, so save i }
9: $start[i] \leftarrow start[i] + 1$; **write**($start[i]$); {update $start[i]$};
10: Push($S, (i, dfs(i), low_{(i)}, dfs(p(i)))$); {save the current vertex on the DFS stack}
11: $i \leftarrow w$;

Explanation:

If an overflow occurs in the internal search structure, the Routine Compact-array is called to clean up the internal search structure and to compact the array A by removing all the visited vertices.

On Line 5, a depth-first search number is assigned to vertex w

On Line 6, $low_{(i)}$ is initialized to $dfs(w)$.

On Line 7, vertex w is stored into the internal search structure, making it the current vertex.

On Lines 9 and 10, the $start$ pointer of the current vertex i is updated and saved in the external memory; the values, $i, dfs(i), low_{(i)}$ and $dfs(p(i))$ are saved on the DFS stack.

On Line 11, vertex w becomes the current vertex of the depth-first search.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Routine Compact-array: when *InterStruct* is full, for each vertex i, $A[start[i]..stop[i]-1]$ is scanned to compute $\delta_{min}(i)$ and all the visited vertices in it are deleted.

Algorithm 17 Compact the array A when *InterStruct* is full

1: **for** i **from** 1 **to** $|V|$ **do**
2: $start[i] \leftarrow stop[i-1]+1$; {Reset $start[i]$, assuming $stop[0]=0$}
3: Determine the set $\bar{B}_i = \{A[j] | (start[i] \leq j < stop[i]) \wedge (A[j] \in InterStruct)\}$;
4: $A[stop[i]] \leftarrow \min\{A[stop[i]], \min\{dfs(v) | v \in \bar{B}_i\}\}$;
5: Remove the vertices in $\bar{B}_i$ from $A[start[i]..stop[i]-1]$;
6: Compact $A[start[i]..stop[i]]$ to make the remaining unvisited vertices consecutive;
7: Append $A[start[i]..stop[i]]$ to $A[1..stop[i-1]]$
8: Update $start[i]$ and $stop[i]$, accordingly;
9: **rof**;
10: $InterStruct \leftarrow \emptyset$.

Explanation:

On Line 2, the pointer *start* is reset to the beginning of the current adjacency list of vertex i.

On Line 3, the set of visited vertices in the current adjacency list of vertex i, $\bar{B}_i$, is determined. (Note that by Lemma 3.3.2, a vertex is visited if and only if it is in the internal search structure.)

On Line 4, the smallest depth-first search number among the vertices in $\bar{B}_i$ is determined and is used to update $A[stop[i]]$.

On Line 5, all the *visited* vertices in the current adjacency list of vertex i are removed.

On Lines 6 and 7, the vertices remain in $A[start[i]..stop[i]-1]$ are packed with $A[stop[i]]$ into a block of consecutive locations, starting from $A[stop[i-1]+1]$.

On Line 8, the pointers $start[i]$ and $stop[i]$ are adjusted accordingly.

The internal search structure is emptied on Line 10.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1.3 Correctness Proof

Let $B_i = \{j|(i,j)$ is a back-edge$\}$,

B_i' be the first non-empty set of visited vertices that appear in the adjacency list of vertex i during a clean up of the array A,

B_i'' be the set of visited vertices that appear in the adjacency list of vertex i during subsequent clean ups of the array A, and

B_i^{DFS} be the set of visited vertices found on the adjacency list of vertex i when the list is scanned for an unvisited vertex during the depth-first search (not during a clean up).

Since every visited vertex found on the adjacency list of vertex i during a clean up or a search for an unvisited vertex corresponds to a back-edge of which i is an end-point, and vice versa, it is easily verified that: $B_i = B_i' \cup B_i'' \cup B_i^{DFS}$.

Lemma 5.1.1 *For each $i \in V$, $\min\{dfs(j)|j \in B_i'\} < dfs(k)$, where k is any visited vertex appearing in the adjacency list of vertex i during a subsequent clean up.*

Proof: Since $k \notin B_i'$ and vertex k is visited after any of the vertices in B_i', therefore $dfs(j) < dfs(k), \forall j \in B_i'$. It follows that $\min\{dfs(j)|j \in B_i'\} < dfs(k)$. □

Lemma 5.1.2 *For each vertex $i \in V$,*

$$low_{(i)} = \min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|j \in B_i'\} \cup \{dfs(j)|j \in B_i^{DFS}\})$$

Proof: By Lemma 5.1.1, $\min(\{dfs(j)|j \in B_i'\} \cup \{dfs(j)|j \in B_i''\}) = \min\{dfs(j)|j \in B_i'\}$. It then follow from the definition of LOWPOINT that:

$$\begin{aligned}
low_{(i)} &= \min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|(i,j) \text{ is a back-edge}\}) \\
&= \min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|j \in B_i' \cup B_i'' \cup B_i^{DFS}\}) \\
&= \min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup (\{dfs(j)|j \in B_i'\} \cup \{dfs(j)|j \in B_i''\} \\
&\qquad \cup \{dfs(j)|j \in B_i^{DFS}\})) \\
&= \min(\min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|j \in B_i^{DFS}\}), \\
&\qquad \min(\{dfs(j)|j \in B_i'\} \cup \{dfs(j)|j \in B_i''\})) \\
&= \min(\min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|j \in B_i^{DFS}\}), \\
&\qquad \min(\{dfs(j)|j \in B_i'\})) \\
&= \min(\{dfs(i)\} \cup \{low_{(j)}|j \text{ is a child of } i\} \cup \{dfs(j)|j \in B_i'\} \cup \{dfs(j)|j \in B_i^{DFS}\})
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

□

Theorem 5.1.1 *For each vertex $i \in V$, $low_{(i)}$ is correctly computed by Algorithm LOWPOINT.*

Proof: We shall apply induction on the *level* of i in the DFS tree, where the level of a vertex v is the number of edges on the path connecting the root with v in the DFS tree.

(Induction basis) Let i be a vertex at the highest level of the DFS tree. Then i must be a leaf in the DFS tree. As a result, $\{low_{(j)} | j \textit{ is a child of } i\} = \emptyset$. When vertex i is found as an unvisited vertex in the adjacency list of the current vertex (i.e. When the depth-first reaches vertex i), $low_{(i)}$ is correctly initialized to $dfs(i)$ on Line 6 in Routine Unvisited-vertex. Vertex i then becomes the current vertex. Let $j \in B_i$. Since i is a leave, vertex j must be visited when it is encountered in the adjacency list of i. It is encountered as a member of B_i' or B_i'' or B_i^{DFS}. In the first two cases, $dfs(j)$ is correctly used in updating $A[stop[i]]$ on Line 4 of Routine Compact-array. In the last case, $dfs(j)$ is correctly used in updating $low_{(i)}$ on Line 17 of Algorithm LOWPOINT. When the adjacency list of vertex i is completely examined, $start[i] = stop[i]$, $low_{(i)} = \min(\{dfs(i)\} \cup \{dfs(j) | j \in B_i^{DFS}\})$, and $A[stop[i]] = \min(\{dfs(j) | j \in B_i'\})$. Therefore, $low_{(i)}$ is correctly given the final value of $\min(\{dfs(i)\} \cup \{dfs(j) | j \in B_i'\} \cup \{dfs(j) | j \in B_i^{DFS}\})$ on Line 22 of Algorithm LOWPOINT.

(Induction hypothesis) Suppose $low_{(i)}$ is correctly computed for every vertex i lying on level h or higher (farther from the root) of the DFS tree.

(Induction Step) Let i be a vertex lying on level $h - 1$ of the DFS tree. As with the base case, when vertex i is found as an unvisited vertex in the adjacency list of the current vertex, vertex i becomes the current vertex and $low_{(i)}$ is correctly initialized to $dfs(i)$ on Line 6 in Routine Unvisited-vertex. Let $j \in B_i$. When vertex j is encountered in the adjacency list of i, it is either unvisited or visited. In the former case, Routine Unvisited-vertex is invoked for vertex j. Since vertex j is on level h, by the induction hypothesis, when the adjacency list of vertex j is completely examined and the depth-first search backtracks to vertex i making vertex i the current vertex again, $low_{(j)}$ is correctly computed. As a result, $low_{(i)}$ is correctly updated on Line 24.

In the latter case, vertex i is encountered as a member of B_i' or B_i'' or B_i^{DFS}. In the first two cases, $dfs(j)$ is correctly used in updating $A[stop[i]]$ on Line 4 of Routine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Compact-array. In the last case, $dfs(j)$ is correctly used in updating $low_{(i)}$ on Line 17 of Algorithm LOWPOINT. When the adjacency list of vertex i is completely examined, $start[i] = stop[i]$, $low_{(i)} = \min(\{dfs(i)\} \cup \{low_{(j)} | j \text{ is a child of } i\} \cup \{dfs(j) | j \in B_i^{DFS}\})$, and $A[stop[i]] = \min(\{dfs(j) | j \in B_i'\})$. Therefore, $low_{(i)}$ is correctly given the final value of $\min(\{dfs(i)\} \cup \{low_{(j)} | j \text{ is a child of } i\} \cup \{dfs(j) | j \in B_i'\} \cup \{dfs(j) | j \in B_i^{DFS}\})$ on Line 22 of Algorithm LOWPOINT. □

5.1.4 Time Complexity Analysis

Theorem 5.1.2 *Let $G = (V, E)$ be a connected undirected graph represented by the arrays A, start and stop. Algorithm LOWPOINT performs $O(\lceil |V|/M \rceil scan(|E|) + |V|)$ I/O operations to compute $low_{(i)}, \forall i \in V$.*

Proof: The I/O cost comes from three resources: accessing the adjacency lists, maintaining the DFS stack and handling overflow occurred at the internal search structure.

When a vertex i becomes the current vertex, an I/O operation is performed to read in a block of vertices starting at $A[start[i]]$. Let $A[start[i]] = k$. If vertex k is not in the internal search structure, then by Lemma 3.3.2, it is unvisited. Vertex k thus becomes the current vertex and an I/O operation is performed for it reading in a block starting at $A[start[k]]$. On the other hand, if vertex k is in the internal search structure, then by Lemma 3.3.2, it is visited. As a result, the edge (i, k) is a back-edge. So the next vertex to be examined is the one following k in the internal buffer. Therefore, no I/O operation is necessary unless vertex k is at the very end of the internal buffer; in which case, the internal buffer must be refilled and an I/O operation is then performed. Hence, an I/O operation is performed whenever a new vertex is encountered or the internal buffer is exhausted. Since the encountering of each new vertex corresponds to an edge in the depth-first search spanning tree, there are a total of $O(|V|)$ I/O operations performed for the former case. The latter case can occur at most $|E|/DB$ times giving rise to a total of $|E|/DB = \text{scan}(|E|)$ I/O operations.

Each vertex i is pushed onto the DFS stack when it is first visited. It is popped out of the stack whenever it becomes the current vertex and is pushed onto the stack when a new child of its is discovered. Therefore, the number of times vertex i is pushed onto or popped out of the DFS stack is $O(deg(i))$, where $deg(i)$ is the degree of vertex i in the depth-first tree. Furthermore, vertex i is also popped out of and then pushed back to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

stack whenever the search backtracks from a child of i to i in order to update $low_(i)$ with the LOWPOINT of that child. This can happen $O(deg(i))$ time. Since each push or pop operation involves $O(1)$ I/O operations, the total number of I/O operations performed is thus

$$\sum_{i \in V} O(deg(i)) = O(|V|)$$

Since each vertex is inserted into the internal search structure once and the internal search structure has a size of $O(M)$, there can be at most $\lceil |V|/M \rceil$ overflows occurred at the search structure. As the process of scanning the array A, discarding visited vertices and compacting the rest of the adjacent lists take $O(scan(|E|))$ I/Os, the total number of I/O operations performed to handle overflows occurred at the internal search structure is $O(\lceil |V|/M \rceil scan(|E|))$.

The total number of I/Os performed by Algorithm EM-CV is thus:
$O(|V|) + O(scan(|E|)) + O(|V|) + O(\lceil |V|/M \rceil scan(|E|))$
$= O(\lceil |V|/M \rceil scan(|E|) + |V|)$. □

5.1.5 Detecting the cut-vertices

Once we know how to compute $low_{(i)}, \forall i \in V$, determining the cut-vertices become straightforward owing to the following lemma.

Theorem 5.1.3 *A vertex v is a cut-vertex if and only if*

- *v is the root of the depth-first search tree and has two or more children, or*
- *v is not the root and there is a child w of v such that $low_{(w)} \geq dfs(v)$*

Proof: See [39]. □

To determine the cut-vertices, we modify Algorithm LOWPOINT based on Theorem 5.1.3 as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Replace Line 25 with the following lines and let the resulting algorithm be called Algorithm EM_CV.

```
if (dfs(i) ≠ 1) then { /* Vertex i is not the root of the DFS tree */ }
    if (dfs_(k) < low(i)) then
        low_(i) ← min{low_(i), low_(k)};
    else
        output vertex i as a cut-vertex;
else if (vertex k is the first child of vertex i) then
        mark down vertex i (the root) has a child;
    else
        output vertex i as a cut-vertex;
```

Theorem 5.1.4 *Algorithm EM_CV correctly determines the cut-vertices of the graph* $G = (V, E)$.

Proof: Immediate from Theorem 5.1.1 and Theorem 5.1.3 □

Theorem 5.1.5 *Let* $G = (V, E)$ *be a connected undirected graph represented by the arrays A, start and stop. Algorithm EM_CV performs* $O(\lceil |V|/M \rceil scan(|E|) + |V|)$ *I/O operations to determine the cut-vertices of G.*

Proof: Immediate from Theorem 5.1.2 □

5.2 An EM Algorithm for Detecting Biconnected Components

As was mentioned in Chapter 2, the cut-vertices of a graph $G = (V, E)$ determine the biconnected components of G. Therefore, we can develop an EM biconnected component algorithm, EM_BCC, based on Algorithm EM_CV. The objective of Algorithm EM_BCC is to determine the vertex set of each biconnected component of the given graph. In [39], Tarjan determined the biconnected components by identifying their edge sets. We shall determine the biconnected components by identifying their vertex sets. This is to ensure that the size of the output is $O(|V|)$ instead of $O(|E|)$. Note that $|E| = O(|V|^2)$ rather than $O(|V|)$. This idea is based on the following lemmas.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 5.2.1 *The set of vertices of a biconnected component induces a subtree of the DFS tree, whose root is either a cut-vertex or the root of the DFS tree, and has a unique child in that subtree.*

Proof: See [39]. □

Theorem 5.2.1 *Let vertex j be a child of vertex i such that $low_{(j)} \geq dfs(i)$. The edge (i, j) and the subtree of the DFS tree rooted at j with all the proper subtrees in it whose roots are cut-vertices being trimmed off form a depth-first search spanning tree of a biconnected component of G.*

Proof: An immediate consequence of Theorem 5.1.3 and Lemma 5.2.1. □

5.2.1 The Description of EM_BCC

During an execution of Algorithm EM_CV, when the depth-first search backtracks from a child j to a vertex i, if the condition $low_{(j)} \geq dfs(i)$ holds, then vertex i is a cut-vertex according to Theorem 5.1.3. Moreover, according to Theorem 5.2.1, the edge (i, j) and the subtree of the DFS tree whose root is vertex j and in which all the proper subtrees rooted at cut-vertices are trimmed off form a depth-first search spanning tree of a biconnected component of G. It is easily verified that this subtree of j is a *maximal* subtree of j with *no* cut-vertex of G residing in it as an internal vertex. We can thus modify Algorthm EM_CV to develop an EM algorithm, called Algorithm EM_BCC, to generate all the biconnected components of the given graph G.

The idea underlying our algorithm is to use an additional stack, called the *BCC stack*, to generate the vertex set of each biconnected component. The idea is as follows: Whenever an unvisited vertex is encountered during the search, the vertex is pushed onto the BCC stack. Whenever the search backtracks from a vertex j to a vertex i such that $low_{(j)} \geq dfs(i)$, the BCC stack is popped until the vertex j is popped out of the stack. The vertices popped out from the stack and the vertex i form the vertex set of the biconnected component containing the edge (v, w).

The following is a formal description of Algorithm EM_BCC:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 18 Algorithm EM_BCC

```
1: Input: The arrays A[1..|V| + |E|], start[1..|V|] and stop[1..|V|] representing a graph
   G = (V, E).
2: Output: The vertex sets of the biconnected components of G.
3: Initialization:
4: i ← 1; dfs(i) ← 1; dfs ← 2; low_(i) ← 1;
5: S ← ∅; BCC ← ∅; {BCC is the BCC stack}
6: Push(S, (i, dfs(i), low_(i), ⊥));
7: Push(BCC_S, i);
8: InterStruct ← ∅; {InterStruct is the internal search structure}
9: Store(InterStruct, (i, dfs(i)));
10: while not empty (S) do
11:   (i, dfs(i), low_(i), dfs(p(i))) ← Pop(S);
12:   read(start[i], stop[i]); read a block of A starting from A[next[i]];
13:   while start[i] < stop[i] do
14:     w ← A[start[i]];
15:     if (w ∉ InterStruct) then
16:       call Routine Unvisited-vertex;
17:     else
18:       low_(i) ← min(low_(i), dfs(w));
19:       start[i] ← start[i] + 1;
20:     end if
21:   end while
22:   if (A[stop[i]] < dfs(p(i))) then
23:     low_(i) ← min(low_(i), A[stop[i]]) {Finalize low'_(i) svalue}
24:   end if
      {Update LOWPOINT of parent vertex }
25:   (k, dfs(k), low_(k), dfs(p(k))) ← Pop (S);
26:   if low_(i) < dfs(k) then
27:     low_(k) ← min(low_(k), low_(i));
28:   else
29:     call Routine Generate-BCC;
30:   end if;
31:   Push (S, (k, dfs(k), low_(k)), dfs(p(k)));
32:   i ← k; {backtrack to the parent vertex k}
33: end while
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 19 Routine Unvisited-vertex: Encounter an unvisited vertex

1: {Insert w into the internal search structure}
2: **if** (*InterStruct* is full) **then**
3: call **Routine** Compact-array to clean up;
4: **end if**
5: $dfs(w) \leftarrow dfs$; $dfs \leftarrow dfs + 1$; $dfs(p(w)) \leftarrow dfs(i)$;
6: $low_{(w)} \leftarrow dfs(w)$; {Initialize $low_{(w)}$}
7: Store($InterStruct, (w, dfs(w))$);
8: **read**($start[w], stop[w]$); read a block of A starting from $A[start[w]]$;
{w becomes the current vertex, so save i}
9: $start[i] \leftarrow start[i] + 1$; **write**($start[i]$); {update $start[i]$};
10: Push($S, (i, dfs(i), low_{(i)}, dfs(p(i)))$); {save the current vertex on the *DFS stack*}
11: Push(BCC_S, i); {push the current vertex onto the *BCC stack*}
12: $i \leftarrow w$; { vertex w becomes the current vertex }

Explanation: Same as Section 3.3.1 except Line 11.

Algorithm 20 Compact the array A when *InterStruct* is full

1: **for** i **from** 1 **to** $|V|$ **do**
2: $start[i] \leftarrow stop[i-1] + 1$; {assuming $stop[0] = 0$}
3: Determine the set $\bar{B}_i = \{A[j] | (start[i] < j < stop[i]) \wedge (A[j] \in InterStruct)\}$;
4: $A[stop[i]] \leftarrow \min\{A[stop[i] - 1], \min\{dfs(v) | v \in B_i\}\}$;
5: Remove the vertices in $\bar{B}_i$ from $A[start[i]..stop[i]]$;
6: Compact $A[start[i]..stop[i]]$ to make the remaining unvisited vertices consecutive;
7: Append $A[start[i]..stop[i]]$ to $A[1..stop[i-1]]$
8: Update $start[i]$ and $stop[i]$, accordingly;
9: **rof**;
10: Empty *InterStruct*.

Explanation: Same as Section 3.3.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 21 Routine Generate-BCC: Generate the vertex set of a biconnected component

1: {a biconnected component whose spanning tree within the DFS tree is rooted at vertex k and contains the edge (k, i) is to be generated.}
2: $BC_{(k,i)} \leftarrow \{k\}$;
3: **repeat**
4: $u \leftarrow$ Pop(BCC_S);
5: $BC_{(k,i)} \leftarrow BC_{(k,i)} \cup \{u\}$;
6: **until** $u = i$;

5.2.2 Correctness Proof

Theorem 5.2.2 *Algorithm EM_BCC generates the vertex set of each biconnected component of the graph $G = (V, E)$.*

Proof:(By induction on the number of biconnected components)

(Induction basis)

Suppose G has only one biconnected component (i.e. G is biconnected).

Then G has no cut-vertex. By Lemma 5.1.3, $\forall v \in V$ such that $v \neq r$, there is no child w of v such that $low_{(w)} \geq dfs(v)$. Therefore Routine Generate-BCC is never invoked for every vertex $v \neq r$. As a result, no vertex had been popped out of the *BCC stack* when the search backtracks to the root r. Let w be a child of r. By Lemma 5.1.3, w is the only child of r. Since $dfs(r) = 1$ and $low_{(w)} \geq 1$, $low_{(w)} \geq dfs(r)$ which results in Routine Generate_BCC being invoked.

Since the root r has w as the only child, vertex w is the first vertex visited after r and is pushed onto the *BCC stack* right after r. It thus lies at the bottom of the *BCC stack* right above r which is at the very bottom of the stack. It follows that when vertex w is popped out of the *BCC stack*, every vertex of G except the root r has been popped out. However, as the $BCC_{(r,w)}$ is initialized to $\{r\}$, all the vertices of G are thus included in $BCC_{(r,w)}$ when execution of EM_BCC terminates. The algorithm thus correctly generate the vertex set of the (only) biconnected component of G.

(Induction hypothesis) Suppose the theorem holds for any graph G having less than b biconnected components.

(Induction step) Suppose G has b ($b > 1$) biconnected components.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Suppose Routine Generate-BCC is first executed when the search backtracks from a vertex w to its parent vertex v. This happens because $low_{(w)} \geq dfs(v)$. We shall show that the vertices on the BCC stack lying above the vertex v form the vertex set of a biconnected component of G with v.

Let $T(w)$ be the subtree of the DFS tree rooted at w. Since this is the first time Routine Generate_BCC is invoked, no cut-vertex had been discovered before. Therefore, the subtree $T(w)$ contains no cut-vertex. By Theorem 5.2.1, the edge (v, w) and the subtree $T(w)$ form the spanning tree of a biconnected component, say B, of the graph G. As a result, the vertex set of the biconnected component B consists of the vertices of $T(w)$ and the vertex v.

Since vertex w is a child of vertex v, it must lie directly above v when it is pushed onto the *BCC stack*. All the other vertices in $T(w)$ are visited by the depth-first search after vertex w and are thus lie above w on the *BCC stack*. As a result, when Routine EM_BCC is invoked and vertices are being popped out of the BCC stack until vertex w is popped out, it is precisely those vertices of $T(w)$ that are popped out. Therefore, when Routine Generate_BCC terminates its execution, $BCC_{(v,w)}$ contains all the vertices of the biconnected connected component B.

After generating the vertex set $BCC_{(v,w)}$, Algorithm EM_BCC behaves as it would on the graph $G - B$. Since $G - B$ has less then b biconnected components, by the induction hypothesis, Algorithm EM_BCC correctly generates all its biconnected components. The Theorem thus follows. $\square$

5.2.3 Time Complexity Analysis

Theorem 5.2.3 *Algorithm EM_BCC performs* $O(\lceil |V|/M \rceil \, \text{scan}(|E|) + |V|)$ *I/O operations to generate all the biconnected components of the graph* $G = (V, E)$.

Proof: Since Algorithm EM_BCC and Algorithm EM_CV differ in the routine **Generate-BCC**. Therefore, the number of I/Os performed by Algorithm EM_BCC is the sum of the number of I/Os performed by Algorithm EM_CV and the number of I/Os performed by Routine **Generate-BCC**. By Theorem 5.1.5, the number of I/Os performed by Algorithm EM_CV is $O(\lceil |V|/M \rceil \, \text{scan}(|E|) + |V|)$.

Since popping out one vertex from the BCC stack takes one I/O operation and every

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertex is popped out once during an execution of Algorithm EM_BCC, the total number of I/O operations performed by Routine **Generate-BCC** is thus $O(|V|)$.

Hence, Algorithm EM_BCC preforms $O(\lceil |V|/M \rceil \, \text{scan}(|E|) + |V|)$ I/O operations. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Comparison of Time Complexities

We shall compare our results with the previously known results. We shall consider the cases in which the input graph satisfies the condition $\lceil |V|/M \rceil = \Theta(1)$ (i.e. $|V|$ is larger than M by a constant factor). This is weaker than the semi-external model proposed by Abello *et al.* [1]. Graphs satisfying this condition are not uncommon in real-life situations. For example, it has been pointed out in Abello *et al.* that a graph induced by monitoring long-term traffic patterns among relatively few nodes in a network, and a graph induced by telephone calls in the AT&T network satisfy such a condition.

For the Graph Connectivity problem, the EM algorithm of Chiang et al. [13] performs: $O(\min\{\text{sort}(|V|^2), \log(|V|/M)\,\text{sort}(|E|)\})$ I/O operations.

When $\lceil |V|/M \rceil = \Theta(1)$, their algorithm performs:
$O(\min\{\text{sort}(|V|^2), \text{sort}(|E|)\}) = O(\text{sort}(|E|))$ I/O operations.

By contrast, our algorithm performs $O(\text{scan}(|E|) + |V|)$ operations.

Since $\text{sort}(|E|) > \text{scan}(|E|)$, therefore

$O(\text{sort}(|E|)) > O(\text{scan}(|E|) + |V|)$

$\Leftrightarrow \text{sort}(|E|) > |V|$

$\Leftrightarrow (|E|/DB) \log_{\frac{M}{B}}(|E|/B) > |V|$

$\Leftrightarrow |E| > (DB|V|)/\log_{\frac{M}{B}}(|E|/B)$

In particular, since $\lceil |V|/M \rceil = \Theta(1)$, therefore $|V|/M \leq k$, or some constant k, which implies that $|V| \leq kM$. But $|E| < |V|^2$. It follows that:

$|E| < (kM)^2 \Rightarrow |E|/B < (kM)^2/B \Rightarrow \log_{\frac{M}{B}}(|E|/B) \leq k'$, for some constant k'.

As a result, when $|E| = \Omega(DB|V|)$, $O(\text{sort}(|E|)) > O(\text{scan}(|E|) + |V|)$ which implies that our algorithm outperforms the algorithm of Chiang *et al.* This includes dense graphs (i.e. $|E| = \Theta(|V|^2)$ as special cases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

On the *Semi-external Memory* model, the input graph satisfies the condition: $|V| \leq M < |E|$ (In other words, the vertex set V is small enough to completely fit inside the main memory whereas the edge set E is not). Abello et al. [1] presented an EM algorithm for the connectivity problem that performs $O(\text{scan}(|E|) \log_{M/B} C(G))$ I/O operations, where $C(G)$ is the number of connected components in G. Since $|V| \leq M \Rightarrow |V|/M = \Theta(1))$, our algorithm EM_GCC thus performs $O(\text{scan}(|E|)$ I/O operations on semi-external model. This is better than that of Abello et al. by a factor of $\log_{M/B} C(G)$.

Munagala et al. [32] proposed yet another EM algorithm for determining the connected components of a graph. Their algorithm requires:

$O(\max\{1, \log\log(|V|DB/|E|)\}(|E|/|V|)\,\text{sort}(|V|)$ I/O operations.

For $|E| = \Theta(|V|^2)$ (i.e. for dense graphs), their algorithm performs

$O(\max\{1, \log\log(|V|DB/|E|)\}(|E|/|V|)\,\text{sort}(|V|))$
$> O((|E|/|V|)\,\text{sort}(|V|))$
$= O(|V|\,\text{sort}(|V|))$
$= O(\text{scan}(|V|^2) \log_{\frac{M}{B}}(|V|/B))$
which is a factor of $\log_{\frac{M}{B}}(|V|/B)$ larger than ours.

The EM biconnected component algorithm of Chiang et al. performs $O(\min\{\text{sort}(|V|^2); \log(|V|/M)\,\text{sort}(|E|)\})$ I/O operations. For dense graphs, i.e. $|E| = \Theta(|V|^2)$, the I/O complexity is:

$O(\min\{\text{sort}(|V|^2); \log(|V|/M)\,\text{sort}(|E|)\}) = O(\log(|V|/M)\,\text{sort}(|V|^2))$.

When $|V|/M = \Theta(1)$, their algorithm requires: $O(\text{sort}(|V|^2))$ I/Os.
By contrast, our algorithm requires $O(\lceil |V|/M \rceil\,\text{scan}(|V|^2) + |V|)$
$= O(\text{scan}(|V|^2) + |V|)$
$= O((|V|^2)/DB + |V|)$
$= O(|V|^2)/DB) \quad (\because |V| > M > DB)$
$= O(\text{scan}(|V|^2))$
Since, $\text{scan}(|V|^2)) < \text{sort}(|V|^2))$, our algorithm thus has a better performance.

In general, when $|E| > (DB)|V|$ (this includes dense graph as a special case), $|E|/DB > |V|$ which implies that $\text{scan}(|E|) > |V|$. Our algorithm still requires $O(\text{scan}(|E|)$ I/Os.

Hence, Algorithm EM_BCC has a better I/O bound, compared to the previously best-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

known EM biconnected component algorithm of Chiang *et al.* under the aforementioned conditions. Although another I/O bound of the form $O(\max\{1, \log\log(|V|DB||E|)\}\cdot(|E|/|V|)\,\mathrm{sort}(|V|))$ for the algorithm of Chiang *et al.* was reported later [34], under the aforementioned conditions, the bound becomes $O(|V|\,\mathrm{sort}(|V|))$ which is still greater than our $O(\mathrm{scan}(|V|^2))$ bound.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7

Conclusions

In this thesis, we have presented external-memory algorithms for solving graph connectivity problems, which include the graph connectivity and biconnectivity problem. Our algorithms are simpler and more I/O efficient than the existing algorithms when $\lceil |V|/M \rceil = \Theta(1)$ and $|E| = \Omega(DB|V|)$, where $G = (V, E)$ is the input graph, M is the internal memory size, B is the block size and D is the number of external disks.

Since every bridge is a biconnected component consisting of a single edge, Algorithm EM_BCC can thus be modified to solve the bridge-connectivity (2-edge connectivity) problem within the same I/O bound.

The following are possible future research:

- Design optimal EM algorithms for the graph connectivity and biconnectivity problem.
- Extend our results to $3-$*vertex connectivity* and $3-$*edge connectivity.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] ABELLO, J., BUCHSBAUM, A. L., AND WESTBROOK, J. R. A functional approach to external graph algorithms. *Algorithmica 32*, 3 (Mar 2002), 437–458.

[2] AGGARWAL, A., AND JEFFREY, S. V. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (1988), 1116–1127.

[3] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The design and analysis of computer algorithms.* Addison-Wesley, (1974).

[4] AHUJA, M., AND ZHU, Y. An efficient distributed algorithm for finding articulation points, bridges, and biconnected components in asynchronous networks. In *Proceedings of the Ninth Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, 1989), Springer-Verlag, 99–108.

[5] ARGE, L., BRODAL, G. S., AND TOMA, L. On external-memory mst, sssp and multi-way planar graph separation. *J. Algorithms 53*, 2 (2004), 186–206.

[6] ARGE, L., TOMA, L., AND ZEH, N. I/O-efficient topological sorting of planar dags. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 2003), ACM Press, 85–93.

[7] ARGE, L., AND VAHRENHOLD, J. I/O-efficient dynamic planar point location. *Comput. Geom. Theory Appl. 29*, 2 (2004), 147–162.

[8] ARGE, L., AND ZEH, N. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science* (Washington, DC, USA, 2003), IEEE Computer Society, 261.

[9] BUCHSBAUM, A. L., GOLDWASSER, M., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. R. On external memory graph traversal. In *SODA '00: Proceedings of*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the eleventh annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA, 2000), Society for Industrial and Applied Mathematics, 859–860.

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

[11] CHAUDHURI, P. An $o(n^2)$ self-stabilizing algorithm for computing bridge-connected components. *Computing 62*, 1 (1999), 55–67.

[12] CHIANG, Y.-J. Dynamic and I/O-efficient algorithms for computational geometry and graph problems: Theoretical and experimental results. Tech. Rep. CS-95-27, (1995).

[13] CHIANG, Y.-J., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., AND VITTER, J. S. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1995), Society for Industrial and Applied Mathematics, 139–149.

[14] CHIN, F. Y., LAM, J., AND CHEN, I.-N. Efficient parallel algorithms for some graph problems. *Commun. ACM 25*, 9 (1982), 659–665.

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

[16] EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. Sparsification-a technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5 (1997), 669–696.

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

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

[19] GALIL, Z., AND ITALIANO, G. F. Maintaining biconnected components of dynamic planar graphs. In *Proc. 18th Int. Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 510* (1991), Springer-Verlag, Berlin, 339–350.

[20] HOHBERG, W. How to find biconnected components in distributed networks. *J. Parallel Distrib. Comput. 9*, 4 (1990), 374–386.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[21] HUTCHINSON, D., MAHESHWARI, A., AND ZEH, N. An external memory data structure for shortest path queries (extended abstract). *Lecture Notes in Computer Science 1627* (1999), 51–60.

[22] IOANNIDIS, Y. E., AND RAMAKRISHNAN, R. Efficient transitive closure algorithms. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1988), Morgan Kaufmann Publishers Inc., 382–394.

[23] JÁJÁ, J. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, (1992).

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

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

[26] KARAATA, M., AND CHAUDHURI, P. A self-stabilizing algorithm for bridge finding. *Distribted Computing 2*, 1 (1999), 47–53.

[27] KAZMIERCZAK, A., AND RADHAKRISHNAN, S. An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanar testing. *IEEE Transactions on Parallel and Distributed Systems 11* (2000), 110–118.

[28] KNUTH, D. E. *The Art of Computer Programming, 2nd Ed. (Addison-Wesley Series in Computer Science and Information.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1978).

[29] KUMAR, V., AND SCHWABE, E. J. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)* (Washington, DC, USA, 1996), IEEE Computer Society, 169.

[30] LAMBERT, O., AND SIBEYN, J. Parallel and external list ranking and connected components on a cluster of workstations. In *Proc. 11th International Conference Parallel and Distributed Computing and Systems of 1999, IASTED* (1999), 454-460.

[31] MANBER, U. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1989).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[32] MEYER, U. External memory bfs on undirected graphs with bounded degree. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2001), Society for Industrial and Applied Mathematics, 87–88.

[33] MEYER, U., SANDERS, P., AND SIBEYN, J. F., Eds. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]* (2003), Lecture Notes in Computer Science 2625, Springer.

[34] MUNAGALA, K., AND RANADE, A. I/O-complexity of graph algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1999), Society for Industrial and Applied Mathematics, 687–694.

[35] NODINE, M., AND VITTER, J. Greed sort: an optimal sorting algorithm for multiple disks. *J. ACM 42*, 4 (1995), 919–933.

[36] PARK, J., TOKURA, N., MASUZAWA, T., AND HAGIHARA, K. Efficient distributed algorithms solving problems about the connectivity of network. *Systems and Computers in Japan 22*, 2 (1991), 1–16.

[37] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994), 17–28.

[38] SWAMINATHAN, B., AND GOLDMAN, K. J. An incremental distributed algorithm for computing biconnected components in dynamic graphs. *Algorithmica 22*, 3 (1998), 305–329.

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

[40] TARJAN, R. E., AND VISHKIN, U. An efficient parallel biconnectivity algorithm. *SIAM J. Comput. 14*, 4 (1985), 862–874.

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

[42] TSIN, Y. H., AND CHIN, F. Y. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Comput. 13*, 3 (1984), 580–599.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[43] TURAU, V. Computing bridges, articulations and 2-connected components in wireless sensor networks. In *Algorithmic Aspects of Wireless Sensor Networks, Second International Workshop ALGOSENSORS 2006, Lecture Notes in Computer Science 4240* (Venice, Italy, 1989), 164-175.

[44] ULLMAN, J. D., AND YANNAKAKIS, M. The input/output complexity of transitive closure. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1990), ACM Press, 44–53.

[45] VITTER, J. S. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys 33*, 2 (2001), 209–271.

[46] VITTER, J. S., AND SHRIVER, E. A. M. Algorithms for parallel memory I: Two-level memories. *Algorithmica 12*, 2-3 (1994), 110–147.

[47] WILSON, R. J. *Introduction to graph theory.* John Wiley & Sons, Inc., New York, NY, USA, (1986).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA AUCTORIS

NAME:	Shan Li
YEAR OF BIRTH:	1976
EDUCATION:	University of Sichuan, Chengdu, China 1993-1997
	University of Windsor, Windsor, Ontario 2005-2007

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.