A NEW LINEAR ALGORITHM FOR CHECKING A GRAPH FOR 3-EDGE-CONNECTIVITY

by

Feng Sun

(Under the direction of Robert W. Robinson)

Abstract

A new algorithm to test an arbitrary graph for 3-edge-connectivity is proposed, implemented and tested. It is a modification of the classic linear algorithm of Hopcroft and Tarjan for dividing a graph into 3-connected components. The algorithm uses three depth-first searches to locate separation pairs. It runs in time $O(m + n)$, where m is the number of edges and n is the number of vertices in the graph. Testing was done on simple graphs and Feynman diagrams. The results show good agreement with the time complexity analysis, validating the algorithm design and implementation.

Index words: edge connectivity, 3-edge-connected graph, linear algorithm, depth-first search

A NEW LINEAR ALGORITHM FOR CHECKING A GRAPH FOR 3-EDGE-CONNECTIVITY

by

Feng Sun

B.S., Peking University, China, 1999

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

2003

A NEW LINEAR ALGORITHM FOR CHECKING A GRAPH FOR 3-EDGE-CONNECTIVITY

by

Feng Sun

Dedication

In memory of Lei Yu, Lin Liqing, Lu Zhen, Yang Lei, Zhang Xingbai and Zhou Huixia

For their courage to explore unknown territory

For their love of mountaineering

Acknowledgements

I am extremely thankful to my major professor, Dr. Robert W. Robinson for his invaluable guidance and encouragement throughout my entire research and academic years. Without his help, it would be difficult to finish this thesis.

I would also like to thank Dr. E. Rodney Canfield and Dr. Eileen T. Kraemer for serving on my advisory committee, and for their advice and guidance.

My sincere gratitude is extended to my wonderful and loving parents, who have been very supportive and encouraging.

Last but certainly not least, I wish to express my sincere thanks to my wife for her love and support.

Table of Contents

Page

Acknowledgements v

List of Figures viii

List of Tables ix

Chapter

1 Introduction 1

1.1 Overview 1

1.2 Basic Definitions 3

1.3 Depth-first search 5

1.4 Identifying separation pairs 6

1.5 Generating unlabeled graphs 8

2 A linear algorithm for checking 3-edge-connectivity . . . 10

2.1 Preliminaries 10

2.2 Analysis of type 1 separation pairs 12

2.3 Analysis of type 2 separation pairs 14

2.4 Time complexity analysis 23

3 Implementation and test results 24

3.1 Testing on general graphs 24

3.2 A simplified algorithm for Feynman diagrams . . . 27

4 Conclusions and future work 32

Bibliography 34

List of Figures

1.1 An example of a 3-edge-connected graph. 2

1.2 A partition of vertices V into disjoint sets V_1 and V_2. 7

1.3 Examples of separation pairs. 8

2.1 A palm tree before and after the second DFS. 17

3.1 Comparison of running times and scaled times for 3-edge-connectivity algorithms. 28

List of Tables

3.1 Numbers of unlabeled and labeled 3-edge-connected graphs by order n. 25

3.2 Numbers of unlabeled 3-edge-connected graphs by order n and size m. 26

3.3 Average testing times and scaled times for two algorithms on random graphs. 27

3.4 Comparison of running times for our linear algorithm and a quadratic algorithm. 31

Chapter 1

Introduction

1.1 Overview

Designing efficient algorithms for determining the connectivity of graphs has been a subject of great interest during the last two decades [5]. Many algorithms for the computation of edge-connectivity and vertex-connectivity have been developed. In the literature, several algorithms [2, 6, 12, 18] for determining 3-edge-connectivity are described. However none of these is based on the classic linear algorithm of Hopcroft and Tarjan for dividing a graph into 3-connected components [9].

In this thesis we develop a linear algorithm for deciding the 3-edge-connectivity of a simple graph by modifying the algorithm of Hopcroft and Tarjan. Their terminology and approach are followed as closely as possible. Nothing essential is lost in restricting the algorithm to simple graphs. Loops have no effect on edge-connectivity. Multiple edges do affect edge-connectivity but they can be easily incorporated into our algorithm. The presentation, however, is significantly simpler when multiple edges are not allowed.

Like the algorithm of Hopcroft and Tarjan, the linear algorithm described in this paper is based on depth-first search (DFS). There are two types of separation pairs for 3-edge-connectivity, and these are located using three DFSs. The first DFS establishes a palm tree, in which edges are partitioned into a tree edge set and a frond set. The DFS also computes several important values for each vertex, including $NUMBER$, $LOWPT1$ and $LOWPT2$. Type 1 separation pairs are checked in the first step based on these values. The next procedure reorders the adjacency list for each vertex so that a child vertex with lower $LOWPT1$ appears before a child vertex with higher $LOWPT1$ in the list. Then the second DFS is performed to generate a set of paths and renumber the vertices from n to 1 in the order they are last visited using the new adjacency structure. The values of $LOWPT1$ and $LOWPT2$ are updated based on the new number for each vertex. The third DFS is used to identify type 2 separation pairs. In this procedure, possible separation pairs are kept in a stack called PStack. The stack is updated when a new vertex is visited. If a candidate pair is determined to be a separation pair, the algorithm returns 0 and stops immediately. At the end, the algorithm returns 1 if no separation pair is found.

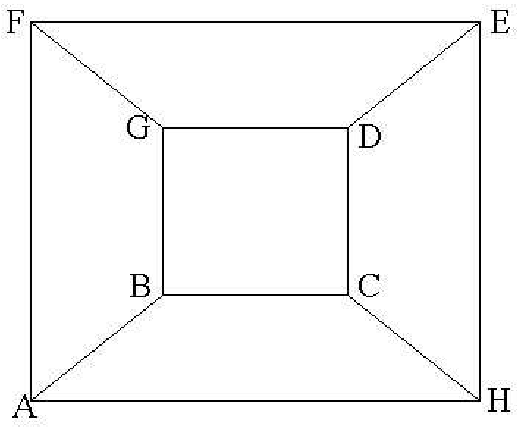

Figure 1.1: An example of a 3-edge-connected graph.

1.2 Basic Definitions

In order to describe graph algorithms, we must give some definitions. Most of these are more or less standard in the literature. (See [1] and [19], for instance.) However the terminology and notation associated with palm trees are from [9].

graph A *graph* (or *simple graph*, for emphasis) G is an ordered pair of disjoint sets (V, E) such that $V \neq \phi$ and E is a subset of the set $V^{(2)}$ of unordered pairs from V. We consider only finite graphs, so V and E are always finite. Here V is the set of *vertices* and E is the set of *edges*. If G is a graph, then $V = V(G)$ is the vertex set of G, and $E = E(G)$ is the edge set. The *order* of G is $|V|$. The *size* of G is $|E|$. If E is a multiset, that is , if edges are allowed to occur more than once, then G is a *multigraph*. If the edges are ordered pairs of vertices, then the structure is a *directed graph*, or *digraph*.

subgraph A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. In this case, we write $G' \subseteq G$.

path A path $p : v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path $p : v \overset{*}{\Rightarrow} v$ is a *cycle* if it contains at least one edge, all its edges are distinct, and the only vertex to occur twice on p is v, which occurs exactly twice (at the beginning and the end).

connected graph We say a graph G is *connected* if and only if every pair of distinct vertices in G is joined by a path.

trivial graph G is *trivial* if and only if it has order 1 and no edges.

component A *component* of a graph G is a maximal connected subgraph of G.

edge-separator An *edge-separator* of a graph is a minimal set $S \subseteq E(G)$ such that $G - S$ has more than one component.

separation pair An edge-separator is a *separation pair* if it contains two edges.

k-edge-connected graph A graph is *k-edge-connected* if every edge-separator has at least k edges.

edge connectivity The *edge connectivity* of a graph G of order $n \geq 2$, written $\lambda(G)$, is the minimum cardinality of an edge-separator.

isomorphic graphs Two graphs G_1 and G_2 are *isomorphic* if there is a one-to-one function ϕ from $V(G_1)$ onto $V(G_2)$ such that $uv \in E(G_1)$ if and only if $\phi(u)\phi(v) \in E(G_2)$.

labeled and unlabeled graph In a graph of order n, if the integers from 1 through n are assigned to its vertices, then it is *labeled.* An *unlabeled graph* is an isomorphism class of labeled graphs.

tree A graph containing no cycle is *acyclic.* A *tree* is an acyclic connected graph.

rooted tree A *rooted tree* is a tree with a special vertex, called the *root.*

directed rooted tree A *directed rooted tree* is a rooted tree with edges directed away from the root (also called an *out-tree* in the literature). The notation $v \to w$ indicates that (v, w) is a directed edge of the directed rooted tree T. The relation "there is a path from v to w in T" is denoted by $v \overset{*}{\to} w$. If $v \to w$, v is the *father* of w and w is a *son* of v. If $v \overset{*}{\to} w$, v is an *ancestor* of w and w is a *descendant* of v. The set of descendants of a vertex v is denoted by $D(v)$. Every vertex is an ancestor and a descendant of itself.

spanning tree A tree T is said to be a *spanning tree* of a graph G if T is a subgraph of G and T contains all the vertices of G.

palm tree Let P be a directed graph consisting of two disjoint sets of edges denoted by $v \to w$ and $v - \to w$. Suppose P satisfies the following properties:

1. The subgraph T containing the edges $v \rightarrow w$ is a spanning tree of P.
2. If $v - \rightarrow w$, then $w \overset{*}{\rightarrow} v$. That is, each edge not in the spanning tree T of P connects a vertex with one of its ancestors in T. The edges $v - \rightarrow w$ are called the *fronds* of P.

Then P is called a *palm tree*.

adjacency list If v is a vertex, the *adjacency list* $A(v)$ contains all w such that (v, w) is an edge of G. A set of such lists, one for each vertex in G, is called an *adjacency structure* for G.

1.3 Depth-first search

Depth-first search is a systematic way of exploring a graph [17]. To carry out a depth-first search of G, we first select a vertex to start with, say z, which is called the *root*. After the root vertex z is visited, 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 leading from z will be selected. This process 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.

We use a set of *adjacency lists*, one for each vertex. If G is undirected, each edge (v, w) is represented twice, once in $A(v)$ and once in $A(w)$. The following is the DFS procedure. It is easy to prove that the vertices are numbered so that $NUMBER(v) < NUMBER(w)$ if $v \overset{*}{\rightarrow} w$ in the spanning tree generated [17].

```
ALGORITHM DFS(G)
1   FOR each vertex v in V(G)
2       NUMBER(v) = FATHER(v) = 0;
3   time = 0;
```

```
  DFS-VISIT(root);
END ALGORITHM

PROCEDURE DFS-VISIT(v)
  NUMBER(v) = time = time + 1
  FOR w in the adjacency list of v DO
  {
       IF NUMBER(w) == 0 THEN
       {
            mark vw as a tree edge in P;
            FATHER(w) = v;
            DFS-VISIT(w);
       }
       ELSE IF NUMBER(w) < NUMBER(v) and w != FATHER(v) THEN
            mark vw as a frond in P;
  }
```

The edges whose exploration by DFS leads to an unvisited vertex are called *tree edges*, and the remaining edges are called *fronds*. The tree edges form the depth-first tree T. All the tree edges are from parent to child, and all fronds are from descendant to ancestor. Thus the depth-first search converts G into a palm tree. It is also obvious that if G is connected then T is a spanning tree of G.

1.4 Identifying separation pairs

By definition, a graph G is not 3-edge-connected if $\lambda(G) < 3$.

1. If $\lambda(G) = 0$, the graph is not connected.

2. If $\lambda(G) = 1$, the graph is not 2-edge-connected. In this case there is at least one edge e such that $G - e$ is not connected.

3. If $\lambda(G) = 2$, the graph is 2-edge-connected but not 3-edge-connected. In this case there is at least one pair of edges $S = \{e_1, e_2\}$ (a separation pair) such that $G - S$ is not connected.

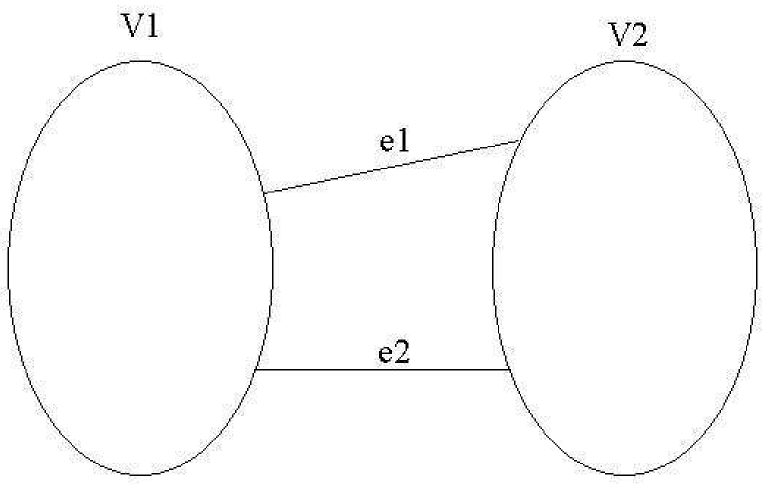

Figure 1.2: A partition of vertices V into disjoint sets V_1 and V_2.

Cases 1 and 2 are easily identified based on a simple depth-first search. We will assume that all graphs considered from now on are 2-edge-connected. An illustration of case 3 is shown in Figure 1.2.

As observed in the previous section, a depth-first search converts G into a palm tree and partitions the edges into a tree edge set and a frond set. If there is a separation pair S, it must consist of two tree edges, or else one tree edge and one frond. This is because it is impossible for two fronds to form a separation pair. Since the tree edges form a spanning tree of G, all the vertices are connected by tree edges. Removing fronds cannot increase the number of components in G. Thus the separation pair cannot consist of two fronds.

We call a separation pair *type 1* if it is formed by one frond and one tree edge. We call a separation pair *type 2* if it is formed by two tree edges. Figure 1.3 gives an example of a type 1 separation pair and an example of a type 2 separation pair.

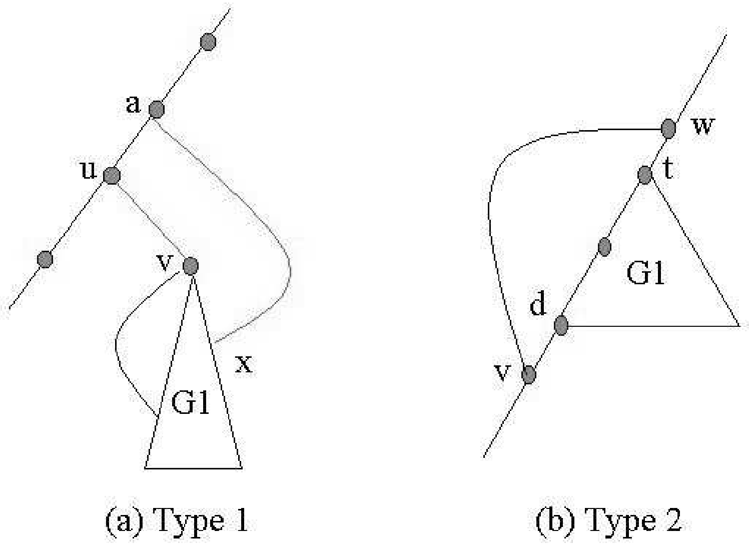

Figure 1.3: Examples of separation pairs.

1.5 Generating unlabeled graphs

- The software package *Nauty* by B.D. Mckay [11] was used to generate all of the unlabeled connected graphs of a given order. We applied our algorithm to check which of the graphs generated by *Nauty* are 3-edge-connected, thus providing the numbers of unlabeled 3-edge-connected graphs of order up to 11.

- *Nauty* can produce the order of the automorphism group for each unlabeled graph generated. A graph of order n with s automorphisms can be labeled in exactly $n!/s$ different ways [7, p.4]. We summed the numbers of labelings for the 3-edge-connected unlabeled graphs to obtain the numbers of labeled 3-edge-connected graphs of order up to 11. As a cross check, a similar test was done to obtain the numbers of labeled 3-edge-connected blocks of order up to 11. A graph is a *3-edge-connected block* if it is 2-connected and 3-edge-connected. The 2-connectivity algorithm used in our test is from *Nauty*. These numbers for labeled graphs were provided by S. K. Pootheri [14], and he also found the numbers of unlabeled 3-edge-connected blocks [15]. His calculations were algebraic and did not involve any graph generation.

- As a comparison, we also tested the linear 3-edge-connectivity algorithm of H. Nagamochi and T. Ibaraki [12]. Our implementation was a straightforward modification of Z. Chen's implementation [2].

The rest of the thesis is organized as follows. Chapter 2 presents the details of our linear 3-edge-connectivity algorithm. Based on our complexity analysis, the time required for the algorithm is $O(m + n)$, where m is the number of edges and n is the number of vertices in the graph. In Chapter 3, we discuss the implementations for general graphs and Feynman diagrams and compare them with implementations of other algorithms. The tests show that our implementations are correct and faster than those to which they were compared. Conclusions and future work are discussed in Chapter 4.

Chapter 2

A linear algorithm for checking 3-edge-connectivity

2.1 Preliminaries

The terminology and lemmas in this chapter are based on those in the paper of J. E. Hopcroft and R. E. Tarjan [9] with modifications for edge-connectivity. The resulting algorithm is a bit simpler than their algorithm for dividing a graph into 3-connected components. Partly this is because edge connectivity is easier to deal with than vertex connectivity; intuitively this is due to the fact that an edge is only incident to two vertices, whereas a vertex may be incident to many edges. But also the algorithm is limited to determining whether or not a graph is 3-edge-connected and does not need to partition the edges into 3-edge-connected components. To begin with, an expanded DFS, called FIRSTDFS, similar to the one in Section 1.3 will be performed to calculate the following additional values for each vertex.

LOWPT1(v) $LOWPT1(v)$ is the vertex w with lowest $NUMBER(w)$ that is reachable from v by traversing zero or more tree edges of graph G followed by at most one frond.

LOWPT2(v) $LOWPT2(v)$ is the vertex w with second lowest $NUMBER(w)$ that is reachable from v by traversing zero or more tree edges of graph G followed by at most one frond. $LOWPT2(v)$ equals $LOWPT1(v)$ if there exist two or more fronds $u_i - \rightarrow LOWPT1(v)$ such that $v \xrightarrow{*} u_i$.

ND(v) $ND(v)$ is the number of descendants of v in the palm tree of G.

We will use the values of $LOWPT1$, $LOWPT2$ and ND to help precisely define the characteristics of the two types of separation pairs in the next section.

```
ALGORITHM FIRSTDFS(G)
    FOR each vertex v in V(G)
    {
        NUMBER(v) = FATHER(v) = ND(v) = 0;
        LOWPT1(v) = LOWPT2(v) = 0;
    }
    time = 0;
    FIRSTDFS-VISIT(root);
END ALGORITHM

PROCEDURE FIRSTDFS-VISIT(v)
    NUMBER(v) = time = time + 1;
    LOWPT1(v) = LOWPT2(v) = NUMBER(v);
    ND(v) = 1;
    FOR w in the adjacency list of v DO
    {
        IF NUMBER(w) == 0 THEN
        {
             mark vw as a tree edge in P;
             FATHER(w) = v;
             FIRSTDFS-VISIT(w);
             IF LOWPT1(w) < LOWPT1(v) THEN
             {
                 LOWPT2(v) = MIN{LOWPT1(v), LOWPT2(w)};
                 LOWPT1(v) = LOWPT1(w);
             }
             ELSE IF LOWPT1(w) == LOWPT1(v) THEN
                 LOWPT2(v) = LOWPT1(w);
             ELSE
                 LOWPT2(v) = MIN{LOWPT2(v), LOWPT1(w)};
             ND(v) = ND(v) + ND(w);
        }
        ELSE IF NUMBER(w) < NUMBER(v) and w != FATHER(v) THEN
        {
             mark vw as a frond in P;
             IF NUMBER(w) < LOWPT1(v) THEN
             {
                 LOWPT2(v) = LOWPT1(v);
                 LOWPT1(v) = NUMBER(w);
             }
```

```
            ELSE IF NUMBER(w) == LOWPT1(v) THEN
                LOWPT2(v) = NUMBER(w);
            ELSE
                LOWPT2(v) = MIN{LOWPT2(v), NUMBER(w)};
        }
    }
```

2.2 Analysis of type 1 separation pairs

First, consider several lemmas related to the structure of a palm tree.

lemma 2.2.1 *$LOWPT1(v) \xrightarrow{*} v$ and $LOWPT2(v) \xrightarrow{*} v$ in P.*

Proof. $LOWPT1(v) \leq v$ by definition. If $LOWPT1(v) = v$, the result is immediate. If $LOWPT1(v) < v$, there is a frond $u - \rightarrow LOWPT1(v)$ such that $v \xrightarrow{*} u$. Since $u - \rightarrow LOWPT1(v)$ is a frond, $LOWPT1(v) \xrightarrow{*} u$. Since P is a tree, $v \xrightarrow{*} u$ and $LOWPT1(v) \xrightarrow{*} u$, either $v \xrightarrow{*} LOWPT1(v)$ or $LOWPT1(v) \xrightarrow{*} v$. Because $LOWPT1(v) < v$, it must be the case that $LOWPT1(v) \xrightarrow{*} v \xrightarrow{*} u$. So the lemma holds for $LOWPT1(v)$. The proof is the same for $LOWPT2(v)$.

lemma 2.2.2 *If G is 2-edge-connected, $LOWPT1(v) < v$ unless v is the root, in which case $LOWPT1(v) = v$.*

Proof. Suppose G is 2-edge-connected and there exists a non-root vertex v such that $LOWPT1(v) \geq v$. If w is the father of v, any path from v not passing through (v, w) remains in the subtree $T(v)$. If we remove this edge, v will be disconnected from w. This violates the assumption that G is 2-edge-connected. Hence it must be the case that $LOWPT1(v) < v$. If v is the root then $LOWPT1(v) = v$ since $LOWPT1(v) \leq v$ and v has no strict predecessor.

Recall that a type 1 separation pair is formed by one tree edge and one frond in the palm tree of G. The following lemma identifies type 1 separation pairs:

proposition 1 *A 2-edge-connected graph G has a type 1 separation pair if and only if there is a vertex w other than the root such that $LOWPT2(w) \geq w$.*

Proof. For the converse direction, suppose G is 2-edge-connected and there exists a non-root vertex w such that $LOWPT2(w) \geq w$. Since G is 2-edge-connected and w is not the root, $LOWPT1(w) < w$ by Lemma 2.2.2. Let $t = LOWPT1(w)$ and v be the father of w. $LOWPT1(w) < w$ implies that there must exist a vertex u such that $w \overset{*}{\rightarrow} u$ and $u - \rightarrow t$. Since $LOWPT2(w) \geq w$, u must be unique. Hence, all paths from w to v must include either (u, t) or (v, w). If we remove (u, t) and (v, w) from G, v and w will be disconnected, showing that $\{(u, t), (v, w)\}$ is a type 1 separation pair for G.

For the forward direction, suppose that a 2-edge-connected graph G has a type 1 separation pair $\{e_1, e_2\}$ in its palm tree. Without loss of generality, we assume that $e_1 = v \rightarrow w$ is a tree edge and $e_2 = u - \rightarrow t$ is a frond in the palm tree. After removing e_1 and e_2, $G' = G - \{e_1, e_2\}$ is not connected. Since we only removed one tree edge $v \rightarrow w$ from G, the vertices in $D(w)$ are connected by the tree edges among them. These vertices and the edges incident to them must form a connected component, say C_1, of D'. Similarly, the vertices in $V(G) - V(C_1)$ are also connected and belong to a connected component of D', say C_2. Now we prove that C_1 and C_2 would be connected if $LOWPT2(w) < w$. Let $t' = LOWPT2(w)$. Since $t' < w$, there is a frond $u' - \rightarrow t'$ such that $w \overset{*}{\rightarrow} u'$. The assumption $t' < w$ indicates that t' is a vertex in C_2. Clearly $u' \in C_1$. Since $u' = u$ and $t' = t$ cannot be true at the same time by the definition of LOWPT2 and the assumption that G is simple, (u', t') is an edge connecting C_1 and C_2 after e_1 and e_2 are removed. Thus all vertices of G are still connected after removing the separation pair. This is a contradiction. Hence $LOWPT2(w) \geq w$.

Proposition 1 leads to an efficient algorithm for finding type 1 separation pairs. If we insert the following lines into procedure FIRSTDFS-VISIT(u) between lines 10 and 11, the modified procedure FIRSTDFS will correctly identify type 1 separation pairs.

```
10.1    IF ( LOWPT1(w) < v and LOWPT2(w) >= w ) THEN
10.2        detect a type 1 separation pair, procedure stop;
```

2.3 Analysis of type 2 separation pairs

Before identifying type 2 separation pairs of a 2-edge-connected graph, we need to perform two depth-first searches and some auxiliary procedures.

Step 1. Perform FIRSTDFS(root) on the graph G to convert G into a palm tree P. Calculate $NUMBER(v)$, $FATHER(v)$, $LOWPT1(v)$, $LOWPT2(v)$ and $ND(v)$ for each vertex v in P.

Step 2. Construct an adjacency structure A for P by ordering the edges e in each adjacency list of P according to nondecreasing value of $\phi(e)$. Here ϕ is defined by:

1. $\phi(e) = NUMBER(w)$ if $e = v- \rightarrow w$.
2. $\phi(e) = LOWPT1(w)$ if $e = v \rightarrow w$.

Below is the algorithm to construct the adjacency structure A for Step 2.

```
PROCEDURE CONSTRUCT-ALIST(P, n)
    FOR i = 1 UNTIL n
        initialize Bucket(i) and A(i) to be empty lists;
    FOR (v, w)  an edge of P DO
    {
        compute phi(v, w);
        add (v, w) to Bucket(phi(v, w));
    }
    FOR i = 1 until n
        FOR (v, w) in Bucket(i) do
            add w to end of A(v);
```

The main purpose of Step 2 is to change the order of each adjacency list so that a child with lower $LOWPT1$ appears before a child with higher $LOWPT1$. Thus when Step 3 is applied to construct a new palm tree, the child with lowest $LOWPT1$ will become the leftmost child.

Step 3. Perform a new depth-first search of P using the adjacency structure A given by step 2. This search generates a set of paths in the following way: each time we traverse a tree edge we add it to the path being built. Each time we traverse a frond, the frond becomes the last edge of the current path. Thus each path consists of a sequence of tree edges followed by a single frond. Because of the ordering imposed on A, each path terminates at the vertex with the lowest possible value of $NUMBER$.

Renumber the vertices of P from n to 1 in the order they are last examined during the search. The new number is just the reverse of the finish time for each vertex. Recalculate $LOWPT1(v)$ and $LOWPT2(v)$ using the new vertex numbers. Also calculate $A(v), A1(v)$ and $HIGHPT(v)$ for each vertex v. Here $A1(v)$ is the first vertex in $A(v)$. $HIGHPT$ is defined as follows. For each vertex w, if $v- \rightarrow w$ is the first frond explored in step 3 which terminates at w, let $HIGHPT(w) = NUMBER(v)$. If there is no frond which terminates at w then $HIGHPT(w) = 0$.

Now we describe the algorithm to carry out Step 3.

```
ALGORITHM PATHFINDER()
1   Initialize s = 0;  //s denotes the start vertex of the current path
               m = n;  //where n is the number of vertices in G
                       //m is the last number assigned to a vertex
2   FOR i = 1 UNTIL n DO
3       NEWNUM(i) = HIGHPT(i) = 0;
4   CALL PATHFINDER-DFS(root);
5   Use NEWNUM to update A(v), A1(v),
   LOWPT1(v) and LOWPT2(v)


PROCEDURE PATHFINDER-DFS(v)
1   NEWNUM(v) = m - ND(v) + 1;
```

```
FOR w in the adjacency list A(v) of v DO
{
    IF s = 0 THEN
    {
        s = NUMBER(v);
        start a new path;
    }
    add (v, w) to current path;
    IF (v, w) is a tree edge THEN
    {
         PATHFINDER-DFS(w);
         m = m - 1;
    }
    ELSE                     //frond
    {
         IF HIGHPT(w) == 0 THEN
             HIGHPT(w) = NUMBER(v);
         save current path and reset s = 0;
    }
}
```

Step 3 renumbers the vertices of G from n to 1 in the order they are last visited during the search. In order for the calculation of $HIGHPT$ to proceed correctly, each vertex must be assigned a number the first time it is reached. Because the vertices to be reached between the first time v is visited and the time v is last visited are just the proper descendants of v, we assign the number to be the total number of unvisited vertices minus the number of descendants of v excluding v.

Figure 2.1 illustrates the differences in the palm tree before and after Step 3. The values in parentheses are the $NUMBER$ assigned to each vertex during DFS. Note that because $LOWPT1(E) < LOWPT1(B)$ in the first DFS, vertex E will be visited before vertex B in the second DFS. But vertex E gets number 5 instead of 2 in the second DFS because the way we assign $NEWNUM$ in Step 3.

lemma 2.3.1 *Let $A(u)$ be the adjacency list of vertex u. Let $u \to v$ and $u \to w$ be tree edges, with v occurring before w in $A(u)$. Then $u < w < v$.*

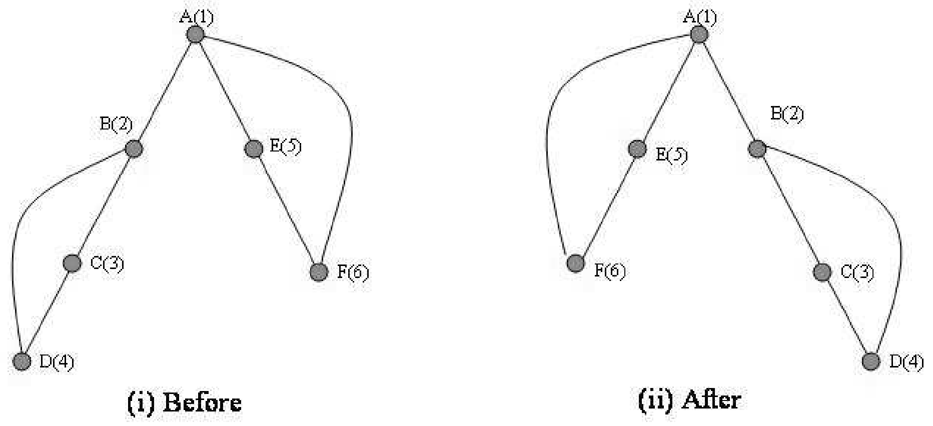

Figure 2.1: A palm tree before and after the second DFS.

Proof. Step 3 numbers the vertices from n to 1 in the order they are last examined in the search. If $u \rightarrow v$ is explored before $u \rightarrow w$, v will be examined last before w is examined last, and v will be assigned a higher number. Clearly u will be last examined after both v and w are last examined, so u receives the smallest number of the three vertices.

Now we give a few more definitions. If $u \rightarrow v$ and v is the first entry in $A(u)$, then v is called the *first son* of u. If $u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_k$, and u_i is a first son of u_{i-1} for $1 \leq i \leq k$, then u_k is called a *first descendant* of u_0.

lemma 2.3.2 *If v is a vertex and $D(v)$ is the set of descendants of v, then $D(v) = \{x | v \leq x < v + ND(v)\}$. If w is a first descendant of v, then $D(v) - D(w) = \{x | v \leq x < w\}$.*

Proof. Suppose we reverse all the adjacency lists $A(v)$ and use them to specify a depth-first search of P. Vertices will be examined for the first time in ascending order from 1 to n, if vertices are identified by their step 3 number. Thus descendants of v are assigned consecutive numbers from v to $v+ND(v)-1$. If w is a first descendant of v, vertices in $D(w)$ will be assigned numbers after all vertices in $D(v)-D(w)$. Thus $D(v)-D(w)=\{x|v \leq x < w\}$.

lemma 2.3.3 *Let $\{e_1, e_2\}$, where $e_1 = u \rightarrow a$ and $e_2 = v \rightarrow b$, be a type 2 separation pair in the palm tree of a 2-edge-connected graph G with $a < b$. Then $a \overset{*}{\rightarrow} b$ is in the spanning tree T of P.*

Proof. Since $a < b$, a cannot be a descendant of b. Suppose b is not a descendant of a. Since neither e_1 nor e_2 belongs to the subtree of a, vertices in $D(a)$ are connected by tree edges after removing e_1 and e_2. These vertices and the edges incident to them form a connected component, say C_1. Similarly, vertices in $D(b)$ also belong to a connected component of G, say C_2. Also, vertices in $V(G)-D(a)-D(b)$ are still connected. First, root s is in this set since neither a nor b is the root (neither is an ancestor of the other). So it's a non-empty set. For any vertex c in this set, c is connected to s because all of the tree edges in the path $s \overset{*}{\rightarrow} c$ are preserved. So all the vertices in $V(G)-D(a)-D(b)$ are connected. These vertices and the edges incident to them form another component, say C_3. Since G is two-edge-connected and a is not the root, $LOWPT1(a) < a$, by Lemma 2.2.2. Thus some edge is incident to a vertex in C_3 and to a vertex in C_1. Thus C_1 is connected to C_3. A similar argument shows that C_2 is connected to C_3. This means that the graph G is still connected after removing e_1 and e_2. This is a contradiction. Hence b must be a descendant of a. This implies that the path $a \overset{*}{\rightarrow} b$ lies in the spanning tree T of P.

proposition 2 *Suppose G is 2-edge-connected. Let $e_1 = u \rightarrow a$ and $e_2 = v \rightarrow b$ be two tree edges in G with $a < b$. Then $\{e_1, e_2\}$ is a type 2 separation pair of G if and only if the following conditions hold.*

(i) b is a first descendant of a;

(ii) every frond $x - \rightarrow y$ with $a \leq x < b$ has $a \leq y$;

(iii) every frond $x - \rightarrow y$ with $a \leq y < b$ has $a \leq x < b$;

Proof: For the converse direction, suppose that pair $\{e_1, e_2\}$ satisfies the three conditions. Since b is a first descendant of a, neither e_1 nor e_2 belongs to the subtree of b. Thus vertices in $D(b)$ are connected after removing e_1 and e_2. Hence they belong to a component of $G - \{e_1, e_2\}$, say C_1. Similarly, all vertices in $D(a) - D(b)$ are connected because all the tree edges among them are preserved. Let C_2 denote the component which contains $D(a) - D(b)$. By Lemma 2.3.2, $V(C_2) = \{x | a \leq x < b\}$. Vertices in $V(G) - D(a)$ are also connected because all the tree edges among them are preserved. These vertices and the edges incident to them also belong a component, say C_3. Since $V(C_2) = \{x | a \leq x < b\}$, condition (ii) states that there is no frond incident to a vertex in C_2 and a vertex in C_3. Condition (iii) states that there is no frond incident to a vertex in C_2 and a vertex in C_1. Hence $\{e_1, e_2\}$ is a type 2 separation pair of G.

For the forward direction, suppose that a 2-edge-connected graph G has a type 2 separation pair $\{e_1, e_2\}$ in its palm tree, where e_1 is $u \rightarrow a$ and e_2 is $v \rightarrow b$ with $a < b$. By Lemma 2.3.3, $a \xrightarrow{*} b$ in the spanning tree T of P. Let $V(1) = D(b)$, $V(2) = D(a) - D(b)$ and $V(3) = V(G) - D(a)$. By the same reasoning as for the converse direction, vertices in each $V(i)$ are connected by tree edges after removing e_1 and e_2.

We now prove that vertices in $V(1)$ are connected to vertices in $V(3)$ by some frond. Since G is two-edge-connected and b is not the root, $LOWPT1(b) < b$, by Lemma 2.2.2. If vertices in $V(1)$ are not connected to vertices in $V(3)$, $LOWPT1(b) >= a$. This means that some edge is incident to a vertex in $V(1)$ and a vertex in $V(2)$. Thus $V(1)$ is connected to $V(2)$. Similarly, $LOWPT1(a) < a$. Hence $V(3)$ is also connected to $V(2)$ or $V(1)$. This is a contradiction to the assumption that G is not connected after removing e_1 and e_2. Hence $LOWPT1(b) < a$ and vertices in $V(1)$ are connected to vertices in $V(3)$. These vertices and the edges incident to them form a component, say C_1. Vertices in $V(2)$ and the edges incident to them also form a component, say C_2. Clearly $C_1 \neq C_2$ since $\{e_1, e_2\}$ is a separation pair for G.

If b is not a first descendant of a, there must exist vertices d and d' such that (1) d is a first descendant of a; (2) $d \rightarrow d' \xrightarrow{*} b$; (3) d' is not the first son of d. Suppose d_1 is the first son of d. Since $d_1 \in C_2$, $LOWPT1(d_1) \geq a$. (Otherwise, C_2 is connected to C_1). Since $d' \xrightarrow{*} b$, $LOWPT1(d') \leq LOWPT1(b)$. Because $LOWPT1(b) < a$, $LOWPT1(d') < a$. Now we have $LOWPT1(d') < LOWPT1(d_1)$, which means that d' appears before d_1 in the adjacency list of d. This contradicts the condition that d_1 is the first son of d. So b must be a first descendant of a.

Since b is a first descendant of a, we must have $V(C_2) = \{x | a \leq x < b\}$. If Condition (ii) is false, there is a frond $x - \rightarrow y$ incident to a vertex in C_2 and a vertex in C_1. This violates the fact that C_1 and C_2 are two components of G. Thus condition (ii) is true. Similarly, condition (iii) is true. This completes the proof of the direct part of the proposition.

Proposition 2 is the basis for finding a type 2 separation pair. The proposition gives easy-to-apply conditions for identifying type 2 separation pairs. This may be done by using another depth-first search. Let $\{e_1, e_2\}$ be a type 2 separation pair satisfying $e_1 = u \rightarrow a$, $e_2 = v \rightarrow b$ and $a < b$. Then one component with respect to $\{e_1, e_2\}$ is $\{x | a \leq x \leq b - 1\}$. This follows from the proof of Proposition 2. An algorithm for finding type 2 separation pairs based on Proposition 2 is given in step 4. A new definition, $LOWEST(v)$, is needed for this algorithm. $LOWEST(v)$ is the vertex w with lowest $NUMBER(w)$ that is reachable from v by traversing zero or more tree edges of graph G followed by at most one frond and avoiding the first son of v. Let $MIN_{LOWEST} = min\{LOWEST(x) : a \leq x \leq b - 1\}$. Note that $a = MIN_{LOWEST}$. First, $a < MIN_{LOWEST}$ cannot be true because $MIN_{LOWEST} <= LOWPT1(a) <= a$. Second, $a > MIN_{LOWEST}$ cannot be true, otherwise $u \rightarrow a$ cannot be part of a separation pair because some frond $x- \rightarrow y$ with $a \leq x < b$ has $a > y$. This fact will be used in next step.

Step 4. The following procedure is used to find a type 2 separation pair. We test for separation pairs with Proposition 2 by examining the paths generated in order. To identify type 2 pairs, we keep a stack (called PStack) of pairs (a, b). The pair (a, b) corresponds to an edge pair $\{FATHER(a) \rightarrow a, FATHER(b) \rightarrow b\}$. When a type 2 separation pair is identified, the procedure returns 0 and exits immediately and at the end 1 is returned if no type 2 separation pair is found.

```
ALGORITHM PATHSEARCH()
FOR i = 1 UNTIL v DO
    LOWEST[i] = 0;
create empty stack PStack;
return PATHFINDER-DFS(root);
```

```
PROCEDURE PATHSEARCH-DFS(v)
(input: vertex v is the current vertex in the depth-first search)
FOR w IN A(v) DO
{
    IF vw is a tree edge THEN
    {
        IF vw is a first edge of a path THEN
        {
            IF LOWPT1(w) < LOWEST(v) and w is not A1(v) THEN
                LOWEST(v) = LOWPT1(w);
            add end-of-stack marker to PStack;
        }
11          returnVal = PATHSEARCH-DFS(w);
12          IF returnVal == 0 THEN
13              return 0;
14      }
15      ELSE // vw is a frond
16          IF NUMBER(w) < LOWEST(v) THEN
17              LOWEST(v) = NUMBER(w);
18  }
19  WHILE (a, b) on PStack satisfies a <= NUMBER(v) < b
       and LOWEST(v) < a
20       remove (a, b) from PStack;
21  IF no pair deleted from PStack THEN
22       add ( LOWEST(v), A1(v) ) to PStack;
23  IF (a, b) is the last pair deleted from PStack THEN
24       add ( LOWEST(v), b ) to PStack;
25  WHILE (a, b) on PStack satisfies  a <= NUMBER(v) < b
       and HIGHPT(v) >= b
26       remove (a, b) from PStack;
27  WHILE (a, b) on PStack satisfies  a = v
28  {
29       IF ( b <= a or a == 0 ) THEN
30          delete (a, b) from PStack;
31       ELSE //we find type 2 pair {(FATHER(a), a), (FATHER(b), b)}
32          return 0;
33  }
34  IF ( FATHER(v), v) is a first edge of a path THEN
35      delete all entries on PStack down to
         and including end-of-stack marker;
36  return 1;
```

lemma 2.3.4 *PATHSEARCH correctly finds type 2 separation pairs in a 2-edge-connected graph G.*

Proof: We must prove two things: (i) if the procedure returns 0, there is at least one type 2 separation pair in G; (ii) if G has a type 2 separation pair, the procedure will return 0.

For (i), suppose that the procedure returns 0 at line 32 after examining $\{(FATHER(a), a), (FATHER(b), b)\}$. The pair must satisfy the test in line 27 upon leaving vertex v. Because a and b lie on a same path generated in step 3,

b must be a first descendent of a . If some frond $x- \rightarrow y$ with $a \leq x < b$ has $y < a$, $(a,\ b)$ would have been deleted from PStack at line 20 when x is visited. Similarly, if some frond $x- \rightarrow y$ with $a \leq y < b$ has $x > b$, $(a,\ b)$ would have been deleted at line 26 when vertex y was examined. It follows that (a, b) meets the conditions in Proposition 2. Hence, $\{(FATHER(a),\ a),\ (FATHER(b),\ b)\}$ is a type 2 separation pair.

For (ii), suppose G has a type 2 separation pair $\{e_1, e_2\}$ where $e_1 = u \rightarrow a$ and $e_2 = v \rightarrow b$ with $a < b$. $(LOWEST(v), b)$ will be added to PStack at line 22 when vertex v is examined. This pair cannot be deleted at line 26 because condition (iii) of Proposition 2 must be true. This pair can only be deleted from PStack at line 20, but it will always be replaced by a new pair of the form $(LOWEST(v'), b)$ at line 24 with $LOWEST(v)\ \geq\ LOWEST(v')\ \geq\ a$ when we visit vertex v'. Eventually such a pair will satisfy the condition at line 27 and the procedure will return 0 at line 32.

2.4 Time complexity analysis

To find a type 1 separation pair, we only need one depth-first search. This requires $O(m + n)$ time, where m is the number of edges and n is the number of vertices in the graph. In the process for finding type 2 separation pairs, step 1 and step 3 are also depth-first searches. These require $O(m+n)$ time, including various tests. Step 2 is a bucket sort of edges. It requires $O(m+n)$ time. In Step 4, the number of pairs added to PStack is $O(m + n)$. Each pair may only be modified if it is on top of the stack. Thus the time necessary to maintain PStack is also $O(m+n)$. Hence the time required for the algorithm is $O(m + n)$.

Chapter 3

Implementation and test results

We have implemented the algorithm for general graphs and Feynman diagrams. For general graphs, we compare the performance with Z. Chen's accelerated algorithm [2]. For Feynman diagrams, we compare the performance with Q. Wang's quadratic algorithm [18].

3.1 Testing on general graphs

Our first implementation was used to check graphs generated by *Nauty*. In *Nauty*, a graph is represented by an array of n sets where each set is made up of a number of setwords. Each set represents a vertex. The i-th set gives the vertices to which vetex i is adjacent, for $0 \leq i < n$. We use *Nauty* to obtain the number of unlabeled 2-connected graphs according to order and number of edges. Then our program is used to check whether a generated graph is 3-edge-connected or not. For each unlabeled graph, we can get the number of labeled isomorphic graph for it. Table 3.1 shows the numbers of unlabeled and labeled graphs by order n. Table 3.2 shows the numbers of unlabeled and labeled graphs by order n and size m. The numbers for labeled graphs agree with those obtained by S. K. Pootheri [14]. Similarly, we obtain the numbers of 3-edge-connected blocks of order up to 11 by combining our 3-edge-connectivity algorithm with a 2-connectivity algorithm from *Nauty*. These numbers are also agree with those obtained by S. K. Pootheri [14, 15].

Table 3.1: Numbers of unlabeled and labeled 3-edge-connected graphs by order n.

n	unlabeled	labeled
4	1	1
5	3	26
6	19	1858
7	150	236926
8	2583	53456032
9	84186	21493860332
10	5202329	15580415345706
11	577063391	20666613177952152

Because the number of graphs increases exponentially when order n increases, it is impractical to test the running time of our program for higher order graphs in *Nauty*. Instead, we wrote another program to generate higher order random graphs. We used an array of adjacency lists to represent a random graph. For comparison purposes, we tested the accelerated algorithm by Z. Chen [2]. For each order, we kept increasing the number of edges in each graph until about half of the graphs generated were 3-edge-connected. Then we used these graphs as inputs to the two algorithms. Table 3.3 shows the average testing times for both algorithms. The data shows that the average running time for either algorithm increases gradually as the order increases. Our algorithm is faster by a constant factor of approximately 1.5. To get a better idea of how the algorithms perform, we also calculated scaled times for each algorithm by dividing the actual testing time of each graph by $(n + m)$, where n is the order and m is the size. The fact that the scaled times for our algorithm stay essentially constant validates our time complexity analysis. Figure 1.2 illustrates these results clearly.

Table 3.2: Numbers of unlabeled 3-edge-connected graphs by order n and size m.

n	m	number of graphs	n	m	number of graphs	n	m	number of graphs
4	6	1	9	23	8235	10	49	424
5	8	1	9	24	5226	10	50	164
5	9	1	9	25	2966	10	51	66
5	10	1	9	26	1537	10	52	26
6	9	2	9	27	737	10	53	11
6	10	4	9	28	333	10	54	5
6	11	5	9	29	144	10	55	2
6	12	4	9	30	62	10	56	1
6	13	2	9	31	25	10	57	1
6	14	1	9	32	11	11	17	159
6	15	1	9	33	5	11	18	4500
7	11	4	9	34	2	11	19	49024
7	12	18	9	35	1	11	20	300079
7	13	30	9	36	1	11	21	1236614
7	14	34	10	15	14	11	22	3792554
7	15	29	10	16	306	11	23	9206283
7	16	17	10	17	3321	11	24	18429089
7	17	9	10	18	18426	11	25	31316963
7	18	5	10	19	64826	11	26	46151147
7	19	2	10	20	163711	11	27	59939408
7	20	1	10	21	319090	11	28	69460785
7	21	1	10	22	503239	11	29	72516806
8	12	4	10	23	663925	11	30	68718688
8	13	32	10	24	750352	11	31	59457139
8	14	134	10	25	739326	11	32	47188377
8	15	309	10	26	643804	11	33	34478944
8	16	465	10	27	500701	11	34	23260220
8	17	505	10	28	350608	11	35	14522369
8	18	438	10	29	222644	11	36	8408495
8	19	310	10	30	129030	11	37	4523777
8	20	188	10	31	68623	11	38	2266257
8	21	103	10	32	33736	11	39	1060080
8	22	52	10	33	15464	11	40	464664
8	23	23	10	34	6657	11	41	191795
8	24	11	10	35	2735	11	42	75099
8	25	5	10	36	1091	11	43	28156
8	26	2	10	37	424	11	44	10216
8	27	1	10	38	164	11	45	3652
8	28	1	10	39	66	11	46	1301
9	14	22	10	40	26	11	47	466
9	15	271	10	41	11	11	48	172
9	16	1357	10	42	5	11	49	67
9	17	3967	10	43	2	11	50	26
9	18	7953	10	44	1	11	51	11
9	19	11904	10	45	1	11	52	5
9	20	14134	10	46	6657	11	53	2
9	21	13828	10	47	2735	11	54	1
9	22	11465	10	48	1091	11	55	1

Table 3.3: Average testing times and scaled times for two algorithms on random graphs.

Order	Size	Averages in msec. for Sun's algorithm	Scaled times for Sun's algorithm	Averages in msec. for Chen's algorithm	Scaled times for Chen's algorithm
10	24	0.025	0.76	0.036	1.08
20	58	0.051	0.66	0.077	0.99
50	187	0.149	0.63	0.221	0.93
100	436	0.356	0.66	0.54	1.01
200	970	0.786	0.67	1.207	1.03
300	1500	1.224	0.68	1.895	1.05
400	2100	1.832	0.73	2.982	1.19
500	2700	2.237	0.7	3.745	1.17
600	3280	2.654	0.68	4.123	1.06
700	3950	3.275	0.7	5.141	1.11
800	4550	3.692	0.69	5.838	1.09
900	5150	4.003	0.66	6.588	1.09
1000	5950	5.064	0.73	7.958	1.15
2000	12530	13.329	0.91	19.102	1.31

3.2 A simplified algorithm for Feynman diagrams

3.2.1 Introduction to Feynman Diagrams

Feynman diagrams, which were introduced into quantum field theory by Richard Feynman in 1949, are widely used in physics to calculate rates for electromagnetic and weak interaction particle processes [13]. The diagrams provide a convenient way for physicists to organize their calculations. A Feynman diagram of order n can be viewed as a perfect matching on $2n$ vertices (edges in the matching are undirected and called V-lines) along with a permutation of the $2n$ vertices (represented by directed edges called G-lines).

Feynman diagram expansions are used in quantum physics to express the energy of a system of particles. Each line in a Feynman diagram represents the propagation of a free elementary particle and each vertex represents an interaction of elementary

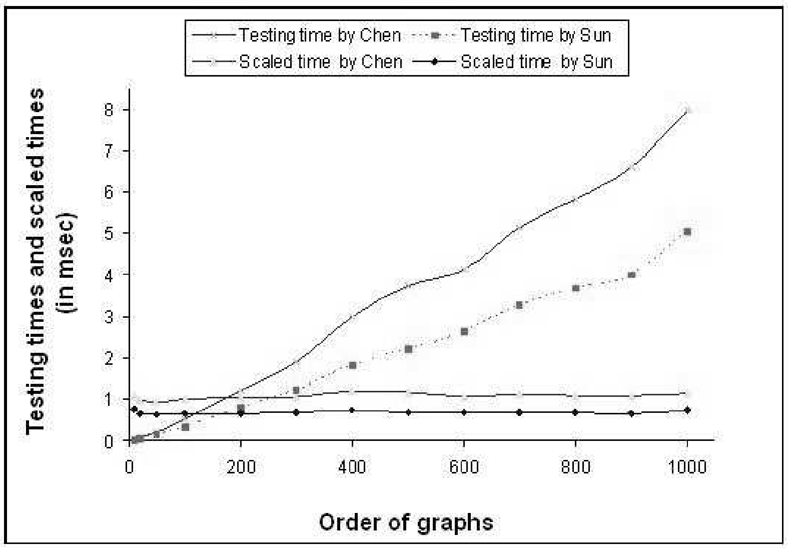

Figure 3.1: Comparison of running times and scaled times for 3-edge-connectivity algorithms.

particles. Since most Feynman expansions employ only connected diagrams, it is essential to maintain diagram connectivity. In more specialized expansions, it is helpful to consider irreducible diagrams (3-edge-connected with respect to G-lines).

For purposes of computation in this thesis, the vertices of an order n Feynman diagram are numbered $\{0, , \ldots, 2n-1\}$, and the V-line labeled i joins vertices $2i$ and $2i+1$, for $i = 0, \ldots, n-1$. Further, the G-lines are labeled $\{0, 1, \ldots, 2n-1\}$ so that the one with label j is incident from vertex j. In this notation, a Feynman diagram of order n can be represented by an array of size $2n$.

3.2.2 Simplifications for Feynman diagrams

A vertex in a Feynman diagram is adjacent to only three edges; a V-line, an outgoing G-line and an incoming G-line. Feynman diagrams satisfy the following five special properties.

1. *Every G-line belongs to a cycle (called a G-cycle) formed only by G-lines. The G-cycles partition both the vertex set and the set of G-lines of the diagram.*

2. *In a connected Feynman diagram, all G-cycles are connected by V-lines.*

3. *Removing a single G-line from a connected Feynman diagram cannot disconnect it.*

   In view of Property 1, every G-line belongs to a G-cycle. Removing an edge from a cycle leaves a path which still connects the vertices of the cycle.

4. *Removing two G-lines $\{g_1, g_2\}$ from two different G-cycles c_1 and c_2 cannot disconnect a connected Feynman diagram.*

   As for Property 3, note that the vertices of c_1 are still connected by the path $c_1 - g_1$, and similarly for c_2.

5. *A separation pair must consist of two G-lines from a same G-cycle.*

   A separation pair in Feynman diagram must consist of two G-lines. By property 4, these two G-lines must belong to a same G-cycle.

Property 5 is the basis for our simplified algorithm. Since a separation pair in a Feynman diagram must belong to the same G-cycle, the G-cycles play the same role as the paths in a general 3-edge-connectivity algorithm. For a Feynman diagram, the second DFS in the original algorithm can be removed due to the simple structure of the palm tree generated by a DFS. Each node in a palm tree can have at most two children since only V-lines and outgoing G-lines can be tree edges. Hence there is no need to reorder the adjacency lists for the vertices. Thus, a simplified algorithm can find separation pairs in a connected Feynman diagram in only two DFSs, as follows:

1. Apply the first DFS to a Feynman diagram of order n to generate a palm tree. The edges are visited in the following order: first the outgoing G-line, then the V-line. There is no need to visit the incoming G-lines explicitly because G-lines are directed and the incoming G-line to a child vertex is the outgoing G-line from the parent vertex. This order ensures that G-lines in a same G-cycle will be placed in a same path. To record a G-cycle, we only need to mark the start vertex of each G-cycle because the whole cycle can be generated by following the G-line out from each vertex in turn. As in the first DFS for the general algorithm, the values of $NUMBER$, $LOWPT1$, $LOWPT2$ and ND will be recorded for each vertex. Also $HIGHPT$ and $NEWNUM$, which were recorded in the second DFS of the general algorithm, will now be recorded during the first DFS. At the same time, type 1 separation pairs will be checked as in the general algorithm but with a test to make sure that both edges are G-lines.

2. Construct a $REVNUM$ array which is the reversed array of $NUMBER$. So if $NUMBER[v] = i$, then $REVNUM[i] = v$.

3. Convert the values in $LOWPT1$, $LOWPT2$, $HIGHPT$ from $NUMBER$ to $NEWNUM$. For example, if $LOWPT1[i] = j$ then $LOWPT1[i]$ should be changed to $NEWNUM[REVNUM[j]]$.

4. Apply a second DFS, similar to PATHSEARCH, to detect type 2 separation pairs.

Table 3.4: Comparison of running times for our linear algorithm and a quadratic algorithm.

order	number of diagrams tested	linear (sec.)	quadratic (sec.)	ratio of times
6	15,338	0.1	0.2	0.5
7	236,122	1.6	3.37	0.47
8	4,093,058	32	70.71	0.45
9	78,951,250	693.4	1624.3	0.43
10	1,678,001,642	16857.4	40855.6	0.41

The implementation for Feynman diagrams has been tested with a program by R.W.Robinson [16] which generates all canonical nearly irreducible (connected and without self-loops on bows) Feynman diagrams of a given order. Because the number of diagrams increases exponentially as the order n increases, it would take too long to test high order diagrams. Thus we only tested diagrams of order up to 10. Q. Wang developed three algorithms to determine the 3-edge-connectivity of Feynman diagrams; a quadratic algorithm, a pseudolinear algorithm and a randomized algorithm [18]. The quadratic algorithm is the best for order less than 12. Table 3.4 compares our results with the results from the quadratic algorithm. Our linear algorithm outperforms the quadratic algorithm even when the order is very low. Another comparison could be done between our algorithm and Z. Chen's accelerated algorithm. However, since our algorithm for general graphs is faster than his algorithm and our algorithm for Feynman diagrams is even faster than for general graphs, we can safely infer that our algorithm will also outperform Z. Chen's algorithm for Feynman diagrams.

Chapter 4

Conclusions and future work

In this thesis, a new algorithm for determining whether a graph is 3-edge-connected is presented. It was developed by adapting the 3-connectivity algorithm of Hopcroft and Tarjan [9]. The algorithm uses three DFSs to check for separation pairs in a general graph. The time required for the algorithm is $O(m + n)$, where m is the size and n is the order of a graph, so the algorithm is theoretically optimal to within a constant factor. A variation of the algorithm was developed specifically for testing Feynman diagrams, and uses only two DFSs. The general algorithm and the variation for Feynman diagrams were both implemented. Experiments were carried out to compare their performance with that of other algorithms and to ensure the correctness of the implementations. These tests showed that the implementations are correct and also faster than the available alternatives.

One direction for future work would be to extend our algorithm to deal with the fully dynamic 3-edge-connectivity problem. In a fully dynamic graph problem, a graph G has a fixed vertex set V. The graph G may be updated by insertions and deletions of edges. These operations may be interspersed with queries about the properties of the graph. A fully dynamic 3-edge-connectivity algorithm should be able to answer queries about whether a graph is 3-edge-connected after some update operations by taking advantage of previous computations. The success of such dynamic algorithms relies on suitable data structures to store information about the graph from previous computations. Several data structures have been applied to this problem, including topology trees [4], ET-trees [8], splay forests and treap forests [10].

For Feynman diagrams, the most natural update operation is a switching, which combines two edge deletions with two edge insertions [10]. The query is always to make sure that the diagram remains 3-edge-connected. If none of the G-cycles contains a separation pair, the diagram is 3-edge-connected. A switching operation can have two types of effect on a Feynman diagram: splitting a G-cycle into two G-cycles or joining two G-cycles into one G-cycle. A promising direction for future research is to find an efficient 3-edge-connectivity algorithm for Feynman diagrams which is dynamic with respect to switching operations.

Bibliography

[1] B. Bollobás, *Modern Graph Theory*, Springer-Verlag, New York, NY, 1998.

[2] Z. Chen, *A Linear Time Algorithm for Testing a Graph for 3-Edge-Connectivity*, M.S. thesis, University of Georgia, Athens, GA, 2001.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 2^{nd} edition, MIT Press, Cambridge, MA, 2001.

[4] G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, *SIAM J. Comput.* 14 (1985), 781-798.

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

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

[7] F. Harary and E. M. Palmer, *Graphical Enumeration*, Academic Press, New York, NY, 1973.

[8] M. R. Henzinger and V. King, Maintaining minimum spanning trees in dynamic graphs. *Internat. Colloq. on Automata, Languages, and Programming* (1997), 594-604.

[9] J. E. Hopcroft and R. E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* 2 (1973), 135-157.

[10] R. Ji, *Dynamic Connectivity Algorithms for Feynman Diagrams*, M.S. thesis, University of Georgia, Athens, GA, 2002.

[11] B. D. Mckay, *Nauty User's Guide* (Ver. 2.0), Technical report TR-CS-9002, Computer Science Department, Australian National University, Canberra, ACT, 1990.

[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] N. Nakanishi, *Graph Theory and Feynman Integrals*, Gordon and Breach Science Publishers, New York, NY, 1971.

[14] S. K. Pootheri, *Counting Classes of Labeled 2-Connected Graphs*, M.S. thesis, University of Georgia, Athens, GA, 2000.

[15] S. K. Pootheri, *Characterizing and Counting Classes of Unlabeled 2-Connected Graphs*, PhD thesis, University of Georgia, Athens, GA, 2000.

[16] R. W. Robinson, Counting irreducible Feynman diagrams, *CATS seminar*, University of Georgia, Athens, GA, 1999.

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

[18] Q. Wang, *A Linear Time Algorithm for Testing a Feynman Diagram for 3-Edge-Connectivity*, M.S. thesis, University of Georgia, Athens, GA, 2001.

[19] D. B. West, *Introduction to Graph Theory*, Prentice-Hall, Upper Saddle River, NJ, 1996.