ZHANGYUN CHEN
A Linear Time Algorithm for Testing a Graph for 3-Edge-Connectivity
(Under the Direction of ROBERT W. ROBINSON)

A graph $G = (V, E)$ is 3-edge-connected if and only if the subgraph $(V, E - S)$ is connected for every set S of at most 2 edges. Two versions of an algorithm to test an arbitrary graph for 3-edge-connectivity due to Nagamochi and Ibaraki were implemented and tested. This required making some modifications to the algorithm as originally described and filling in many details. The two versions implemented are called simplified and accelerated. For a graph with n vertices and m edges the simplified algorithm takes time $O(nm)$, whereas the accelerated algorithm runs in time $O(n + m)$. The simplified algorithm is less complex to code, so the implied constant is smaller for it than for the accelerated algorithm. Testing was done on Feynman diagrams; irreducibility for a Feynman diagram of order n is exactly 3-edge-connectivity for a contracted graph which has n vertices and maximum degree at most 4, hence $m \leq 2n$. No significant difference in speed between the two algorithms was found for $n \leq 150$. For larger n, the accelerated algorithm gradually becomes faster than the simplified one, for example being 10 times faster for n just over 2000.

INDEX WORDS: Edge connectivity, 3-edge-connected graph, Linear algorithm, Quadratic algorithm

A LINEAR TIME ALGORITHM FOR TESTING A GRAPH FOR 3-EDGE-CONNECTIVITY

by

ZHANGYUN CHEN

B.Ed., Beijing University of Physical Education, China, 1990

M.Ed., Beijing University of Physical Education, China, 1997

Ph.D., The University of Georgia, 2001

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2001

A LINEAR TIME ALGORITHM FOR TESTING A GRAPH FOR

3-EDGE-CONNECTIVITY

by

ZHANGYUN CHEN

Approved:	
Major Professor:	Robert W. Robinson
Committee:	Thiab Taha
	Eileen T. Kraemer

Electronic Version Approved:
Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2001

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to the following people:

Dr. Robert W. Robinson who served as my major professor and spent countless hours in discussion with me. He is always willing to help and is available all the time even on late Friday afternoons. Dr. Robinson guides me through my research step by step and always takes his students' learning needs into consideration. Thank you very much for your guidance, encouragement, and patience. Without your help, it would be difficult to finish my thesis. You are truly a mentor.

Dr. Taha and Dr. Kraemer for their guidance and assistance during my graduate study and their willingness to serve on my committee.

My wife, for her love, understanding, encouragement, and support.

My parents and parents-in-law, for their love, support and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Research on connectivity is one of the most important topics in graph theory [3]. In this paper, connectivity refers to edge-connectivity unless explicitly stated otherwise. In the literature of graph theory on determining 3-edge-connectivity, several linear algorithms [9, 12, 17] are available. The current study is based on the work of [12] for computing 3-edge-connected components in a multigraph. The algorithm used by the current study for determining the 3-edge-connectivity of a graph can be divided into two stages. The first stage starts from a depth-first search (DFS), which establishes a spanning tree *T* (or else rejects *G* as not being connected). The DFS also establishes a rank for each vertex, sometimes called the DFS discovery time or index. The algorithm processes the non-tree edges of *G* in order of the minimum of the ranks of its end-vertices. Each non-tree edge is traced back through *T*, and each edge in *T* is assigned a count according to the number of times it is encountered in these traces. At the end of the tracing, if some edge of *T* has a count of 0 or 1, then *G* is not 3-edge-connected. A system of pointers is used to skip over tree edges that already have counts of ≥ 2 so as to achieve linear time. The second phase performs a contraction. Because some cut-pairs of tree edges cannot be detected by the first phase, the contraction is used to simplify the graph. *G* is 3-edge-connected if it consists of a single vertex after contraction. Phases 1 and 2 are applied alternately until 3-edge-connectivity is decided one way or the other.

1.2 BASIC DEFINITIONS

A *graph* $G = (V, E)$ represents an undirected multigraph with a set V of vertices and a set E of edges. An *edge* e is denoted by $e = (u, v)$, where u and v are the end vertices of the edge. If $e = (u, v)$ is an edge of G, we say the vertex u is *adjacent* to a vertex v, and e is *incident* with vertices u and v. We also define $ends(e) = \{u, v\}$ in order to allow for multiple edges. For instance it may be that $e \neq f$ but $\text{ends}(e) = \text{ends}(f)$, so that there is a multiple edge joining u and v. Loops are not allowed, so $u \neq v$ here. The number of edges incident with v is called the *degree* of v. A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called a *subgraph* of $G = (V, E)$. If $V' = V$, G' is called a *spanning* subgraph. For any edge set F, the subgraph $(V, E - F)$ is usually written $G - F$. An edge set of F with $F \subseteq E$ is referred to as a *cut* if it satisfies the following two conditions: (i) $G - F$ has more connected components than G, and (ii) no proper subset of F satisfies (i). A cut F is called a *k-cut* if $|F| = k$. A graph $G = (V, E)$ is said to be *k-edge-connected* if G has no r-cut for $r < k$. Thus, a 3-edge-connected graph is precisely one that is connected and has no 1-cut or 2-cut. A 1-cut is called a *bridge*, and a 2-cut is called a *cut-pair*. An example of a 3-edge-connected graph is illustrated in Figure 1.1. In Figure 1.1, $\{a, b, c\}$ is a 3-cut, while $\{a, b, c, f\}$ disconnects the graph but is not a cut since it is not minimal.

Let Ψ be the contraction mapping which identifies x and y and leaves all other vertices fixed. For $e \in E$, the *contraction* G/e is defined as follows: let $\Psi(u) = \Psi(v) = z$ for some $z \notin V$, and $\Psi(a) = a$ for $a \in V - \{u, v\}$. Then contraction G/e has vertices

$V' = V - \{u, v\} \cup \{z\}$, edges $E' = \{f \in E : \text{ends}(f) \neq \{u, v\}\}$ and $\text{ends}'(f) = \Psi(\text{ends}(f))$ for $f \in E'$. For an edge set E' in E, the graph obtained by contracting all edges in E' is denoted by G/E'. Note that the order of contraction makes no difference. The *local edge-connectivity* $\lambda(x, y; G)$ for $x, y \in V$ with $x \neq y$ is defined to be the minumum number $|F|$ with $F \subseteq E$ so that x and y are disconnected in $G - F$, and $\lambda(x, x; G)$ is defined to be $+\infty$.

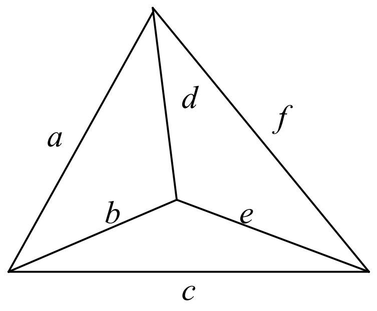

Figure 1.1: An example of a 3-edge-connected graph

Let $\lambda(G)$ denote the *edge-connectivity* of G, which is defined to be k if the graph G is k-edge-connected but not $(k+1)$-edge-connected. If G is the trival graph (just one vertex), then $\lambda(G) = +\infty$. Clearly $\lambda(G) = \min\{\lambda(x, y; G) : x, y \in V(G)\}$ for any graph G. We note another fact about local edge connectivity which will be useful later.

FACT 1.1: For fixed G and k, $\lambda(x, y; G) \geq k$ is an equivalence relation on $V = V(G)$.

This relation is obviously reflexive and symmetric. To see that it is transitive, suppose

$\lambda(x, y; G) \geq k$ and $\lambda(x, z; G) \geq k$. For $r < k$, any r-cut F leaves x connected to y in $G - F$ and y connected to z in $G - F$, hence x is connected to z in $G - F$. So, x and z cannot be disconnected by any r-cut for any $r < k$, hence $\lambda(x, z; G) \geq k$.

A *path* is defined as follows: $P = (v_0, e_1, v_1, \ldots, e_k, v_k)$ is a path if $v_i \in V$ for $0 \leq i \leq k$, $e_i \in E$ for $1 \leq i \leq k$, and ends$(e_i) = \{ v_{i-1}, v_i\}$ for $1 \leq i \leq k$. A path is *simple* if no vertex appears twice in it. Two paths are *edge-disjoint* if their edge sets are disjoint. In general, the number of vertices of a graph G is denoted by n and the number of edges is represented by m.

The input graphs for testing our algorithms are special graphs called *Feynman diagrams*. In a Feynman diagram D, there are two kinds of edges: *V-edges* (undirected) and *G-edges* (directed). Each vertex is incident with exactly 3 edges, *viz*., one *V*-edge, one *G-in-edge* and one *G-out-edge*. Figure 1.2 illustrates an example of a Feynman diagram with order 2. The *order* of a Feynman diagram is the number of *V*-edges in the diagram. The *V*-edges are shown as wavy lines in the figure.

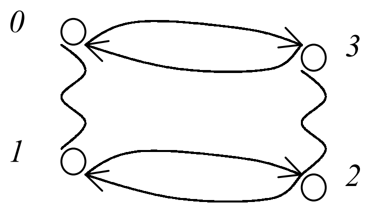

Figure 1.2: An example of a Feynman diagram

Feynman diagrams are used to explore the interacting fermion problem, which is critical in a number of research fields, including strongly correlated electron physics, the electronic structure theory of solids, the theory of nuclear matter, and quantum chemistry [6,10,14]. These diagrams arise from Hubbard modes and the mode Hamiltonian [6,2,19]. Determination of the numbers of all connected diagrams has been studied in both mathematics [18] and physics [4]. Our motivation for investigating algorithms for 3-edge-connectivity is the need for a fast algorithm to determine whether a diagram is irreducible for use in quantum Monte Carlo simulations [16]. A Feynman diagram D is irreducible if it cannot be disconnected by the removal of fewer than 3 G-edges. If D' denotes the graph obtained by contracting the V-edges of D and ignoring the orientation of the G-edges, then clearly D is irreducible if and only if D' is 3-edge-connected. Figure 1.3 is an example of the contraction of Feynman diagram with order 3. Note that if D has order n then D' has n vertices and maximum degree at most 4, hence $m \leq 2n$ edges.

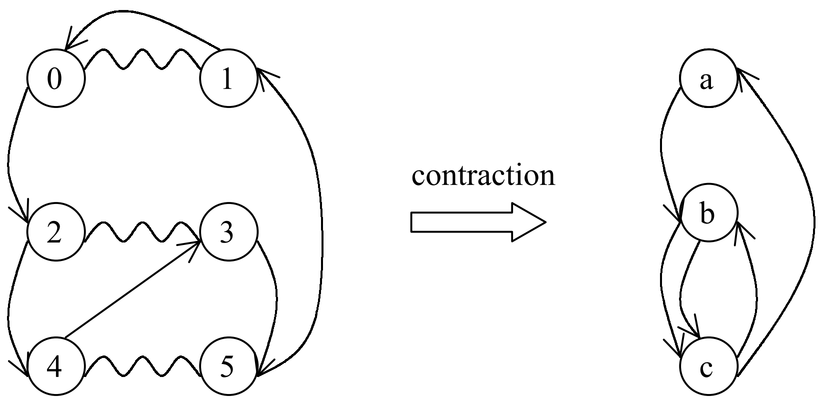

Figure 1.3: An example of the contraction of Feynman diagram with order 3

1.3 DEPTH-FIRST SEARCH

Depth-first search (DFS) visits the vertices of a graph according to the following procedure [1, Section 5.2]. Initially, no vertex has been visited and no edge has been explored. The vertex to start with, say z, is called the *root*. First the root vertex z is visited, and an edge (z, v) incident to z is selected. If v has not been visited, then v is visited, and a new search starts recursively at vertex v. If v has already been visited, then another unexplored edge incident to z will be selected. This process of examining unexplored edges continues until all edges at z have been explored. Since this approach visits unexplored edges recursively as deeply as possible, it is called depth-first search. The edges whose exploration by DFS leads to an unvisited vertex are called *tree edges*, and the remaining edges are called *non-tree edges*.

The following is the DFS in outline form:

DFS(G) makes sequential calls to DFS(G, x) for $x \in V(G)$ which are unvisited, according to some linear ordering of $V(G)$. The function DFS(G,) calls itself recursively.

DFS(G, x):

1. Mark x “visited”.
2. Record the rank of x (denoted here by $r[x]$, this is the discovery time of x in the DFS, sometimes called the DFS index).
3. For each edge e (taken in some order) incident to x, if the other end y (say) is unvisited then e is designated as a tree edge and DFS(G, y) is called.

The following are standard facts about DFS [1, Section 5.2].

FACT 1.2: The tree edges form a *spanning forest* of G consisting of one spanning tree for each connected component of G. Each tree is considered to be the rooted at the vertex of minimum rank.

FACT 1.3: In any sub-tree of a tree in the DFS forest, the ranks of the vertices are always $\geq$ the rank of the root of the sub-tree.

FACT 1.4: Non-tree edges or *back* edges always join an (ancestor, descendent) pair.

1.4 OUTLINE OF THE THESIS

The current thesis investigates simplified and accelerated algorithms for determining 3-edge-connectivity in undirected graphs. Some of the well-known edge-connectivity algorithms [5, 8, 11] are based on the approach that edge-connectivity can be determined by solving max-flow problems. The time complexity for determining k-edge-connectivity is $O(m + k^2 n \log n)$ for $k > 3$ based on [7]. It has been shown that 3-edge-connectivity can be determined in linear time $O(m + n)$ by [9, 12, 13]. The current thesis is motivated by the work of [12]. Two approaches were explored to determine 3-edge-connectivity in graphs. The first one is called the *simplified* algorithm and the second one is termed the *accelerated* algorithm. Timing tests were performed on Feynman diagrams. Note that the V-edge contraction of an order n Feynman diagram D has n vertices and $m \leq 2n$ edges, and it is this contraction which must be checked for 3-edge-connectivity. Generally, there is no significant difference between the two algorithms in timing tests for small orders. However, the accelerated algorithm becomes faster than the simplified starting around $n = 150$. The difference between the two algorithms gradually increases as n increases, with the ratio becoming almost a factor of 10 by $n = 2000$. This bears out the asymptotic time complexity analysis, which shows that for Feynman diagrams of order n the simplified algorithm takes time $O(n^2)$ whereas the accelerated algorithm runs in $O(n)$ time.

The thesis is organized as follows.

Chapter 2 describes a simplified algorithm to detect certain cut-pairs. The algorithm applies DFS to build a set *E1* of tree edges. The set of non-tree edges is called *E2*, and the cut-pairs to be detected will be those having one edge in *E1* and one edge in *E2*. These will be called *type 1* cut-pairs. Then the algorithm randomly selects a non-tree edge and starts from its initial vertex (the higher rank vertex) to traverse the tree edges until its terminal vertex (the lower rank vertex) is encountered. A count field is associated with each tree edge and is used to record the number of times that each tree edge is visited in these path traversals. When all non-tree edges have been processed, we can check if any count for a tree edge is 1 or 0 to determine if there is a cut-pair (of type 1) or a bridge, respectively.

In Chapter 3, we show how to improve the time complexity of the simplified algorithm to linear time by using a pointer for each vertex. The accelerated algorithm selects the non-tree edges from the lowest terminal vertex rank. Each tree edge will be visited at most twice, so the time complexity is linear.

The fourth chapter introduces an algorithm to contract a graph *G* in a way that preserves all 3-components of *G*. Then the same procedure for finding a cut-pair is applied recursively to the contracted graph.

The last chapter compares the performance of the two algorithms for determining 3-edge-connected graphs when applied to Feynman diagrams of various orders. Some conclusions are drawn and possible future work is discussed.

CHAPTER 2

SIMPLIFIED SEARCH AND TRACE ALGORITHM

2.1 FIND A BRIDGE OR A TYPE 1 CUT-PAIR

When DFS is first applied to G the number k of connected components is noted. If $k > 1$ then G is not 3-edge-connected. For Chapters 2 and 3 it will be assumed that G is connected. For a connected graph $G = (V, E)$, we apply depth-first search to G starting from some vertex that we designate as root. Then let $E1$ be the set of edges in the DFS spanning tree of G and $E2$ be the set of non-tree edges, $E – E1$. In order to find a cut-pair with one edge in $E1$ and one edge in $E2$, we assign a count for each edge in $E1$. Let $i(e)$ denote the initial vertex of an edge e (*i.e.*, the edge with higher DFS rank); let $t(e)$ represent the terminal vertex of an edge e (*i.e.*, the edge with lower DFS rank), and let e_v represent the edge in $E1$ with initial vertex v (for $v \neq$ root). Note that $E1 = \{ e_v : v \in V - \{\text{root}\}\}$. Also, let $Pi(e)t(e)$ denote to the unique path in $E1$ starting from the initial vertex $i(e)$ and ending at terminal vertex $t(e)$. Initially, count(e) is set to 0. Each time, we trace a path $Pi(e')t(e')$ in $E1$, for $e' \in E2$, count(e) will be increased by 1 if e lies on $Pi(e')t(e')$. Next, we will show that after all tracing is complete, for any $e \in E1$ count(e) will be 0 if and only if e is a bridge and 1 if and only if e is contained in a cut-pair $\{e, e'\}$ for some $e' \in E2$.

FACT 2.1: e is a bridge in G if and only if $e \in E1$ and count(e) = 0.

Figure 2.1 illustrates this case. Note that for all figures in this thesis, edges in *E1* are shown as solid lines and edges in *E2* are represented by dotted lines. In this case, the count(*e*) is 0 since *e* is the only edge that connectes the component *G1* and *G2*, and

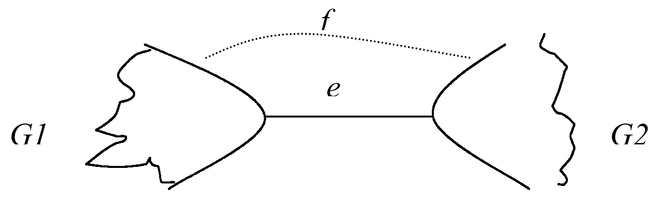

Figure 2.1: *e* is a bridge in *E1*

$G - e$ is disconnected. However, if we assume there is an edge *f* connecting *G1* and *G2*, as Figure 2.1 illustrates, then the graph $G - e$ is still connected, which contradicts our assumption *e* is a bridge. Thus, count(*e*) is 0 if *e* is a bridge. Conversely, *e* is a bridge if we have $e \in E1$ and count(*e*) = 0. For if there is an edge *f* connecting *G1* and *G2* as Figure 2.1 illustrates, then count(*e*) will be at least 1, which contradicts our assumption.

FACT 2.2: for $e \in E1$, count(*e*) = 1 if and only if there is an edge $e' \in E2$ such that $\{e, e'\}$ is a cut-pair for *G*.

Assume $\{e, e'\}$ is a cut-pair with $e \in E1$ and $e' \in E2$ as Figure 2.2 illustrates. In this case, the count(*e*) will be 1. This is because $G - e - e'$ is disconnected into *G1* and *G2*. However, $G - e$ is connected since e' joins *G1* and *G2*. When we trace the path $Pi(e')t(e')$ in *E1*, e' is the only edge of $G - e$ for which the path goes through *e*, i.e., count(*e*) = 1. Conversely, some $\{e, e'\}$ is a cut-pair for *G* if we have $e \in E1$ and count(*e*) = 1. For if *e* is a bridge rather than a cut-pair with e', then count(*e*) will be 0, which contradicts our assumption count(e) = 1. On the other hand, if e' is the edge which contributed to the count of 1 to e then $e' \in E2$ and G - e' contains e as a bridge by Fact 2.1, so $G - \{e, e'\}$ is disconnected. No edge in $E2$ can be a bridge, so $\{e, e'\}$ is a cut-pair for G.

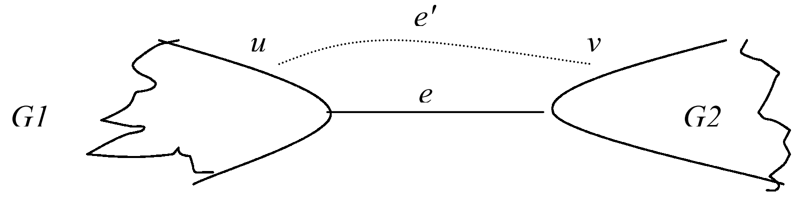

Figure 2.2: e is in a cut-pair in $E1$

Now we are ready to present a simplified algorithm to find cut-pairs with one edge in $E1$. Note that any cut-pair must contain at least one edge from $E1$ since G is connected.

2.2 ALGORITHM DESCRIPTION

The simplified search and trace algorithm computes the counts for edges in $E1$ in a straightforward way. By Fact 2.1 and 2.2, any bridge or cut-pair with an edge in $E2$ will be deleted. The simplified algorithm is based on the work of [12] and is presented in outline form:

Simplified_Search_Trace(G, n):

1. Apply DFS to G and get a partition of $E1$ and $E2$, where $E1$ is the set of tree edges determinted by DFS and $E2$ are non-tree edges. Each vertex $v \in V$ is assigned a rank $r[v]$. If a vertex with degree 1 or 2 exists, then return "not 3-edge-connected".

2. Initialize count(e) to 0 for all e in $E1$.
3. Choose an edge e' from $E2$ and mark it as explored.
4. Trace the path $Pi(e')t(e')$ in $E1$. Start from $v = i(e')$.

   While $r[v] > r[t(e')]$, then increase each count(e_v) in path $Pi(e')t(e')$ by 1 and update v to $t(e_v)$, i.e., $v = t(e_v)$.
5. Repeat 3-4 until all edges of $E2$ have been explored.
6. Check each count(e_v) with $e_v \in E1$. If a count(e_v) = 0 or 1 is found, return "not 3-edge-connected". Otherwise, return "may be 3-edge-connected".

2.3 ANALYSIS OF TIME COMPLEXITY

The correctness of algorithm Simplified_Search_Trace is supported by Section 2.1. The time complexity of step 1 to 5 is $O(|E2||E1|)$ as we trace a path $Pi(e')t(e')$ in $E1$ for each $e' \in E2$. Clearly, the time complexity for step 6 is $O(|V| + |E|)$. Thus, the overall time complexity for this algorithm is $O(|E2||E1|)$. For a connected graph with n vertices and m edges, $|E1| = n - 1 = O(n)$ and $|E2| = m - n + 1 = O(m)$. Therefore, the overall time is $O(nm)$.

CHAPTER 3

ACCELERATED SEARCH AND TRACE ALGORITHM

3.1 ALGORITHM DESCRIPTION

In order to make the time complexity linear, we need to make some changes to steps 4 and 5 of Simplified_Search_Trace(G, n). The objective of the modification is to ensure that if count(e) ≥ 2 is true for an edge $e \in E1$, e will never be visited again. Let E^* denote the set of edges e in $E1$ with count(e) ≥ 2. We maintain a pointer ptr(v) for each vertex v. Initially each ptr(v) points to itself. It will be the case that ptr(v) = v if and only if v is the root or count(e_v) < 2. Otherwise, ptr(v) = ptr($t(e_v)$). (These facts will be proved in Lemma 3.1.) When we trace the path $Pi(e')t(e')$ for a given edge e' in $E2$, the pointers will enable us to skip all edges in E^*. Figure 3.1 is an example illustrating how these edges are skipped. Assume $e1$,$e2$, and $e3$ have already had count(e) = 2 with ptr(u) = v. When we trace up the path Puz, ptr(u) enables us to skip the edges $e1$, $e2$ and $e3$. However, updating pointers must be done with care. When a count(e) with $e \in E1$ becomes 2, all pointers with ptr(v) = $i(e)$ should be assigned as ptr(v) = ptr($t(e_v)$). Assume that in Figure 3.1, count($e4$) is 1 at the current time. When we trace up the path Puz, and count($e4$) changes to 2, then pointers such as ptr($e1$), ptr($e2$), and ptr($e3$) should point to ptr($t(e4)$), i.e., to v'. This means that the subpath in E^* ending at $i(e)$ must merge with the path beginning with $t(e)$. The total time for this operation would be $O(mn)$ if carried out explicitly. But we will show that this merge will be unnecessary if the edges e' in $E2$

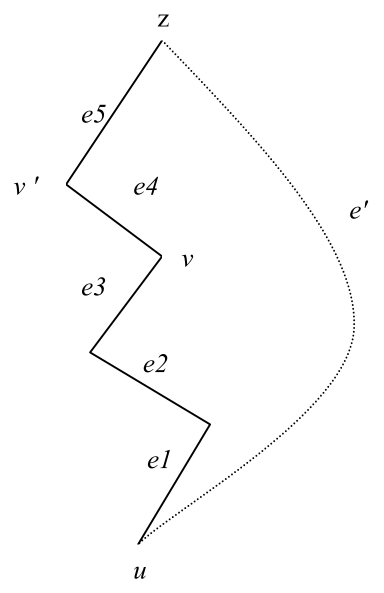

Figure 3.1: An example of skipping edges *e* with count(*e*) ≥ 2

are explored in nondecreasing order of *r*[*t*(*e'*)]. This order is based on the ranks established by DFS, which helps us avoid explict merging and develop a linear time algorithm.

LEMMA 3.1 Let edges *e'* ∈ *E2* in step 2 of Simplified_Search_Trace(*G*, *n*) be selected in nondecreasing order of *r*[*t*(*e'*)]. Then, when a new edge *e* ∈ *E1* is assigned count(*e*) = 2, ptr(*v*) = *i*(*e*) if and only if *v* = *i*(*e*). This also requires that pointers be updated from lower ranks to higher ranks along the path in *E1*.

Proof. Assume there is a ptr(*v*) = *i*(*e*) for some *v* ≠ *i*(*e*) with *e* ∈ *E1*, and count(*e*) will be increased to count(*e*) ≥ 2 by tracing path *Pi*(*e'*)*t*(*e'*) for *e'* ∈ *E2*. Since ptr(*v*) = *i*(*e*) means that *Pvi*(*e*) is in *E** and *e'* is selected in nondecreasing order of *r*[*t*(*e'*)], we thus can conclude that count(*e*) was already 2 before starting to trace the path *Pi*(*e'*)*t*(*e'*). This is a contraction, which proves the lemma.

Figure 3.2 is an illustration for Lemma 3.1. Pointers are indicated by bold solid directed edges in the figure.

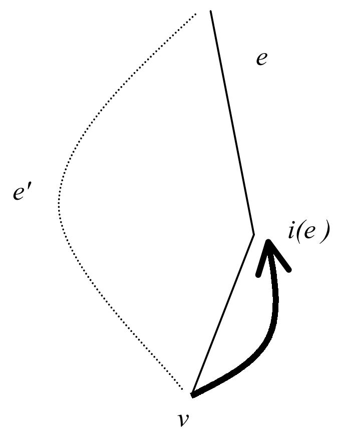

Figure 3.2: Diagram to illustrate Lemma 3.1

Now the accelerated algorithm for determining 3-edge-connected graph can be presented. The accelerated algorithm is based on the work of [12]. However, in that paper the updating of pointers starts from the highest ranked vertex and proceeds to the lowest ranked vertex. In general this does not skip all edges with count(*e*) ≥ 2. We solved this problem by using a path stack. The pointers are updated in reverse order, following the path $Pt(e')i(e')$ from the lowest ranked vertex to the highest ranked vertex. Also, we use the cut_count variable to record the number of cut-pairs and the bridge_count variable to record the number of edges with count 0 during the trace phase and thus do not need to go through all values of count(*e*) again at the end. The following is the accelerated algorithm in outline form:

Accelerated_Search_Trace(G, n):

1. edges. Each vertex $v \in V$ is assigned its DFS discovery time as its rank, denoted $r[v]$. If there is a vertex with degree of 1 or 2, then return "not 3-edge-connected".

2. Initialize count(e) to 0 for all $e \in E1$, cut_count to 0 and bridge_count to n -1. Also set ptr(v) = v for all vertices v and designate all edges in $E2$ as unexplored.
3. Initialize an empty path stack.
4. Choose an unexplored edge e' from $E2$ with the smallest $r[t(e')]$.
5. Trace the path $Pi(e')t(e')$ in $E1$. Start from v = ptr($i(e')$). While $r[v] > r[t(e')]$, increment count(e_v) and cut_count and decrement bridge_count if count(e_v) is equal to 0; otherwise, if count(e_v) was already 1, then push v onto the path stack and set v = ptr($t(e_v)$).
6. After tracing the path $Pi(e')t(e')$ in $E1$, while the path stack is non-empty:
   (i) pop the top vertex v and update ptr(v), *i.e.*, ptr(v) = ptr($t(e_v)$);
   (ii) update count(e_v) to 2, and decrease cut_count;
7. Repeate 3-6 until all edges of $E2$ have been explored.
8. If bridge_count = 0 and cut_count = 0, return "may be 3-edge-connected". Otherwise, return "not 3-edge-connected".

To demonstrate how to update the pointers, we use Figure 3.3 as an example. Assume $r[t(e')]$ is the lowest rank associated with an unexplored edge e', $i(e') = u$, $t(e') = v'$, and at this point count($e1$), count($e2$) and count($e3$) are 1. When we start from u to trace the path Puv', u is pushed onto the stack since count($e1$) is already 1,. Similarly we push u' and then v onto the stack. After tracing the path Puv', it is time to update the pointers. Since the path stack is non-empty, we pop the top vertex in the stack and update its pointer. First, v is popped and ptr(v) is updated as ptr(v) = v'. Then u' is popped and ptr(u') is assigned the ptr(v), *i.e.*, ptr(u') = v'. Last, u is popped and assigned ptr(u'), which is ptr(*u*) = *v′*. Now ptr(*u*), ptr(*u′*) and ptr(*v*) all point to the vertex *v′* as Figure 3.3 illustrates. Pointers are indicated by bold solid directed edges in the figure.

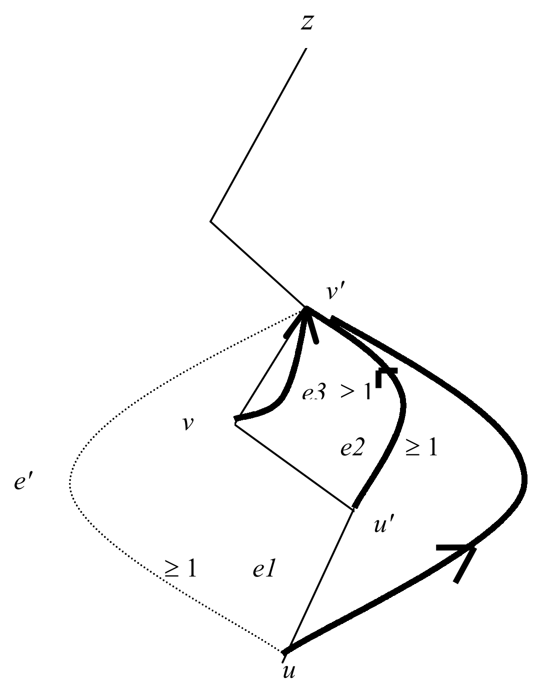

Figure 3.3: An example of updating pointers

3.2 ANALYSIS OF TIME COMPLEXITY

The correctness of Algorithm Accelerated_Search_Trace is clearly supported by Lemma 3.1. In the lemma, it is shown that either count(e_v) < 2 or pointer(v) = z for a path Pvz in *E1* which contains *e*. This implies that each edge in *E1* is traced no more than three times during the Accelerated_Search_Trace execution. The total time for steps 4 and 5 is $O(|E| + |E1|) = O(m + n)$. Selecting edges *e′* in nondecreasing order of $r[t(e')]$ from *E2* is also realized in time $O(m + n)$ by applying the depth-first search rank recorded during the process of partitationing *E* into *E1* and *E2*. The other computations are minor.

Note that Search_Trace (either algorithm) can detect only those cut-pairs with one edge in *E1* and the other in *E2*. Cut-pairs having both edges in *E1* (*type 2* cut-pairs) will not be identified by a Search_Trace algorithm. In order to find such cut-pairs, we need to contract *G* into a smaller graph and then employ Search_Trace and contraction alternately. In the next section, a contraction operation is presented and proved to preseve all 3-components of a graph.

CHAPTER 4

GRAPH CONTRACTION

4.1 CONTRACTION

Recall that contraction was defined in 1.2. The contraction phase of our algorithm will always follow an application of Search_Trace, and will replace G by $G/E2$. We assume throughout Chapter 4 that Search_Trace(G, n) returned "may be 3-edge-connected". Thus, G is connected, and contains no bridges and no cut-pairs with an edge in $E2$. The next two lemmas will show that G is 3-edge-connected if and only if $G/E2$ is 3-edge-connected.

LEMMA 4.1 If $e = (x, y) \in E2$, then $\lambda(x, y; G) \geq 3$.

Proof. If not, then $\{e, e'\}$ is a cut-pair for some $e' \in E1$, contrary to the hypothesis.

LEMMA 4.2 If $e \in E2$, then G/e is 3-edge-connected if and only if G is 3-edge-connected.

Proof. Let Ψ be the contraction mapping that identifies x and y and leaves all other vertices fixed. Since $\lambda(x, y; G) \geq 3$ by Lemma 4.1 and $\lambda(u, v; G) \geq 3$ is an equivalence relation on the vertices of G as noted in Section 1.2, it follows that for all u and v, $\lambda(u, v; G) \geq 3$ if and only if $\lambda(\Psi(u), \Psi(v); G/e) \geq 3$. Hence G is 3-edge-connected if and only if G/e is 3-edge-connected.

Now let $G' = G/E2$. After contraction, $G/E2$ is still 3-edge-connected if and only if G is 3-edge-connected by Lemma 4.2. Moreover, all edges remaining in G' are from $E1$. The next section describes the algorithm. First G is contracted to G', and then a Search_Trace algorithm is applied to G'. Contraction and Search_Trace are called alternately until it is decided whether or not G is 3-edge-connected.

4.2 ALGORITHM DESCRIPTION

The algorithm presented here combines the contraction and Search_Trace algorithms to decide 3-edge-connectivity. The following is the algorithm in outline form:

Determination (G, n):

1. Return "3-edge-connected" if G contains just one vertex;
2. Apply either the Simplified_Search_Trace or the Accelerated_Search_Trace algorithm to G. If "not 3-edge-connected" is returned, this is passed on as the output of Determination (G, n) and the algorithm ends. Otherwise, go to step 3.
3. Apply contraction to obtain a new graph $G' = G/E2$, then replace G by G' and return to step 1.

In [12], the idea of preserving the 3-components of G by contraction is presented, but how to implement contraction so that the entire algorithm is linear is not presented. Actually, this is a very important step in the overall algorithm. If this step is not linear, then the entire algorithm will not be linear. In order to make contraction linear, the approach we used is based on applying depth-first search to $G - E1$. Following the outline of DFS given in Section 1.3, DFS($G - E1$) makes sequential calls to DFS($G - E1$, x) and all vertices discovered during the ith such call are assigned the component index $i - 1$. At the end, a single pass through G suffices for the contraction of G' using these component labels for G - $E1$.

4.3 ANALYSIS OF TIME COMPLEXITY

The correctness of the algorithm is supported by the discussion in Section 4.1. Now we consider its time complexity. Clearly, labeling the new vertices in the contraction step is done in time $O(|E2|)$ and going through edges in $E1$ and $E2$ is done in $O(|E1| + |E2|) = O(|E| + |V|)$ if the time for recursive calls of Determination (G, n) does not count. The running time of Accelerated_Search_Trace is $O(|E| + |V|)$. Thus, the overall time complexity for a single pass through steps 1, 2 and 3 of Determination (G, n) when using Accelerated_Search_Trace is still completed in time $O(|E| + |V|) = O(n + m)$.

Let G_1, G_2, G_3, … be the orginal graph, its contraction, etc. If the contraction of G_i to G_{i+1} is actually performed, then the minmum degree of every vertex in G_i is at least 3. Thus, if G_i has n_i vertices and m_i edges we have $2\ m_i \geq 3\ n_i$, while $m_{i+1} \leq n_i - 1$, so $m_{i+1} \leq (2/3)\ m_i$. Since $m_1 = m$, the total number of edges considered in the iterations of Search_Trace and contraction is at most $m + (2/3)\ m + (2/3)^2\ m + \ldots = 3\ m$.

From this we see that the total number of vertices considered in the entire course of the algorithm is at most $n + 2m$, since $n_1 = n$ and $n_i \leq (2/3)\ m_i$ if G_i is actually contracted to G_{i+1}. Finally, the number of iterations performed by Determination (G, n) is $O(\log m)$ since no contraction can occur if $m_i = 1$. Thus the entire process takes time $O(n + m)$ for the accelerated algorithm. The analysis for the simplified algorithm is similar, replacing $O(n + m)$ by $O(nm)$ in the trace step. This leads to $O(nm)$ time for the simplified algorithm.

It is easier to see that the space required by either the simplified or the accelerated algorithm is $O(n + m)$. For there is no need to retain G_i once G_{i+1} has been calculated.

CHAPTER 5

IMPLEMENTATION AND TESTING RESULTS

5.1 IMPLEMENTATION

Simplified_Search_Trace and Accelerated_Search_Trace were implemented separately so that their efficiencies could be compared. For testing purposes, the lower order Feynman diagrams are created by using an existing program *irrn.cc* by Robinson [15]. The higher order Feynman diagrams were produced by using a random Feynman diagram generator program. Even if we start with a Feynman diagram, the contractions produce a series of graphs that must be treated as arbitrary multigraphs. In our implementation, in order to represent an arbitrary multigraph we dynamically allocate memory space for each vertex based on the information obtained by the DFS in the first execution of step 3 of Determination *(G, n)*. Every new graph is presented as an array of adjacency lists rather than an $n \times n$ matrix because an $n \times n$ array would need to allocate contiguous space in main memory for $n \times n$ elements, which is $\Theta(n^2)$ space. In our tests, this led to memory allocation failures for n around 5000. However, with the array of adjacency lists, the space allocation is $\Theta(n+m)$, which is $\Theta(n)$ for Feynman diagrams. In our tests, this works for high order (*e.g*, n = 37000) before running out of memory space. To illustrate, a possible multigraph obtained by contraction is shown in Figure 5.1, and its representation as an array of adjacency lists is illustrated in Figure 5.2. Note that our representation is symmetric, in that an edge {*u*, *v*} will be indicated by an occurrence of *v* in *u*'s adjacency list and *vice versa*.

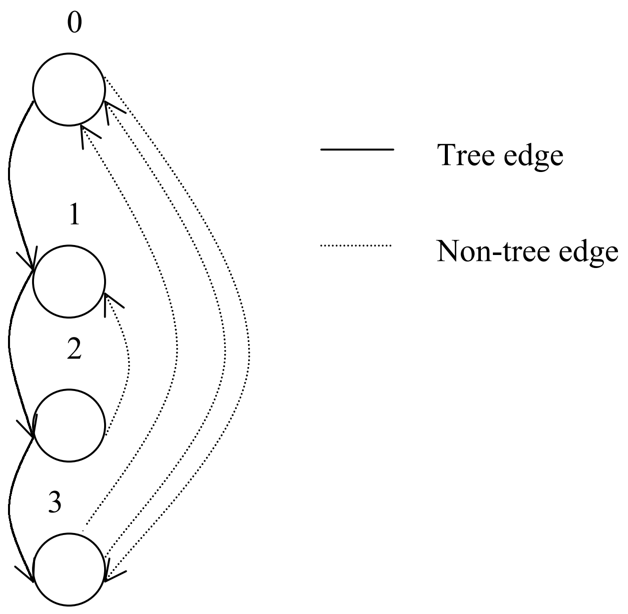

Figure 5.1: A new multiple graph after contraction

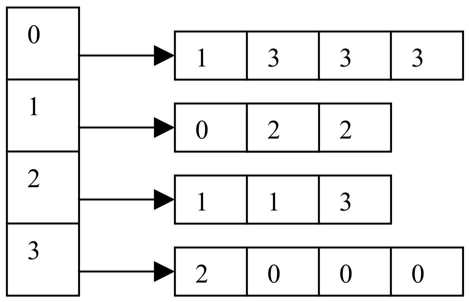

Figure 5.2: A presentation of *G* in adjacency list structure

5.2 COMPARISON OF THE TWO ALGORITHMS

The difference between the simplified algorithm and the accelerated algorithm is that the first approach continues to increase each count(*e*) for *e* in *E1* and potentially increments count(*e*) once for every edge in *E2* to determine if the graph is 3-edge-connected graph; however, the second approach visits each edge *e* in *E1* at most twice by using the pointer ptr(*e*). Once an edge in *E1* has been visited twice, it will never be visited again. There is some extra overhead (*e.g.*, pointers and pointer updates) for the accelerated algorithm compared to the simplified algorithm. Thus for lower order *n* the accelerated algorithm may not show benefits or run faster than the simplified algorithm. We expect to find a threshold value for *n*, beyond which the accelerated algorithm will show advantages and run faster compared to the simplified algorithm.

To test this hypothesis, two series of Feynman diagrams were created using different programs. The first timing tests were conducted by using lower order Feynman diagrams. Our program created all the different connected Feynman diagrams with given order up to 8, testing each with the simplified algorithm or the accelerated algorithm. The data in Table 5.1 show the average testing time for one graph by using the two algorithms with orders ranging from 2 to 8. For the average testing time, the data in Table 5.1 indicate that the simplified algorithm is close to the accelerated algorithm except at order 4. This supports our hypothesis that the accelerated algorithm may not show any advantages compared to simplified algorithm when testing graphs with lower order. To find a possible threshold, we performed a series of timing experiments with random Feynman diagrams. The results of these timing experiments (Table 5.2) indicated that there was no significant difference between the two algorithms when the order *n* was less than 150. When the order was gradually increased for *n* > 150, the difference in the times gradually increased as well. As the order *n* reached 2000, the accelerated algorithm was almost 10 times faster than the simplified algorithm.

Table 5.1: The average testing times in seconds *per* diagram for the simplified and the accelerated algorithms

order	number of diagrams	creation time	average time for accelerated	average time for simplified
2	2	3.50E-06	1.74E-05	1.80E-05
3	10	2.70E-05	2.48E-05	2.53E-05
4	98	2.36E-04	3.00E-05	1.62E-06
5	1138	2.58E-03	3.52E-05	3.62E-05
6	15338	3.80E-02	4.35E-05	4.21E-05
7	236122	2.40E-01	2.53E-05	2.48E-05
8	4093058	3.98	3.05E-05	2.95E-05

For high order *n*, the total number of diagrams would be hundreds of billions, and we could not test all of them with a general workstation. Thus, we wrote another program that creates high order random Feynman diagrams for the timing tests. Table 5.3 shows the data for average testing time for both accelerated and simplified algorithms. The data show that the average testing time for the accelerated algorithm increases gradually as the order increases. However, the simplified algorithm is totally different. The average

Table 5.2: The average time and ratio of two algorithms with orders between 10 and 950

order	Average testing time for accelerated	Accelerated ratio*	Average testing time for simplified	Simplified ratio*
10	9.80E-06	9.80E-07	1.02E-05	1.02E-06
20	1.34E-05	6.70E-07	1.56E-05	7.80E-07
30	3.41E-05	1.14E-06	3.60E-05	1.20E-06
40	4.76E-05	1.19E-06	5.23E-05	1.31E-06
50	5.36E-05	1.07E-06	6.29E-05	1.26E-06
60	6.76E-05	1.13E-06	8.20E-05	1.37E-06
70	7.92E-05	1.13E-06	1.01E-04	1.44E-06
80	9.76E-05	1.22E-06	1.24E-04	1.55E-06
90	1.08E-04	1.20E-06	1.44E-04	1.60E-06
100	1.16E-04	1.16E-06	1.59E-04	1.59E-06
150	1.79E-04	1.19E-06	2.76E-04	1.84E-06
200	2.61E-04	1.31E-06	4.49E-04	2.25E-06
250	3.15E-04	1.26E-06	5.83E-04	2.33E-06
300	3.87E-04	1.29E-06	7.30E-04	2.43E-06
350	4.36E-04	1.25E-06	9.88E-04	2.82E-06
400	4.71E-04	1.18E-06	1.19E-03	2.97E-06
450	6.27E-04	1.39E-06	1.58E-03	3.51E-06
500	6.58E-04	1.32E-06	1.76E-03	3.51E-06
550	7.75E-04	1.41E-06	2.41E-03	4.38E-06
600	8.57E-04	1.43E-06	3.08E-03	5.13E-06
650	9.50E-04	1.46E-06	3.79E-03	5.83E-06
700	9.24E-04	1.32E-06	4.31E-03	6.16E-06
750	1.04E-03	1.38E-06	5.15E-03	6.86E-06
800	1.28E-03	1.60E-06	6.36E-03	7.95E-06
850	1.50E-03	1.76E-06	7.38E-03	8.68E-06
900	1.35E-03	1.50E-06	6.97E-03	7.74E-06
950	1.24E-03	1.31E-06	7.56E-03	7.96E-06

* Note: ratio = (actual testing time / # of graphs) / order.

Table 5.3: The average testing time and the ratio of two algorithms for different orders

Order	Average testing time for accelerated	Accelerated ratio*	Average testing time for simplified	Simplified ratio*
1000	1.42E-03	1.42E-06	8.47E-03	8.47E-06
2000	2.93E-03	1.46E-06	3.81E-02	1.90E-05
3000	5.08E-03	1.69E-06	9.80E-02	3.27E-05
4000	6.50E-03	1.62E-06	1.55E-01	3.87E-05
5000	1.84E-2	3.68E-06	2.38E-01	4.77E-05
6000	1.36E-2	2.27E-06	2.05E-01	3.42E-05
7000	2.16E-2	3.08E-06	4.08E-01	5.83E-05
8000	1.82E-2	2.27E-06	4.55E-01	5.69E-05
9000	1.92E-2	2.13E-06	8.28E-01	9.20E-05
10000	3.23E-2	3.23 E-06	1.75	1.75 E-04
11000	3.43E-2	3.12E-06	1.98	1.81 E-04
12000	2.45E-2	2.04E-06	1.59	1.32 E-04
13000	2.75E-2	2.11E-06	1.99	1.53 E-04
14000	3.54E-2	2.53E-06	2.71	1.94 E-04
15000	5.06E-2	3.37E-06	4.02	2.68 E-04
16000	4.14E-2	2.58E-06	3.70	2.31 E-04
17000	5.18E-2	3.04E-06	5.01	2.94 E-04

* Note: ratio = (actual testing time / # of graphs) / order.

testing time starts from 8.47E-03 sec with order 1000 and dramatically increases as the order increases.

By comparing the ratio of the two algorithms in Figure 5.1, a clear trend was observed. The ratio for the accelerated algorithm stays essentially constant when the order increases. In contrast to the accelerated algorithm, the ratio for the simplified algorithm keeps increasing when the order increases. Figure 5.3 perfectly demonstrates the advantage of the accelerated algorithm compared to the simplified algorithm when high order graphs are used. Thus, our asymptotic analysis of the two algorithms is clearly supported by the timing test results. Note that in the figure the number shown on the horizontal scale represents the number of vertices in units of one thousand.

5.3 CONCLUSIONS

In this thesis, two algorithms were explored to determine if a graph is 3-edge-connected. The simplified algorithm is simpler and easier to implement compared to the accelerated algorithm. Both algorithms give correct results. When the order of the graph is lower, the simplified algorithm works almost as well as the accelerated algorithm and even slightly faster sometimes because there is some extra overhead for the accelerated algorithm. The experimental timing results clearly demonstrate that there is a threshold at which the accelerated algorithm starts showing its advantage. The data from timing tests with high order graphs indicate that the accelerated algorithm is much faster than the simplified algorithm when the order is 2000 or more.

One direction for future work would be to look for a way to avoid contractions of the graph while still retaining the linear time performance of the accelerated algorithm.

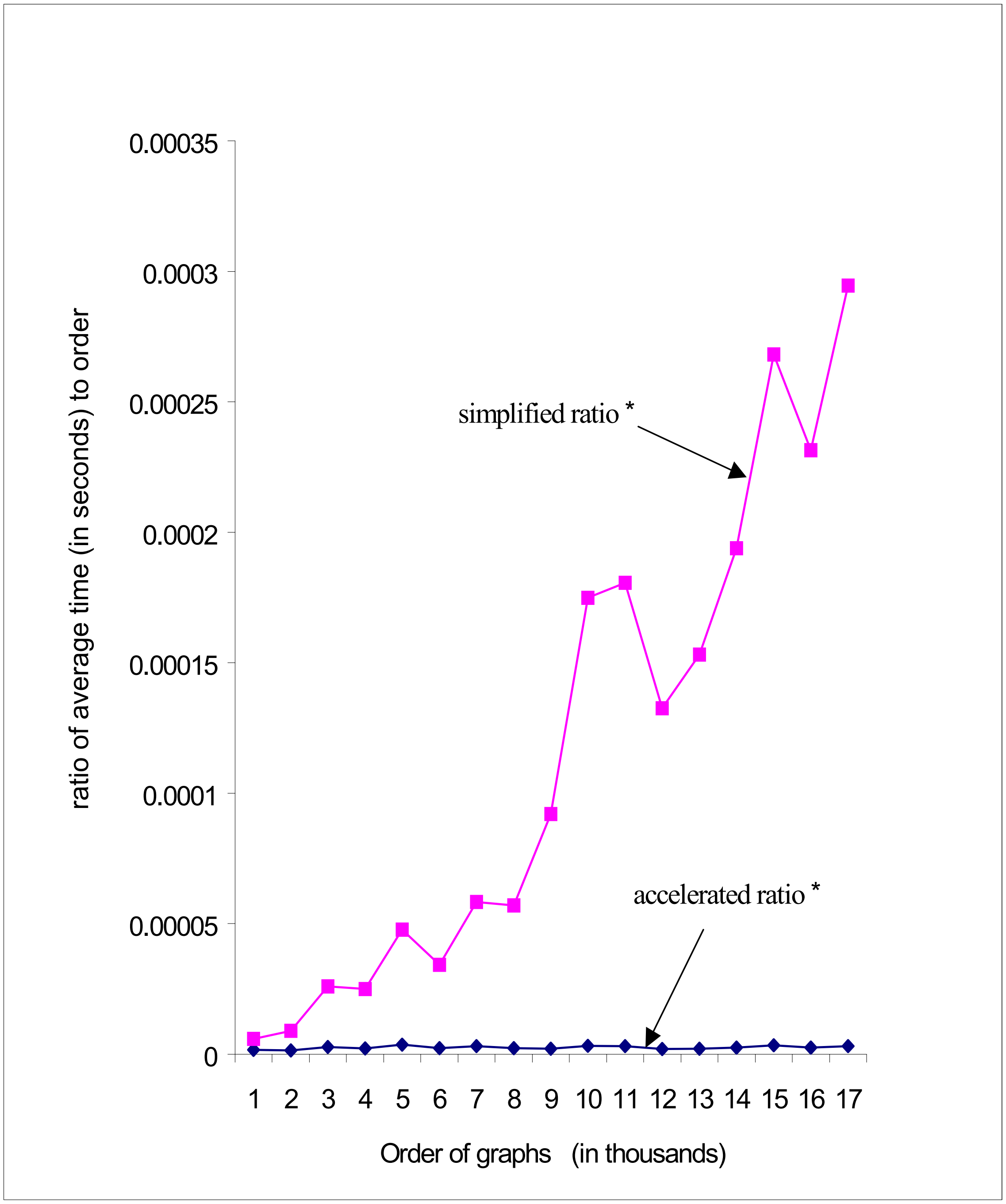

Figure 5.3: The comparison of ratio for both accelerated and simplified algorithms

* Note: ratio = (actual testing time / # of graphs) / order.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer *Algorithms*, Addison-Wesley, Redwood City, CA, 1974.

[2] P.W. Anderson, Resonating valence bond state in La_2CuO_4 and superconductivity, *Science* **235** (1987), 1196-1198.

[3] C. Berge, *Graphs*, 2nd revised ed., North-Holland, New York, 1985.

[4] P. Cvitanovic, B. Lautrup and R.B. Pearson, Number of weights of Feynman diagrams, *Phys. Rev. D* **18** (1978), 1939-1949.

[5] A.H. Esfahanian and S.L. Hakimi, On computing the connectivities of graphs and digraphs, *Networks* **14** (1984), 355-366.

[6] P. Fulde, *Electron Correlations in Molecules and Solids*, 3rd ed., Spring Ser. Solid-State Sci., Springer, Berlin, 1995.

[7] H.N. Gabow, A matroid approach to finding edge connectivity and packing arborescences*, Proc. 23rd ACM Symp. On Theory of Computing* (1991), 112-122.

[8] Z, Galil, Finding the vertex connectivity of a graph*, SIAM J. Comput*. **9** (1980), 197-199.

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

[10] G.D. Mahan, *Many-Particle Physics*, 2nd ed., Plenum Press, New York, 1990.

[11] D.W.Matula, Determining edge connectivity in O(nm), *Proc. 28th IEEE Symp. Found. Compt. Sci.* (1987), 249-251.

[12] H. Nagamochi and T. Ibaraki, A linear time algorithm for computing

3-edge-connected components in a multigraph, *Japan J. Indust. Appl. Math.* **9** (1992), 163-180,

[13] H. Nagamochi and T.Ibaraki, A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph, *Algorithmica* (1992), 583-596.

[14] J.W. Negele and H. Orland, *Quantum Many-Particle Systems*, Addison-Wesley, Redwood City, CA, 1988.

[15] R.W. Robinson, Program for generating connected Feynman diagrams, http://www.uga.edu/~rwr/ITR_PROJ/rwr_timing/irrn.cc

[16] H.B. Schüttler, J.N. Corcoran, D.K. Lowenthal and R.W. Robinson. *Stochastic summation of high-order Feynman graph expansions*. NSF Proposal Number: 0081789 (2000).

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

[18] J. Touchard, Sur un problème de configurations et sur les fractions continues, *Canad. J. Math.* **4** (1952), 2-25.

[19] F.-C. Zhang and T.M. Rice, Effective Hamiltonian for the superconducting Cu oxides, *Phys. Rev. B* **37** (1988), 3759-3761.