University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

2006

A study of word frequency in written Urdu

Quratulain H. Khan
University of Windsor

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

Recommended Citation

Khan, Quratulain H., "A study of word frequency in written Urdu" (2006). *Electronic Theses and Dissertations*. 828.
https://scholar.uwindsor.ca/etd/828

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

A study of 3-edge connectivity algorithms - refinement and implementation

by

Nima Norouzi

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

Windsor, Ontario, Canada
2007

©2007 Nima Norouzi

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

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-34913-7
Our file Notre référence
ISBN: 978-0-494-34913-7

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

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

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

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

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

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

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

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

Canada

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

Abstract

There are quite a number of linear algorithms to compute 3-edge connected components of a multi-graph. In this thesis, we study the three most efficient algorithms and exclude other algorithms that are obviously inferior as they use different types of transformation in multiple phases. We present a data structure model for cut-pair deletion in order to save space and to be able to handle larger input sizes on a platform. Using complexity arguments we also present a modification to one of the three algorithms that does not look for cut-pairs. We then show through our experimental results that this algorithm and another one that does not distinguish between cut-pairs have the fastest execution time, and each of them is better than the other for some cases. To the best of our knowledge, till now, there is no such an effort to show how the performance of the algorithms varies as the type and the size of given graph changes. Correctness proofs of the proposed way for cut-pair deletion and the modification are presented as well.

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

Dedication

I would like to dedicate this thesis to my lovely wife and my wonderful parents.

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

Acknowledgements

My entire master thesis has come to completion through the steady and invaluable support of the academic guidance for which I am eternally grateful to my supervising advisor, Dr. Y. H. Tsin. Without him, this thesis could not have been fulfilled. I also wish to express my gratitude to the other members of my master thesis committee, Dr. N. G. Zamani, Dr. J. Lu, and Dr. X. Chen for their helpful advice and critical readings during this work.

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

Contents

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

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

List of Figures

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

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

Chapter 1

Introduction

1.1 The graph abstract terminology

In mathematics and computer science a graph is an ordered pair $G = (V, E)$, where V is a set of *vertices* (or *nodes*) and E is a set of *edges* such that every edge in it is associated with two vertices in V. If E is a set of unordered pairs, then G is an undirected graph. Otherwise, G is a directed graph. Below we give some necessary graph related definitions:

End-point (or end-vertex)

The two vertices associated with an edge are called the *end-points* (or *end-vertices*) of the edge. The two vertices are said to be *connected* by the edge.

Incident on

An edge is *incident on* a vertex if the vertex is an end-point of the edge.

Adjacent

Two vertices of a graph are *adjacent* if there is an edge connecting them. Two edges of a graph are *adjacent* if they share a common end-point.

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

Degree

The *degree* of a vertex $v \in V$ in a graph G, denoted by $deg_G(v)$, is the number of edges incident on the vertex in G. The subscript G can be removed if no confusion could occur as a result.

Self-loop

A *self-loop* is an edge that connects a vertex to itself.

Trivial graph and Null graph

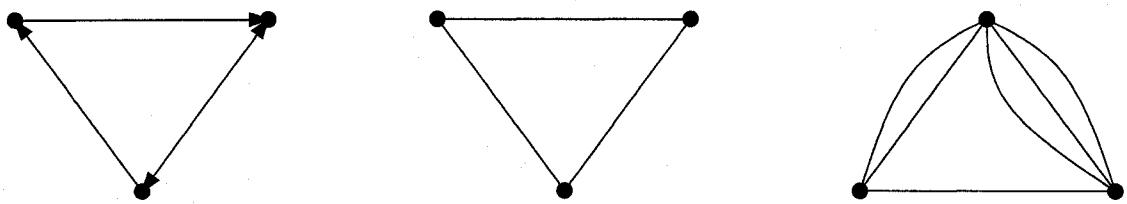
A *trivial graph* has one vertex and no edges. A *null graph* is an edgeless graph.

Subgraph

A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

Undirected and Directed edges

An edge in an undirected graph is an *undirected edge*. An edge in a directed graph is a *directed edge*. In a directed graph G, an ordered pair (v, w) denotes an edge from the vertex v to the vertex w in G. In an undirected graph G, (v, w) is an unordered pair which represents an edge in G having the vertices v and w as its end-points.



Directed Graph Undirected Graph Multigraph

Figure 1.1: Examples for directed graph, undirected graph and multigraph.

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

Multi-graph and Parallel edges

A *multi-graph* is a graph that can have multiple edges with the same endpoints. The edges are called *parallel edges*. In Figure 1.1, a multigraph is shown.

Clique

A *clique* in an undirected graph G, is a set of vertices V' such that each pair of V' is connected by an edge in G.

Path

A *path* P in a graph G is a sequence of vertices such that starting from the first vertex, there is an edge from each vertex to the next vertex in the sequence. Let v and w be the first and last, respectively, vertex in the sequence. Then the path *connects* or *is between* vertices v and w. We denote the path by $P_G(v, w)$ throughout this thesis. If G is directed then the path P is directed as well. The vertex v is called the *start vertex* and the vertex w is called the *end vertex*. We denote the *directed path* by $P_G < v, w >$. A *simple path* is a path that does not repeat any vertex except the first and the last which may be identical.

Cycle

A *cycle* is a simple path in a graph that starts and ends at the same vertex.

Acyclic graph

A graph with no cycle is an *acyclic* graph.

Connected graph

A graph that has a path between every pair of vertices.

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

Tree

A connected graph with no cycle is called a *tree* and is denoted by $T = (V_T, E_T)$.

Spanning tree

A *spanning tree* $T = (V_T, E_T)$ of a graph $G = (V, E)$ is a connected, acyclic subgraph of G such that $V_T = V$.

1.2 Depth First Search

Depth First Search (DFS) is a widely used technique which Tarjan [14] analyzed its properties in 1972. Let $G = (V, E)$ be the graph we wish to explore. Initially, all the vertices and edges in G are *unvisited.* We start at a vertex called "*root*", which can be any vertex in V, and explore as far as possible along each branch before backtracking, as we describe below:

- We select a vertex v.
- From vertex v, we follow an edge to reach another vertex w.
  - If the vertex w is unvisited, we apply the DFS to w and do the same thing to reach other vertices. However, any vertex with possibly unexplored edges is stored on a stack each time we apply DFS.
  - If the vertex w is visited, we select another unexplored edge to follow from the vertex.
- Whenever there is no unexplored edge from the vertex, we pop the top vertex on the stack, backtrack to that vertex and continue DFS from that vertex.

The algorithm clearly terminates because each vertex can only be visited once. Furthermore, each edge in the graph is examined exactly twice. Therefore, with a proper graph representation the time and space required by the search is $O(|V| + |E|)$ which is linear in $|V|$ and $|E|$ [14].

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

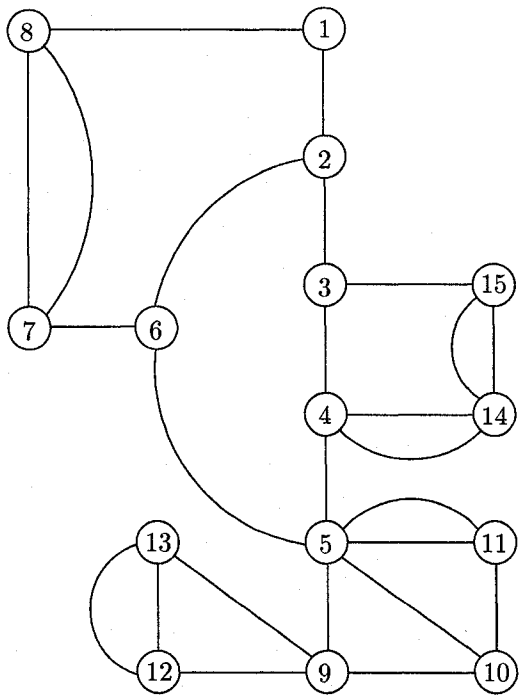


Figure 1.2: A multigraph $G = (V, E)$ that we wish to explore by the depth-first search technique.

1.2.1 DFS-tree and back-edges

Suppose we apply the DFS on the graph G. The DFS determines a spanning tree T of G and divides E into two edge sets E_T and B. The spanning tree T is also called "*DFS-tree*". B denotes a set of directed edges called "*back-edge*". Suppose $v, w \in V$ and $e = (v, w) \in E$. When the DFS follows the edge e from the vertex v and visits the vertex w for the first time, e is added to E_T and labeled as "tree-edge". On the other hand, when the DFS follows the edge e from the vertex v and visits the vertex w but verex w has been visited, then edge e is added to B and labelled as "back-edge". The DFS-tree and back-edges of the graph in Figure 1.2 are shown in Figure 1.3.

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

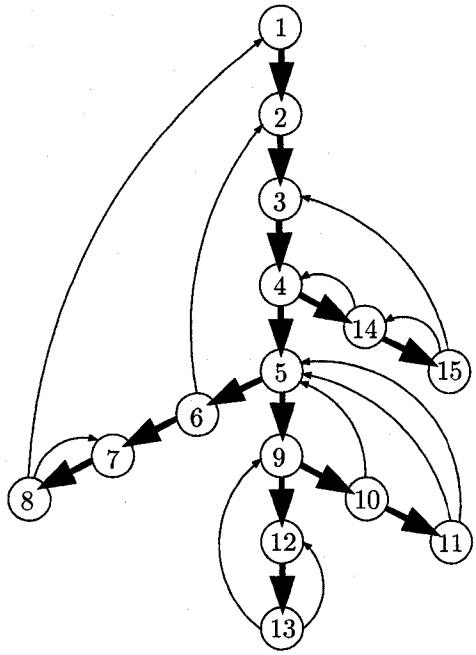


Figure 1.3: The depth-first search defines a DFS-tree T and back-edges from G. The bold edges denote to tree-edges and thin edges denote to back-edges.

1.2.2 Some definitions

Parent and Child and Leaf

A *parent* and a *child* are the end-points of a tree-edge in a spanning tree (see the DFS-tree in Figure 1.3) such that the direction of the tree-edge is from the parent to the child. A *leaf* is a vertex of a spanning tree that has no children. Let T be a spanning tree. Then the leafs in it cannot be parents and the root of it cannot be a child. Any other vertex in T is both a parent and a child.

Subtree

Let T be a DFS-tree of $G = (V, E)$ and $v \in V$. The *subtree* at v, denoted by T_v, is the largest subgraph of T which is a tree and has v as its root.

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

Ancestor and Descendant

Let $u, v \in V_T$. The vertex u is an ancestor of the vertex v iff v is a vertex in the subtree at u. Furthermore, the vertex u is a proper ancestor of v if $u \neq v$. The vertex v is a (proper) descendant of the vertex u iff u is a (proper) ancestor of v.

DFS number

A depth-first search assigns a distinct number to each vertex v in T, which is denoted by $dfs(v)$ and called the *DFS number* of v. The depth-first search assigns the first number, namely 1, to the root and based on the order it encounters unvisited vertices, it increments the previously assigned number and then assigns the number to the new vertex.

Incoming back-edge and Outgoing back-edge

A back-edge (v, w) is called an *incoming back-edge* of v if $dfs(v) \leq dfs(w)$ and is called an *outgoing back-edge* of v, if $dfs(v) > dfs(w)$.

Lowpt

$$lowpt(v) = min(\{dfs(v)\} \cup \{dfs(b) | there\ exists\ a\ P_T < v, a >\ and\ a\ back-edge\ (a, b)\})$$

In other words, $lowpt(v)$ is the smallest DFS number of a vertex which is reachable from v by traversing zero or more tree-edges followed by exactly one back-edge [14].

The calculation of $lowpt(v)$ for each vertex v is done as follows. When the DFS visits v for the first time, $lowpt(v)$ is initialized to $dfs(v)$. Whenever the DFS backtracks from a child w such that $lowpt(w)$ is smaller than the current value of $lowpt(v)$ or encounters a back-edge (v, u) with $dfs(u)$ smaller than the current value of $lowpt(v)$, $lowpt(v)$ is changed to $lowpt(u)$ or $dfs(u)$; respectively. When the DFS backtracks from v to its parent, the value of $lowpt(v)$ has been finalized.

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

1.2.3 Adjacency Lists

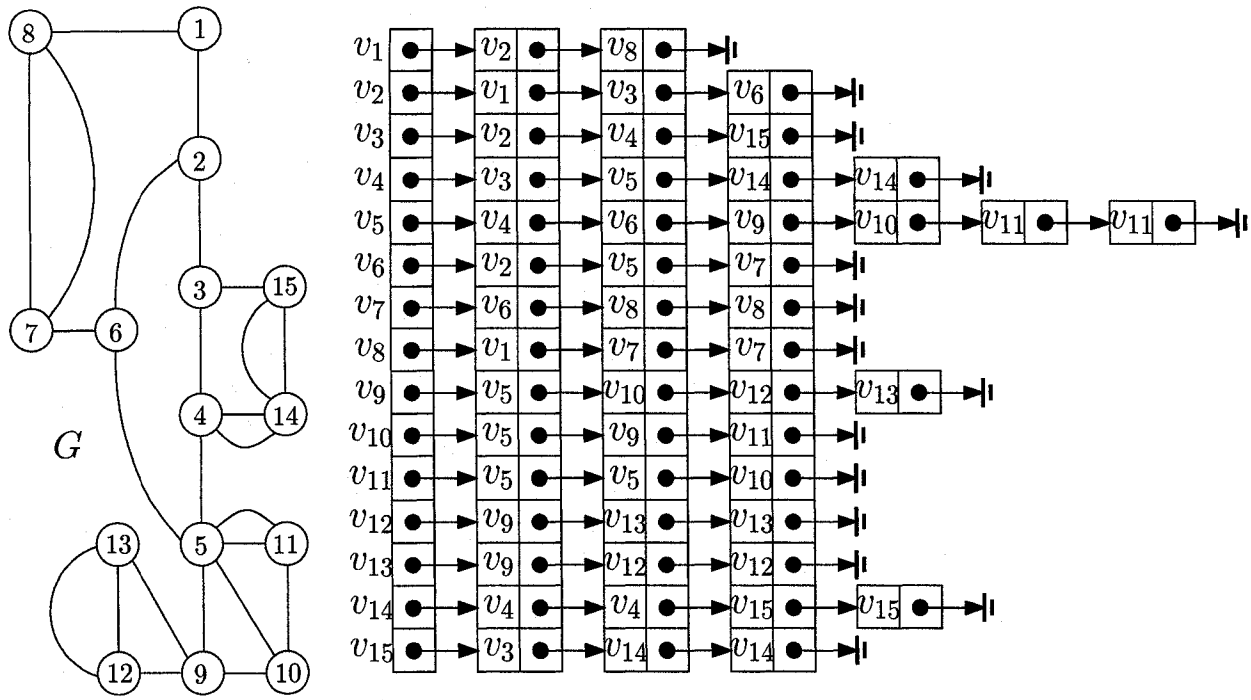


Figure 1.4: The graph G and its adjacency lists.

As we had mentioned, with a good graph representation, the DFS can be done in linear time and space. In fact the representation is nothing but a data structure [14]. Let $G = (V, E)$ be a graph. For each vertex v we construct a list containing all vertices w such that $(v, w) \in G$. Such a list is called an adjacency list for vertex v. Each element of the list is called *an edge* (v, w) *in the adjacency list of* v. Clearly, the edge (v, w) is denoted twice; once in the adjacency list of v, and once again in the adjacency list of w. A set of such lists, one for each vertex in G, is an adjacency lists data structure for G.

The depth first search provides ways to explore each edge and vertex of a graph so that it establishes the base of simple and effective graph connectivity algorithms.

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

1.3 Graph connectivity

The simplest connectivity problem is to determine whether the given graph is actually connected.

Connected component

A connected component of a graph is a maximal connected subgraph of that graph. Two vertices are in the same connected component if and only if there exists a path between them.

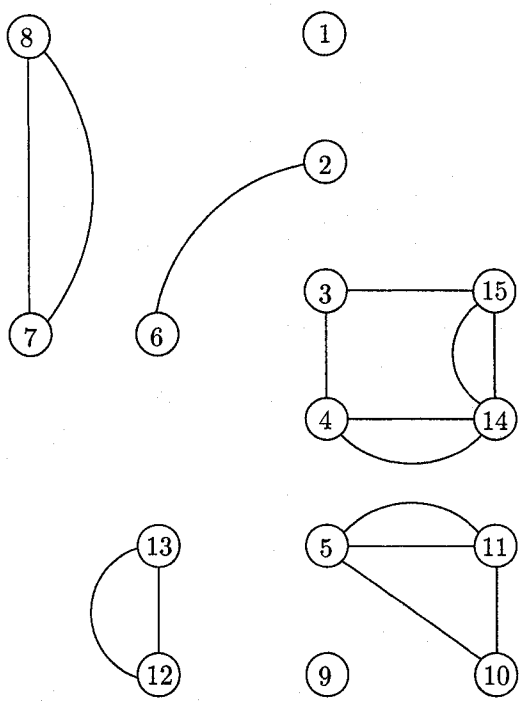


Figure 1.5: *A graph with 7 connected components.*

We use the DFS to find all connected components of an undirected graph. The idea is to have a list of vertices of G and select a vertex v of the list and call DFS(v). When the depth first search backtracks to v, the connected component including v is found. If there is a vertex u in the list that has not been visited, then call DFS(u). When the depth first search backtracks to u, another connected component including u is found. We continue the same process until no vertex is unvisited in the list. Then all the connected components of G are

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

found. The algorithm is as below:

```
Algorithm Connect; [17]
Input:  An undirected graph represented by adjacency lists;
Output:  The vertex sets of the connected components of G;
1  c = 0;
2  for (v=1;v<=Vnum;v++) visited[v] == 0;
3  for (v=1;v<=Vnum;v++)
4    if (visited[v] == 0) {
5      c = c + 1;
6      Print("A connected component:  ");
7      DFS-connect(v);
8    }

1  Procedure DFS-connect(v); {
2    S_c = S_c ∪ v; S_c is the set of vertices of connected component number c.
3    Print(v);
4    visited[v] = 1;
5    for each w in the adjacency list of v {
6      if (visited[w] == 0)
7        DFS-connect(w);
8    }
9  }
```

Measuring the connectivity of graphs can be used to analyze a broad range of structures and relationships. There are two different ways of measuring graph connectivity; vertex connectivity and edge connectivity. For example in the context of telephone network reliability, the vertex connectivity of the network is the smallest number of switching stations that must

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

fail in order to make the network disconnected so that some working stations will no longer be able to communicate with some other stations; the edge connectivity is the smallest number of wires that must be cut to give the same result [11] .

1.3.1 Vertex connectivity

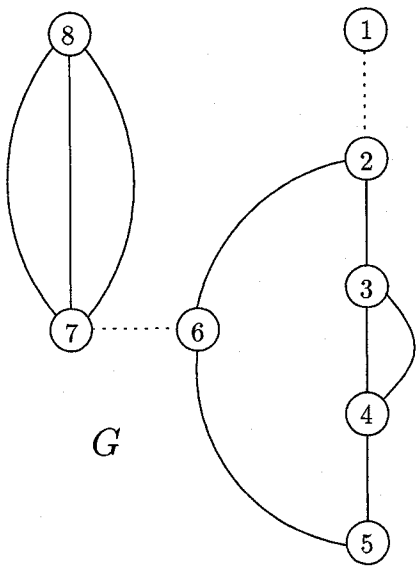
The smallest number of vertices whose deletion results in a disconnected graph.

Cut-vertex

A vertex whose removal from the given graph results in a disconnected graph.

1.3.2 Edge connectivity

The smallest number of edges whose deletion results in a disconnected graph. The edge connectivity of a trivial graph is defined to be ∞ [9]. The edge connectivity of G is denoted by $ec(G)$.



2-Edge Connected Components

{1}
{2,3,4,5,6}
{7,8}

Figure 1.6: A graph G and its 2-edge connected components. The dotted edges $(1,2)$ and $(6,7)$ are the bridges in G whose removals lead to the determination of 2-edge connected components in G.

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

Bridge

An edge whose removal from the given graph results in a disconnected graph.

2-edge connectivity

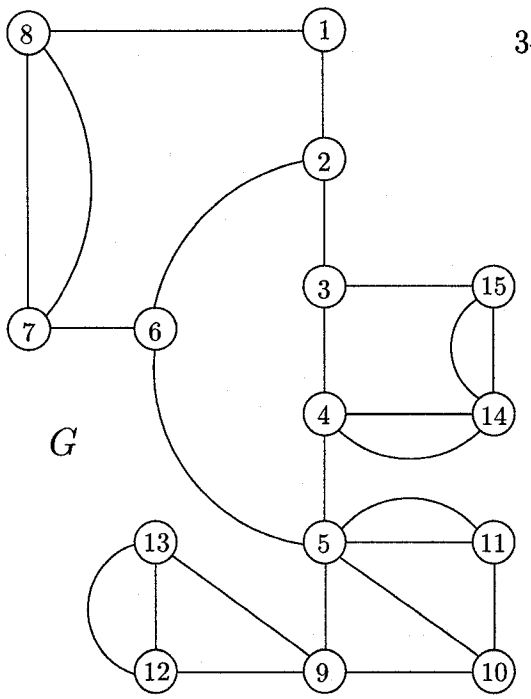
A graph G is *2-edge connected* or *bridgeless* if and only if $ec(G) \geq 2$. A 2-edge connected component in the graph is a maximal vertex set in which there exist two edge-disjoint paths between any pair of vertices in the set. Removing all bridges from G leads to the determination of 2-edge connected components in G.

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

Chapter 2

3-edge connectivity

A graph G is called *3-edge connected* if and only if $ec(G) \geq 3$. A 3-edge connected component in the graph is a maximal vertex set in which there exist three edge-disjoint paths between any pair of vertices in the set.



3-Edge Connected Components

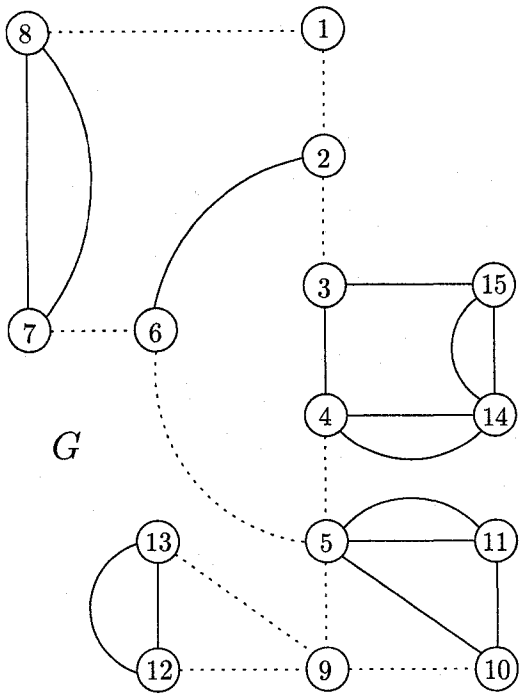
{1}
{2,6}
{3,4,14,15}
{5,10,11}
{7,8}
{9}
{12,13}

Figure 2.1: The graph G and its 3-edge connected components.

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

2.1 Cut-pair

A pair of edges is a *cut-pair* in G if their removal results in a disconnected graph and none of them is a bridge. A *cut-edge* is an edge in a cut-pair. It follows that if two vertices belong to the same 3-edge connected component, there is no bridge or cut-pair whose removal disconnects the graph.



Cut Pairs

{(1,2),(1,8)}
{(6,7),(1,8)}
{(9,12),(9,13)}

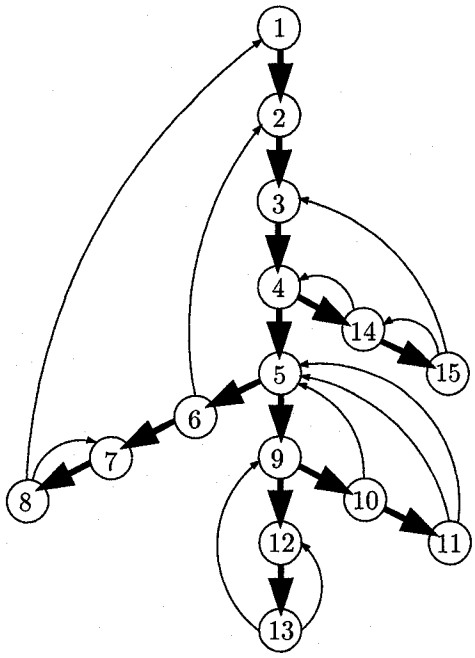
{(1,2),(6,7)}
{(2,3),(5,6)}
{(4,5),(5,6)}
{(2,3),(4,5)}
{(5,9),(9,10)}

Figure 2.2: The graph G and its cut-pairs which are shown by dotted lines.

2.1.1 Type-1 and Type-2 cut-pairs

After the DFS is applied on G, the cut-pairs of G are classified into two types: type-1 and type-2. A type-1 cut-pair is a cut-pair consisting of a tree-edge and back-edge. A type-2 cut-pair is a cut-pair consisting of two tree-edges. In Figure 2.3, the classified cut-pairs of G are shown.

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



Type 1 Pairs

{(1,2),(1,8)}
{(6,7),(1,8)}
{(9,12),(9,13)}

Type 2 Pairs

{(1,2),(6,7)}

{(2,3),(5,6)}
{(4,5),(5,6)}
{(2,3),(4,5)}

{(5,9),(9,10)}

Figure 2.3: The classified cut-pairs of G.

2.2 Generator

Suppose $e = (x, y)$ is a tree-edge and a cut-edge in T such that there is no edge in the subtree at y that forms a cut-pair with e. Then e is called a *generator*. It follows that a generator is a cut-edge in a type-2 cut-pair. In Figure 2.3, each of the tree-edges $(5, 6), (6, 7)$, and $(9, 10)$ is a generator.

2.3 Related works, and choosing 3 algorithms for study

In physics, a Feynman diagram consists of some undirected lines (Wiggly) denoting photons and directed lines (solid) denoting electrons and positrons. Each wiggly line has two end-points (interaction points), and two solid lines are attached to each end-point of a wiggly line in a Feynman diagram (see Figure 2.4). An *one-particle-irreducible* diagram is a connected Feynman diagram that cannot be disconnected by removing a single solid internal line [2]. An internal line is a line that is either a self-loop or participates at two interaction points;

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

any other line is an external line. On the other hand, a *G-irreducible* diagram is a connected Feynman diagram that cannot be disconnected by removing no more than two solid internal lines [3]. The G-irreducibility is useful in a new application in order to estimate the self energy of an interacting Fermion model [3]. In the G-irreducibility problem each solid line is called *G-line* and each wiggly line is called *V-line*.

An undirected graph G is obtained by contracting the end-points of each V-line, removing external lines, and disregarding the directions of all G-lines in a Feynman diagram. Note that all self-loops after the contractions are deleted. The relationship between G and the Feynman diagram is that the diagram is G-irreducible if and only if G is 3-edge connected. The order of a Feynman diagram is the number of wiggly lines which is denoted by n. In Figure 2.4 a few simple examples are shown.

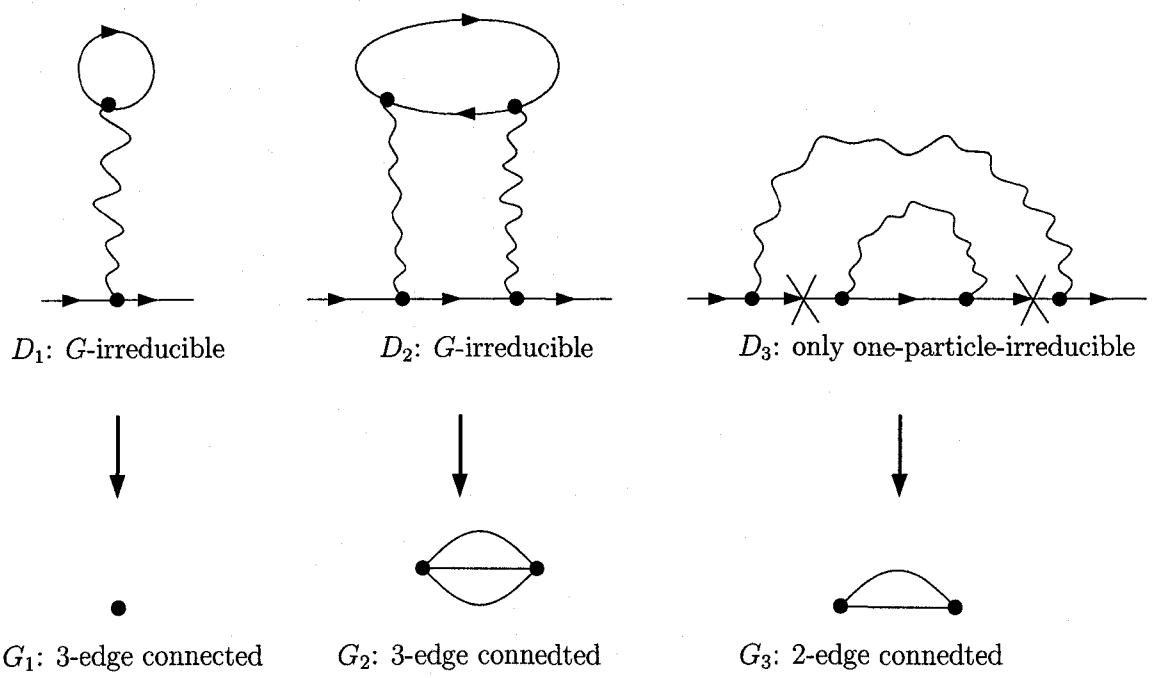


Figure 2.4: A Feynman diagram D_1 of order $n = 1$ and two Feynman diagrams D_2 and D_3 of order $n = 2$. The graphs G_1, G_2, and G_3 are obtained from the diagrams.

There is another research going on in the field of Bioinformatics so that 3-edge connectiv-

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

ity is useful, and one of the researchers had asked for our implementations [10]. The idea is to find relations between genes in a biological network obtained from microarray experiments using the *Cluster Editing* problem [4]. The Cluster Editing problem is defined as follows. *Input:* An undirected graph $G = (V, E)$ and a non-negative integer k. *Question:* Can G be transformed, by inserting and deleting at most k edges, into disjoint cliques? In the research the set of these edges includes cut-pairs of G as well.

Hopcroft and Tarjan [6] proposed a linear-time algorithm using the Depth-First Search Technique to divide a graph into 3-vertex connected components. Then Galil and Italiano [5] showed an approach for reduction of edge connectivity to vertex connectivity, and obtained a linear-time algorithm for computing all the 3-edge connected components of an undirected graph using the linear-time algorithm of Hopcroft and Tarjan [6].

A less complicated algorithm based on DFS was then reported by Nagamochi and Ibaraki [9] to solve the 3-edge connectivity problem directly without using reduction. The idea underlying the algorithm is to gradually remove vertices from the given graph so as to transform the graph into a trivial graph. First, any vertex with degree 2 (if exists) in the given graph is removed from the graph and transformed into a trivial graph; this transformation repeats more later if the remaining main graph is nontrivial (most of the time this is the case). The resulting nontrivial graph is then passed to a procedure called REDUCE. By calling the procedure, a DFS is applied on the graph in order to find three types of edge set, and the edge sets are used to find two other types of edge set of the graph; one of which contains all type-1 cut-pairs. The two edge sets are then used to find again two other types of edge set while there are consequently three kinds of major transformation involving some deletion (including type-1 cut-pairs) and addition of edges in order to break the input graph into a collection of smaller graphs each contains one or more 3-edge connected components of the given graph. Any of these smaller graphs, which is nontrivial, is recursively passed to the

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

REDUCE procedure and the same method is applied on each of them. In this way all type-2 cut-pairs of the given graph got passed to the procedure REDUCE, are eventually found as type-1 cut-pairs in the consequence calls of procedure REDUCE.

Nagamochi et al. proved that the total number of edges of the graphs which are passed to procedure REDUCE during the entire execution of the algorithm is bounded above by the number of edges of the original graph times 3. Besides being very complicated, this algorithm cannot use *simple* adjacency lists for graph representation because of the transformations and edge sets creations it performs. Therefore in order to have a linear time execution (to insert/access/delete any edge in $O(1)$ time), the algorithm needs to have a list of edge containers and must thus use the improved edge list representation (see chapter 2). This has resulted in the algorithm consuming substantially more time and space in comparison with the following three algorithms that we choose to study. In contrast with this algorithm, the three algorithms do not have to use improved edge lists. Instead, they can use simple adjacency lists for graph representation. We will show that by using simple adjacency lists, deleting cut-pairs can be done efficiently during the DFS.

Taoka et al. algorithm: The algorithm reported by Taoka et al. [13] uses only the DFS technique. It has three major phases and performs three depth-first searches. The idea is to find two types of cut-pair, type-1 and type-2, so that all 3-edge connected components "appear as connected components" after all the cut-pairs are removed and some necessary edges are added. In phase one, during a DFS on the given graph, all type-1 cut-pairs are found by computing some parameters for every vertex. Moreover, an important parameter called "path-partition number" is calculated for every vertex so that all end-points of each cut-pair must be located in one disjoint tree path in the DFS tree. In phase two, two key parameters, which will be used for determining type-2 cut-pairs in the next phase are computed for all the vertices. In phase three, type-2 cut-pairs for every path-partition are found;

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

during this phase a parameter for storing an end-point is updated for each vertex. After this phase, the value of the parameter is checked for each vertex and if it is not empty then an edge having the end-point value and the vertex as its end-points is added to the given graph. It should be mentioned that the depth-first searches in phase two and three traverse the DFS spanning tree which has been obtained in the first phase, and that back-edges are only used during phase two. This algorithm is simpler than that of Nagamochi et al. mentioned above. However, it performs multiple depth-first searches which induces a lot of overhead and lacks elegancy of the following two algorithms which each performs only one DFS.

Tsin's algorithm (The one without reduction):

This algorithm is first proposed by Tsin [16] as a distributed algorithm for finding 3-edge connected components on an asynchronous distributed computer network. However it can be easily converted into a simple linear algorithm which performs only one DFS on the input graph G. The algorithm does not classify cut-pairs and deals with each of them whenever it encounters the cut-pair during the DFS. This is accomplished through a transformation which converts the given graph G into a new graph G' so that the cut-pairs of the former are the type-2 cut-pairs of the latter. However, the transformation needs not be carried out explicitly. Therefore, the DFS is performed over G rather than the graph G'. Each time a cut-pair is found, it is deleted from G and a parameter for each vertex is updated so that after the DFS some virtual edges can be added to the graph. At the end, the 3-edge connected components of G "appear as connected components" of the modified G which is same as that produced by the algorithm of Taoka et al. [13].

Tsin's algorithm (The one with reduction):

This is an elegant linear algorithm proposed by Tsin [15]. The idea is to use one type of reduction to transform each 3-edge connected component of the given graph into a trivial graph in order to determine the vertex set of that 3-edge connected component. In contrast

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

with previous algorithms, this algorithm is not interested in finding cut-pairs. The entire computation on the given graph is done through only one DFS without any actual modification on the graph and the vertex set of each 3-edge connected component is determined by having a vertex in the component absorbs all the other vertices gradually during the DFS.

To the best of our knowledge, no other linear time algorithm based on the depth-first search technique has been reported for 3-edge connectivity.

Similar research had been carried out before. In his M.Sc. thesis [1], Chen implemented the algorithm of Nagomachi et al. [9] to test for 3-edge connectivity and presented experimental results. Later, in his M.Sc. thesis [12], Sun implemented the algorithm of Taoka et al. [13] to test for 3-edge connectivity and compared his experimental reults with those of Chen. Comparing to ours, their works were done at a much smaller scale. Firstly, they did not generate the 3-edge connected components. Secondly, they did not determine the cut-pairs. Thirdly, the graphs they had tested were of sizes at most 185,000. By contrast, we implemented and compared three, rather than just one, 3-edge connected component algorithms and not only test for 3-edge connectivity but also determine the cut-pairs as well as the 3-edge connected components. Furthermore, the input sizes we use in producing our experimental results are in the interval of 649,437 to 100,230,424, which is much larger than those of Chen and Sun.

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

Chapter 3

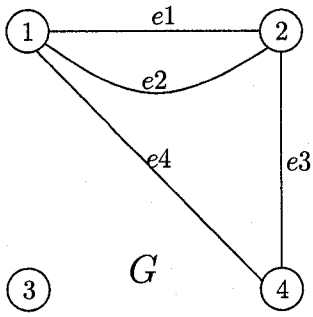
A model for cut-pair deletion from a multigraph

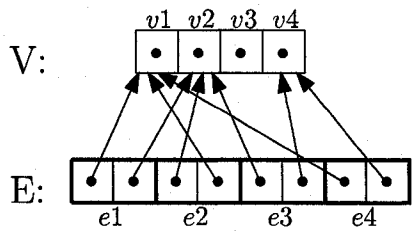
It is very important not to choose an imperfect data structure model to represent graphs for implementation of 3-edge connectivity algorithms; especially those that will find and delete cut-pairs. When it comes to delete a cut-pair, our graph representation model must allow us to access the cut-edges in an efficient way which does not affect our algorithm performance. For example, a naïve way is to search for the cut-edges in adjacency lists, which results in an undesirable non-linear time complexity.

There are three major approaches to represent graphs, namely *Edge List*, *Adjacency Lists*, and *Adjacency Matrix*. One might ask how to choose between representations? The answer is that it depends on the algorithm and trade-off between space and time efficiency. Any of the first two structures keeps real edges existing in a given graph but the adjacency matrix reserves a space for any pair of vertices regardless of whether an edge connecting them actually exists. Each of the edge list and adjacency list structures uses $O(|V| + |E|)$ space, while the adjacency matrix uses $O(|V|^2)$ space.

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

The edge list structure is a sequence of all vertices in G, and a sequence of unordered pairs of size $|E|$. Each pair consists of the end-points of a distinct edge in G (see Figure 3.1).





An Edge List representation of G.

Advantages: Finding end-points of an edge, access to/insertion/removal of an edge, and inserting a vertex can be done in $O(1)$ time.
Disadvantages: Finding incident edges of a vertex, determining whether two vertices are adjacent or not, and removing a vertex can be done in $O(|E|)$ time. Also this is not an efficient representation for the DFS technique.

Figure 3.1: *A graph with its Edge List representation.*

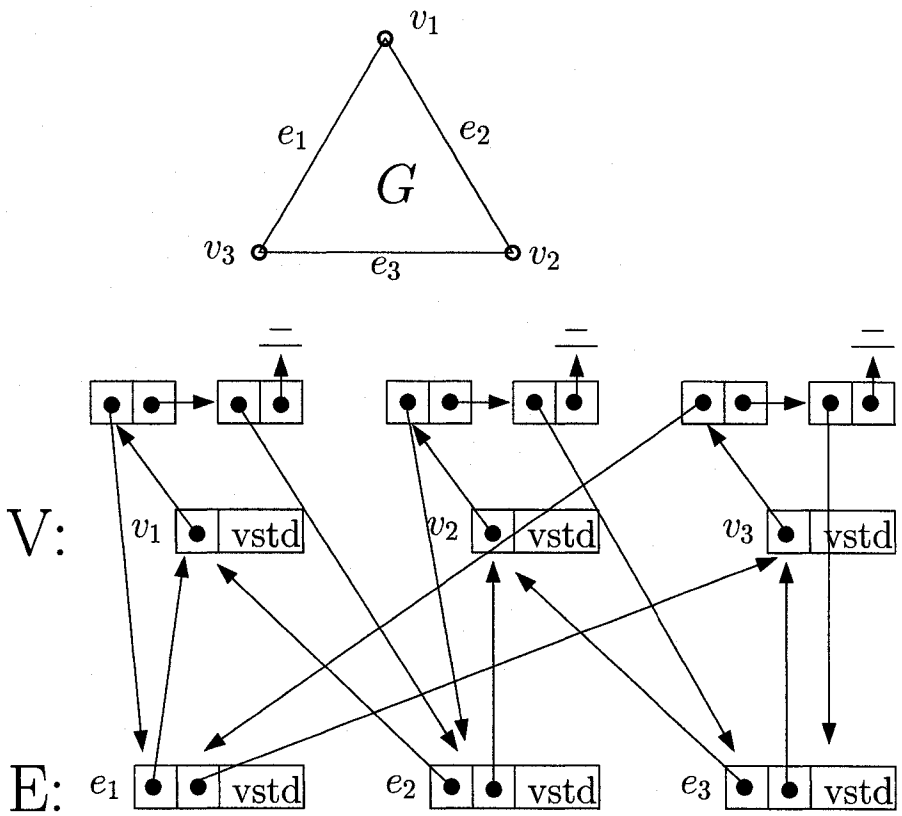
Therefore we can easily search for anything interacts with edges. We can insert/ access/ delete any edge in $O(1)$ time. That is why this structure is called "edge list". However, we also have a sequence of vertices, but "in keeping with the tradition" [7, 8], the structure is still called edge list structure.

But the edge list structure is not suitable for DFS based algorithms because handling a vertex is done in $O(|E|)$ time; so the entire time complexity would be $O(|V|.|E|)$. In fact the problem with edge list is that each vertex does not know which edges are incident to it. However, putting pointers from each vertex to the edges incident to the vertex, can improve the edge list structure. Since each vertex can have a lot of edges incident to it, we need a sequence of incident edges. As we mentioned in the previous chapter, this sequence is called an *Adjacency List.*

For the general DFS algorithm, Goodrich et al. [8] use the improved version of edge list (i.e. The edge list structure combined with adjacency lists; see Figure 3.2), in which each

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

vertex has also a list of edges incident to it. This could be useful for linear 3-edge connectivity algorithms which delete cut-pairs after they find them, because any edge can be deleted in $O(1)$ time, and access to incident edges of a vertex v can be done in $O(deg(v))$ time so that DSF also runs linearly; see Figure 3.2. Clearly this model itself needs at least $6|E| + O(V)$ space.



vstd: indicates whether the object is visited or not.

Figure 3.2: The improved Edge List model in order to represent input graphs for DFS.

Since in the 3-edge connectivity algorithms, we search and find cut-pairs, therefore instead of using the improved version of edge list, one might use the simple adjacency lists we explained in Chapter 1.2.3 and add back pointers from the list of vertex v ($v \in V_G$) to other lists that v exists in them in order to be able to efficiently delete both corresponding undirected edges. These edges are cut-pairs that must be deleted when they are found. Clearly

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

this model itself needs at least $4|E| + O(V)$ space.

Now we shall explain our data structure model which is especially useful for cut-pair deletion from a multigraph by using simple adjacency lists (See Figure 1.4) and adding two simple 1-dimensional arrays to the model each of size $|V|$; we will also explain these two arrays in the following definitions. The space needed for this model would be $2|E| + O(V)$ which outperforms the others we discussed above. To the best of our knowledge there is no better model than ours in order to be used for cut-pair deletions in 3-edge connectivity algorithms. When it comes to dealing with dense graphs ($|E| = \Omega(|V|^2)$) the difference among the described models would be significantly remarkable. In comparison with the other two models, our model can handle much larger input sizes for experiments on a platform and has shorter execution time.

Lemma 1: *Let (v, w) be a cut-edge that is an outgoing back-edge of v. Then there is no other cut-edge which is an outgoing back-edge of v.*

Proof: Suppose to the contrary that two outgoing back-edges (v, w) and (v, u) of v are both cut-edges. First, consider the case where $dfs(w) < dfs(u)$ (i.e. DFS number of w and u are not equal). Since these two edges are back-edges, they cannot form a cut-pair with each other because the DFS spanning tree keeps all the vertices connected. So each of the two edges must belong to a different cut-pair of which the other cut-edge (a, b) is a tree-edge. Clearly, the tree-edge (a, b) that forms a cut-pair with (u, v) must lie on the path $P_T < u, v >$ located in $T < u >$. Consider removing this cut-pair from G; $P_T < u, v >$ must be broken into two disjoint paths $P_T < u, a >$ and $P_T < b, v >$. Furthermore, G must become disconnected and the vertices a and b must be located in two different connected components. Since there is still a path $P_T < w, u >$, by concatenating the path $P_T < u, a >$ to it we then have a path $P_T < w, a >$ in T and hence in G. On the other hand, we have a path $P_T < b, v >$ in T,

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

and clearly in G. By adding the back-edge (v, w) to this path; a path $P_G(b, w)$ is obtained. Now, concatenating $P_G(b, w)$ and $P_G(w, a)$ gives rise to a path $P_G(b, a)$ which implies that vertices a and b belong to some connected components, a contradiction. In the case where $dfs(w) = dfs(u)$, the two back-edges are parallel edges, and none of them can be a cut-edge. The lemma thus follows. □

Definition 1:

$$back\_cutedge[v] = \begin{cases} w & \text{if } (v, w) \text{ is an outgoing back edge of } v \text{ that is a cut edge} \\ NULL & \text{otherwise} \end{cases}$$

Corollary 1: $\forall v \in V$, $back\_cutedge[v]$ is a singleton.

Definition 2:

$$parent\_edge[v] = \begin{cases} (v, w) \in G \text{ such that } (w, v) \text{ is a tree-edge and} & \\ \qquad w \text{ is the parent of } v & \text{if } v \neq r \\ NULL & \text{if } v = r \end{cases}$$

Lemma 2: *All cut-edges can be correctly marked as deleted during the DFS algorithm using the proposed model.*

Proof: Let (v, w) be a cut-edge. Suppose it is first discovered as an outgoing back-edge when the DFS encounters the edge at vertex v. Clearly, the other cut-edge that forms a

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

cut-pair with (w, v) is a tree-edge (a, b) such that $dfs(v) >= dfs(b) > dfs(a) >= dfs(w)$. The cut-pair is discovered when DFS backtracks to vertex a and before any other edge in the adjacency list of a is traversed. Then at this step of DFS when the tree-edge (a, b) is currently at hand, we mark it as deleted in the adjacency list of a. Using *parent_edge*[b], we can easily mark the same tree-edge as deleted in the adjacency list of b. We then mark the outgoing back-edge (v, w) in the adjacency list of v. This can be accessed using *lpa* of a in Taoka's algorithm [13] or using top of stack of a in Tsin's algorithm [16]. Now the only job left to do is deleting the incoming back-edge (w, v) in the adjacency list of w. At this point of time, *back_cutedge*[v] is set to w. After DFS backtracks from a, it will finally traverse the incoming back-edge (w, v) of w. At that time, the value of *back_cutedge*[v] is checked; if it is w itself then the back-edge (w, v) is marked as deleted in the adjacency list of w. Otherwise, since by Corollary 1 *back_cutedge*[v] is unique, it is guaranteed that the incoming back-edge (w, v) of w cannot be a cut-edge. On the other hand, suppose two tree-edges (x, y) and (a, b) forms a cut-pair such that $dfs(b) \leq dfs(x)$ and the cut-pair is discovered when DFS backtracks to vertex a. Then (a, b) can be marked as deleted in the adjacency list of a. Furthermore, as (x, y) must be a generator and can be accessed from the top of the stack in both Taoka and Tsin algorithms, it can be marked as deleted in the adjacency list of x. The edge (a, b) can be marked as deleted in the adjacency list of b using *parent_edge*[b] while the edge (x, y) can be marked as deleted in the adjacency list of y using *parent_edge*[y]. The lemma thus follows. □

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

Chapter 4

Computation

Recall that a type-2 cut-pair consists of two tree-edges. In the following we give definitions for two kinds of back-edges such that the existence of any of them in a section of T would prevent the generation of type-2 cut-pairs from the section.

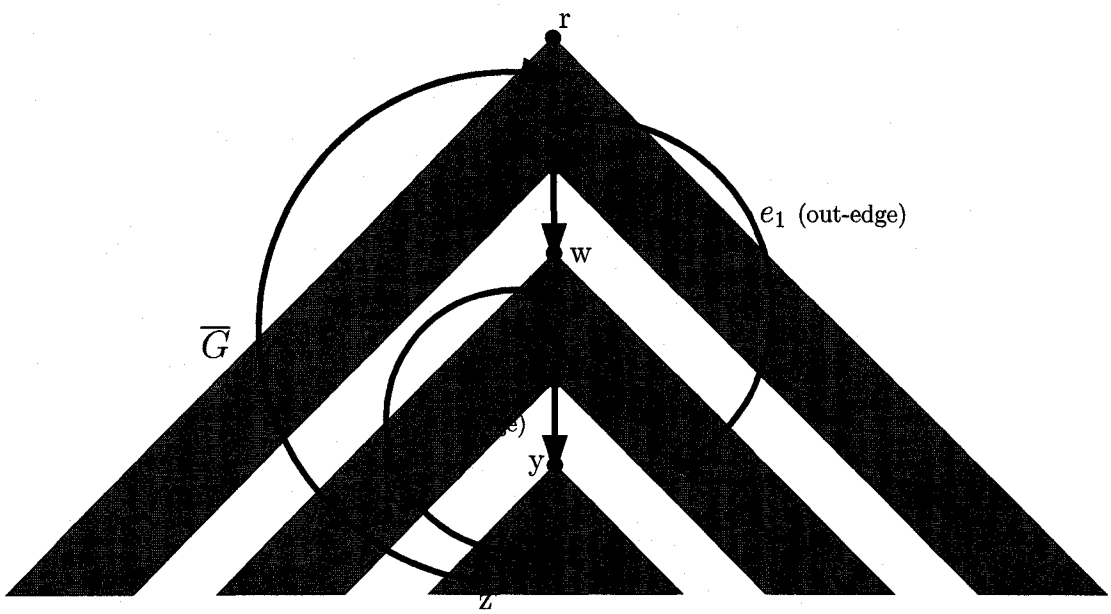


Figure 4.1: e_1 is an out-edge and e_2 is an in-edge with respect to (v, w) and (x, y).

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

In-edge and Out-edge

Suppose $dfs(y) \geq dfs(x) \geq dfs(w) \geq dfs(v)$ and $lowpt(y) \leq lowpt(v)$. A back-edge (u, u') is called an *out-edge* (*in-edge*, respectively) of G (with respect to (v, w) and (x, y)) if $u' \in V_{P_T<r,v>}$ and $u \in V_{T_w} - V_{T_y}$ ($u' \in V_{P_T<w,x>}$ and $u \in V_{T_y}$, respectively).

It is easily seen that with the existence of either an out-edge or an in-edge, $\{(v, w), (x, y)\}$ cannot be a type-2 cut-pair because deleting the two edges does not result in a disconnected graph.

4.1 Taoka et al. algorithm

For clarity we shall call the algorithm as *Taoka* hereafter. The algorithm was reported by Taoka et al. [13] in 1992 and is based on the DFS technique. However, the algorithm accepts only 2-edge connected graphs as inputs. It also executes in three major phases and performs three depth-first searches.

In phase one, during a depth-first search on the given graph, all type-1 cut-pairs are found by computing the parameters, $dfs(v), lowpt(v), medium(v)$, and $lpa(v)$, for every vertex $v \in V$. The definitions for $dfs(v)$ and $lowpt(v)$ were given in Section 1.2. Here we give other definitions for Algorithm Taoka:

$lpa(v) = (a, b)$, *where* $dfs(b) = lowpt(v)$ *and the back−edge* (a, b) *is the one by which* b *is set to* $lowpt(v)$.

$medium(v) = lowpt(v)$ *in* G', *where* $G' = G - lpa(v)$.

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

let $v, w \in V$, whenever the DFS backtracks from w to its parent v, if $lowpt(w) < dfs(w)$ and $medium(w) = dfs(w)$ then (v, w) is a bridge in $G - lpa(w)$ and $\{(v, w), lpa(w)\}$ is a type-1 cut-pair in G.

Moreover, the algorithm assigns a value to each $v \in V$, which is denoted by $path - number(v)$ and is called the *path-partition number* of v, so that the end-points of every cut-pair must be located in a tree path in which all the vertices have the same path-partition number. Specifically, let $v, w \in V$ such that $dfs(v) < dfs(w)$ and $lpa(v) = lpa(w)$ and no proper ancestor, x, of v with $lpa(x) = lpa(v)$ or proper descendant, y, of w with $lpa(y) = lpa(w)$. By the definition of $lpa(w)$, any vertex u lying on $P_T < v, w >$, the tree-path connecting v and w, must have $lpa(u) = lpa(v)$. In other words, every u on $P_T < v, w >$ is assigned the same path-partition number.

Therefore, V can be partitioned into n subsets $V^{(1)}, ..., V^{(n)}$, where $V^{(i)} = \{v | path - number(v) = i\}$ and $V^{(i)} \cap V^{(j)} = \emptyset$, for $i \neq j$. A path PB_i is defined as follows: (let s_i and t_i be the start vertex and end vertex (respectively) of PB_i with $dfs(s_i) < dfs(t_i)$.)

$$PB_i = \begin{cases} \text{the subgraph of } T \text{ induced by } V^{(i)} & \text{if } s_i \in V^{(i)} \\ \text{the subgraph of } T \text{ induced by } V^{(i)} \cup \{s_i\} & \text{if } s_i \text{ is not in } V^{(i)} \end{cases}$$

E_T is partitioned into n subsets $E_{PB_1}, ..., E_{PB_n}$, where $E_{PB_i} \cap E_{PB_j} = \emptyset$, $(i \neq j)$. Also the vertex set of each PB_i is denoted by V_{PB_i}. $\{PB_1, ..., PB_n\}$ is a path-partition of T and is shown in Figure 4.2.

In phase two, two key parameters, *local_min* and *local_high*, which will be used for determining type-2 cut-pairs in the next phase are computed for all the vertices. For any vertex

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

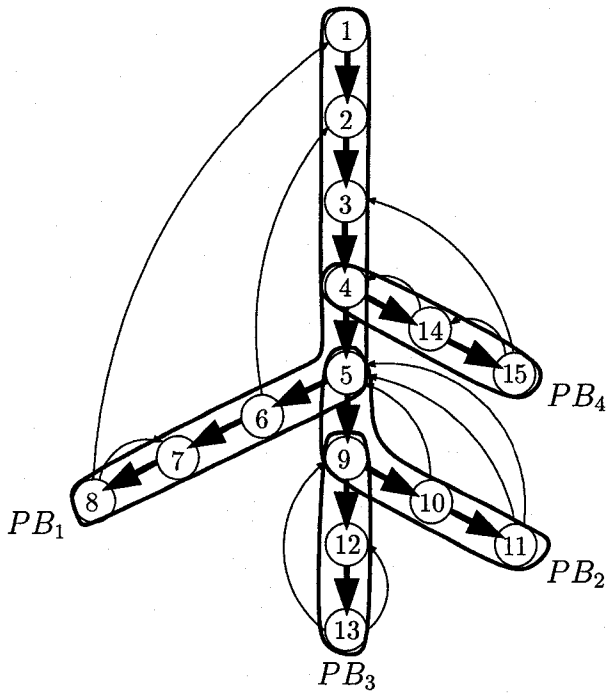


Figure 4.2: A path_partition of T.

$u \in PB_i$, let $T' = T - V_u$, where

$$V_u = \begin{cases} V_{T_v} & if\ u \neq t_i, \\ \emptyset & if\ u = t_i. \end{cases}$$

A path $P_G < u, u' >$ is called a *back-path* of u (with respect to PB_i) if the following conditions (1)-(3) hold:

(1) $u' \in V(P_T < r, u >)$,

(2) any inner vertex of $P_G < u, u' >$ is not in V_{PB_i},

(3) the last edge $< u'', u' >$ of $P_G < u, u' >$ is a back edge and any other edge is in $E_{T'}$.

Note that there might be more than one back-path of u.

$$B_i(u) = \{u\} \cup \{u' | \text{there is a back-path } P_G < u, u' > \text{ of } u \text{ with respect to } PB_i\},$$

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

$$local - min_i(u) = \begin{cases} s_i & if\ minB_i(u) < s_i, \\ minB_i(u) & otherwise, \end{cases}$$

$local - high_i(u) = max\{\{u\} \cup \{u' \in V(P_T < u, t_i >)|$there is a back-path $P_G < u', u >$ of u' with respect to $PB_i\}\}$.

In Figure 4.2, $local - min_1(4) = 3, local - min_1(5) = 5, local - high_1(3) = 4$ and $local - high_1(7) = 8$.

In phase three, the type-2 cut-pairs on every PB_i path are determined. In this phase, through another depth-first search, a stack is manipulated to find type-2 cut-pairs. Each entry on the stack is a pair $\{(x, y), < p, q >\}$ indicating that if there is any edge (v, w) such that $\{(x, y), (v, w)\}$ is a type-2 cut-pair then $dfs(p) \leq dfs(v) < dfs(w) \leq dfs(q)$. (x, y) and $< p, q >$ are called a *candidate generator* and *candidate path* in T, respectively.

Traversing each tree-edge (v, w), where $path - number(v) \neq path - number(w)$ means the DFS starts to visit a PB_i such that v is the starting vertex s_i of BP_i and $w \in V_{PB_i}$. Whenever, the DFS backtracks from w to its parent v, one of the following cases can occur:

- if the edge $(local\_high_i(w), w)$ is determined $(dfs(y) \leq dfs(local\_high_i(w)))$ as an in-edge of G with respect to an edge on $P_T < p, q >$ and (x, y), then the entry is pop-up from the top of the stack since it is guaranteed that the candidate generator (x, y) forms a cut-pair with no edge on $P_T < p, q >$.

- if the edge $(w, local\_min_i(w))$ is determined $(dfs(p) > dfs(local\_min_i(w)))$ as an out-edge of G with respect to an edge on $P_T < p, q >$ and (x, y), then the entry is pop-up from the top of stack since it is guaranteed that the candidate generator (x, y) forms a cut-pair with no edge on $P_T < p, q >$.

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

- if the edge $(w, local\_min_i(w))$ is determined $(dfs(q) > dfs(local\_min_i(w)))$ as an out-edge of G with respect to an edge on $P_T < p, q >$ and (x, y), then the entry (Top Of Stack) is updated to $\{(v, w), < p, local\_min_i(w) >\}$ since it is guaranteed that the candidate generator (x, y) forms a cut-pair with no edge on $P_T < q, local\_min_i(w) >$.

- if $dfs(w) = dfs(q)$ then the candidate generator (x, y) forms a type-2 cut-pair with the edge (v, w) since it is guaranteed that there neither an in-edge nor an out-edge with respect to (v, w) and (x, y). Furthermore, the entry on the top of stack is changed to $\{(x, y), < p, v >\}$ since still the generator (x, y) might form another cut-pair with an edge on $P_T < p, v >$.

- if $local\_min_i(v) > q$ and $v \neq s_i$ then an entry $\{(v, w), < q, local\_min_i(v) >\}$ is pushed to the stack because it is guaranteed that no edge on the path $P_T < local\_min_i(v), v >$ can form a cut-pair with the tree-edge (v, w). However, there might be some edges on $P_T < q, local\_min_i(v) >$ with which, the candidate generator (v, w) forms a type-2 cut-pair.

- if $dfs(v) = s_i$, then the search for type-2 cut-pairs in PB_i has been over and any dummy entry from the top of stack must be popped-up.

During this phase a parameter $virtual\_edge(v)$ for storing an end-point is updated for each vertex $v \in V$. Whenever a type-2 cut-pair $\{(v, w), (x, y)\}$ is found, if the generator (x, y) does not participate in any type-1 cut-pair, then $virtual\_edge(y)$ is updated to v. If the edge (x, y) forms another type-2 cut-pair then $virtual\_edge(v)$ is updated again. This is to take care of some cases in which removing cut-pairs from the given graphs cause some vertices belong to the same 3-edge connected component appear in different connected components. After this phase, the value of $virtual\_edge(v)$ is checked for each vertex v and if it is not empty then an edge having the end-point value and the vertex v as its end-points is added to the given graph.

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

It should be mentioned that the depth-first searches in phase two and three traverse the DFS spanning tree which has been obtained in the first phase, and that back-edges are only used during phase two. As we mentioned before, the algorithm Taoka performs multiple depth-first searches which induces a lot of overhead and lacks elegancy of the following two algorithms which each performs only one DFS.

In algorithm Taoka, the improved edge list structure (see Chapter 3) is used in order to deal with cut-pair deletion. However, in order to represent graphs and perform comparisons among the three algorithms, the model we proposed and explained in the previous chapter will be used in our implementation. Hence our implementation for algorithm Taoka does not follow exactly the same as what they have proposed. As was explained in Chapter 3, the changes result in consuming less time and space. As we know, $lpa(v)$ is a back-edge (a, b) by which $lowpt(v)$ is set as $dfs[b]$. Since we do not have any edge object in our data structure, we do not use $lpa(v)$ for each vertex v. Instead we shall define a simple array called "$start\_of\_lpa[v]$" of size $|V|$ in order to store the starting vertex of $lpa(v)$ (the starting vertex of back-edge (a, b) is the vertex a). Besides for both Taoka algorithm and Tsin [16] algorithms, we have another array called "$to\_low\_pointer[v]$" of size $|V|$. It keeps a pointer to the vertex b in the adjacency list of a which represents the lpa edge (a, b) for each $v \in V$. We also use a pair, $(start\_of\_lpa[v], b)$, for examining and assignments of each $lpa(v)$.

We can reuse the latter array in Procedure Type2 [13] without any initialization in order to keep pointers pointing at generator edges (x, y). It can be used as either $to\_low\_pointer[TOS]$ (Where TOS indicates the top of stack) or $to\_low\_pointer[x]$ because each vertex $x \in V$ can associate with only one of its children in order to form a generator.

Furthermore, there are at most $|V|$ TOSs (The size of stack). Because for each path partition we have a dummy entry, and also for each potential generator we only need one space in

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

the stack (i.e. the candidate is updated in order to compute other type-2 pairs with the same generator). On the other hand each entry has a candidate generator and a candidate path. The length of this path is at least one edge, and also it is not possible that two candidate paths in two different entries have any edge in common. Because $E_{PB_i} \cap E_{PB_j} = \emptyset, (i \neq j)$. Thus, the number of candidate generators in a path partition are less than the number of tree-edges in the same path partition. Suppose the number of candidate generators plus a dummy entry in a path partition is at most the number of tree-edges in the same path partition. Since sum of the number of tree-edges in all path partitions is at most $|V|$, Hence the number of candidates in our stack is at most $|V|$.

In order to find 3-edge connected components of G, after cut-pairs are deleted from G and some necessary edges are added, Procedure DFS_connect (Chapter 1.3) is applied. In the procedure, the DFS finds the vertex set for each connected component of the resulting graph. This can be done in $O(|V| + |E|)$ time.

To determine if a given graph is 3-edge connected or not, the algorithm may go through all Procedures in order to find type-1 and type-2 cut-pairs, and compute path-partition, local-min and local-high parameters. However, sometimes it turns out that the graph is not 3-edge connected just by performing the first DFS. This is because the algorithm finds a type-1 cut-pair during the execution of type-1 Procedure. So the algorithm terminates without completely finishing even the first DFS. Another case in which the graph turns out to be a non 3-edge connected, is when there is no type-1 cut-pair and there is at least one type-2 cut-pair. In this case the algorithm has to perform all DFS's and then terminates when it finds a cut-pair in the last DFS (type-2 Procedure). Otherwise, when there is no cut pair at all, the algorithm has to complete all DFS's to be able to make a decision and report the graph as a 3-edge connected one (Yes instance).

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

4.2 Tsin's without-reduction algorithm(WOR)

For ease of reference, we shall call the Tsin's without-reduction algorithm [16] as *WOR* henceforth. This algorithm was proposed in 2006 and takes care of all cut-pairs by performing only one DFS. The algorithm removes cut-pairs and add some necessary edges in order to find a graph $\mathcal{G}$ in which the vertex sets of its connected components are 3-edge connected components of G.

Tsin used a transformation in which any back-edge (x, y) is replaced by a new tree-edge (x, y') and a new back-edge $(y'y)$, where y' is a new vertex. This results in a new graph $G' = (V', E')$ and its DFS tree $T' = (V', E_{T'})$ such that there is a one-to-one correspondence between the cut-pairs in G and the type-2 cut-pairs in G'. Therefore, instead of finding cut-pairs in G, we can find just the type-2 cut-pairs in G'. However, it should be pointed out that the algorithm does not carry out the transformation explicitly. The new edge and vertex are thus *fictitious*. Therefore, the depth-first search is performed on G rather than G' and the generators are considered with respect to T'. In the following, we shall give some necessary definitions for each $v \in V$ (Note the values of the parameters below are considered to be the same in both T' and T for each v unless otherwise stated):

$$lowpt(v) = min(\{dfs(t) | \exists\ s \in V \text{ such that } s \text{ is a descendant of } v \text{ in } T' \text{ and } (s,t) \text{ is a } back-edge\} \cup \{dfs(v)\})$$

For each fictitious vertex y', $lowpt(y') = y$, where (y, y') is the fictitious back-edge.

Let w be a child of v in T' such that $lowpt(w) = lowpt(v)$. Then

$$2nd - lowpt(v) = min(\{lowpt(w') | w' \text{ is a child of } v \text{ in } T' \text{ and } w \neq w'\})$$

$low(v)$ is the vertex whose DFS-number is $lowpt(v)$.

$2nd - low(v)$ is the vertex whose DFS-number is $2nd - lowpt(v)$.

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

$to - low(v)$ is the first vertex from whom v receives the final $lowpt(v)$ value.

Note that the $to - low(v)$ values are not the same in T and T' when v is a leaf in T.

The WOR algorithm manipulate a $stack[v]$ on which there is an entry $\{(x,y), p \rightsquigarrow q\}$ for each generator (x,y) in the subtree at v such that $p \rightsquigarrow q$ consists of all the cut-edges that could form cut-pairs with (x,y), and is also denoted by $P_T < p,q >$. A similar stack was used in the last DFS of algorithm Taoka. The DFS is applied on G starting from the root, which could be any arbitrary vertex in V. For each vertex $v \in V$, when there is no more incident edge on v to be encountered by the DFS, all of $dfs(v), lowpt(v), 2nd - lowpt(v), low(v), 2nd - low(v)$, and $to - low(v)$ have been finalized. At this point of time, $to - low(v)$ is the first child from whom v receives the $lowpt(v)$ value, where $to - low(v)$ is a fictitious child if the edge $(v, to - low(v))$ is a back-edge in T. By the definitions of generator and in-edge/out-edge, any generator, lying in the subtree at v, which forms a cut-pair with a cut-edge on the $r - v$ tree-path, must be the edgre $(v, to - low(v))$ or lying in the subtree at $to - low(v)$. The stacks of other children are destroyed as will be explained later. Now depending on whether $stack(v)$ is empty, two cases are to be considered separately:

1. If $stack(v)$ is empty and $lowpt(v) \leq 2nd - lowpt(v)$, then the edge $(v, to - low(v))$ is a potential generator. Moreover, if v is a leaf in T, the edge is a back-edge in T and corresponds to the fictitious tree-edge $(v, to - low(v))$ in T'; otherwise it is a tree-edge in both T and T'. By the definitions of generator and in-edge/out-edge, any edge that could form a cut-pair with this edge must lie on the path $low(v) \rightsquigarrow 2nd - lowpt(v)$. Therefore, the entry $\{(v, to - low(v)), low(v) \rightsquigarrow 2nd - low(v)\}$ (with respect to T) is pushed onto the stack of v.

2. If $stack(v)$ is not empty (v cannot be a leaf), let $\{(x,y), p \rightsquigarrow q\}$ be the top entry of the stack. If $dfs(q) < 2nd - lowpt(v)$ and $(v, to - low(v))$ does not form a cut-pair with (x,y), then the edge $(v, to - low(v))$ is a potential generator and an entry

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

$\{(v, to{-}low(v)), q \rightsquigarrow 2nd{-}low(v)\}$ is pushed onto the stack, where $q \rightsquigarrow 2nd{-}low(v)$ includes all the cut-edges that could form cut-pairs with $(v, to{-}low(v))$. Otherwise, all the entries $\{(x, y), p \rightsquigarrow q\}$ on the stack of v satisfying $dfs(p) \geq 2nd - lowpt(v)$ are popped out of $stack(v)$, because none of them could form cut-pairs due to the definition of generator and in-edge/out-edge. If the top entry satisfies $dfs(p) < 2nd{-}lowpt(v) < dfs(q)$, the path $p \rightsquigarrow q$ in the top entry is replaced by $p \rightsquigarrow 2nd - low(v)$ because (x, y) cannot form a cut-pair with any edge lying on the path $P_T < 2nd - low(v), q >$ due to the generator and in-edge/out-edge definitions.

The set of incoming back-edges of v is then examined and any entry $\{(x, y), p \rightsquigarrow q\}$ on $stack(v)$ such that there exists an incoming back-edge (v, u) of v and u is a descendant of y, is popped out of $stack(v)$. This is because none of them could generate cut-pairs due to the generator and in-edge/out-edge definitions. The depth-first search then backtracks to the parent of v.

Here, it should be mentioned that in traversing incident edges on each vertex $v \in V$ whenever the DFS encounters an outgoing back-edge (v, w) of v and $dfs(w) \leq lowpt(v)$ then $stack(v)$ becomes empty because by the definitions of generator and in-edge/out-edge no generator in the subtree of v can form a cut-pair with any edge on $P_T < r, v >$. Furthermore; whenever the DFS backtracks from a child w, (w is a fictitious vertex if the DFS backtracks through a back-edge) all cut-pairs in the subtree at w have been found. If the top entry $\{(x, y), p \rightsquigarrow q\}$ on $stack(w)$ satisfies $q = w$, then the edges (v, w) and (x, y) form a cut-pair. Furthermore (regardless of whether $q = w$), if $lowpt(w) \leq lowpt(v)$, $stack(v)$ is assigned to $stack(w)$ because there might be some generators in the subtree at w that could form cut-pairs with edges on $P_T < r, v >$. Otherwise, $stack(w)$ becomes empty since no generator in the subtree of w can form a cut-pair with any edge on $P_T < r, v >$.

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

When the DFS backtracks to r, all cut-pairs have been determined and deleted from G. Moreover, for every cut-pair $\{(v,w),(x,y)\}$ such that (x,y) is a generator that is not a back-edge, and (v,w) is the tree-edge closest to r that forms a cut-pair with (x,y), a virtual edge (v,y) is added to G. The resulting graph is the desired graph $\mathcal{G}$.

In order to handle stacks, only a stack of size $|V|$ (since there are at most $|V|$ candidate generators) in one time memory allocating is constructed. Also four variables *vtop*, *vbot*, *wtop* and *wbot* are defined; each points to an entry in the constructed stack. They correspond to the top of $stack(v)$, bottom of $stack(v)$, top of $stack(w)$ and bottom of $stack(w)$; respectively. The two variables *wtop* and *wbot* are global while *vtop* and *vbot* are local variables for each DFS call. When v is visited for the first time, *vtop* and *vbot* are initialized to *wtop* and *wbot*; respectively. Each time the DFS backtracks from v to its parent, the assignments of $wtop \leftarrow vtop$ and $wbot \leftarrow vbot$ are performed. Each time $stack(w)$ is get empty, the assignment of $wtop \leftarrow wbot$ is performed. And also each time $stack(v) \leftarrow stack(w)$, the assignments of $vtop \leftarrow wtop$ and $vbot \leftarrow wbot$ are performed.

For determining whether the given graph is 3-edge connected, the algorithm stops once it finds the first cut-pair since it is obvious that the graph is not 3-edge connected, and the graph is reported as a no instance. If there is no cut-pair in the graph then the DFS would run to its completion and the graph is reported as a yes instance.

In order to find the 3-edge connected components of G, after all the cut-pairs are deleted from G and some necessary virtual edges are added, Procedure DFS_connect (Chapter 1.3) is applied on $\mathcal{G}$. In the procedure, the DFS finds the vertex set for each connected component of $\mathcal{G}$. This can be done in $O(|V|+|E|)$ time.

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

4.3 Tsin's with-reduction algorithm(WR)

For simplicity we shall call the algorithm as *WR*. This algorithm was proposed in 2005 and uses an operation called *absorb-eject* to gradually transform the given graph G into a null graph by performing only one depth-first search over the given graph. Each vertex in the resulting graph corresponds to a 3-edge connected component as it has absorbed all other vertices in the same 3-edge connected component. Hence, by keeping track of all the vertices of G absorbed by each vertex of the null graph, the set of all 3-edge connected components of G is computed. In contrast with the other two algorithms, algorithm WR does not look for cut-pairs and it directly computes all the 3-edge connected components without any further step after the DFS terminates.

It should be mentioned that algorithm WR constructs link lists of all back-edges (other than adjacency lists) and searches through them for back-edges that have become self-loops, in order to determine whether $deg(u) = 2, u \in V$, at the time the DFS backtracks from vertex u to its parent. This part of the algorithm takes $O(|E|)$ time and space. We shall show that the determination of $deg(u)$ can be done in $O(|V|)$ time and space, and the construction of such lists can be avoided. However, since the DFS traverses adjacency lists, the complexity of the entire algorithm remains as $O(|V| + |E|)$.

Each time an absorb-eject operation is applied on an edge $e = (w, u)$ at vertex w in a graph G', both the vertex u and the edge e are absorbed by the vertex w. At this time, if $deg(u)$ in G' is 2, vertex w throws away vertex u in order to make it an isolated vertex in the final null graph. Regardless of whether $deg(u) = 2$, all the remaining edges incident on vertex u become incident edges on vertex w where any resulting self-loop is deleted.

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

Absorb-eject operation

Let $G' = (V', E')$ and $e = (w, u) \in E'$ such that either (i) $deg_{G'}(u) = 2$, or (ii) e is not a cut-edge. The graph obtained from G' by applying an absorb-eject operation on e at w is the graph $G'/e = (V'', E'')$ such that $E'' = E' - E_u \cup E_{w+}$, where E_u is the set of edges incident on u in G' and $E_{w+} = \{f' = (w, z) | \exists f \in E_u,$ *such that* $f = (u, z)$ *for some* $z \in V' - \{w\}\}$, and

$$V'' = \begin{cases} V' & \text{if } deg_{G'}(u) = 2, \\ V' - \{u\} & \text{if } e \text{ is not a cut-edge.} \end{cases}$$

All possible cases of the absorb-eject operation is shown in Figure 4.3. In case (i) vertex u becomes an isolated vertex in G'/e. In case (ii) vertex w absorbs vertex u. In both cases the edge e is absorbed by vertex w. In case (ii), since e is not a cut-edge therefore $deg_{G'}(u) \neq 2$; this clearly shows that each of these two cases is distinct.

An *embodiment* of an edge f is the edge f itself, or the edge $f' = (w, z) \in E_{w+}$ such that $f = (u, z) \in E_u$, or an embodiment of an embodiment of f. For each $w \in V'$, let $\sigma(w) = \{w\}$ initially, and let $\sigma(w) = \sigma(w) \cup \sigma(u)$ when vertex w absorbs vertex u. Clearly, $\sigma(w)$ denotes the set consisting of vertex w and all the vertices that have been absorbed either by vertex w or by vertices that have been absorbed by vertex w.

It should be mentioned that when the absorb-eject operation is applied on a tree-edge $e = (w, u)$ at vertex w, the types of the edges incident on u are kept so that if f is a tree-edge (back-edge, respectively) in G', then its embodiment f' is a tree-edge (back-edge, respectively) in G'/e as well.

A depth-first search is applied by the algorithm on G starting at a vertex r, which can

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

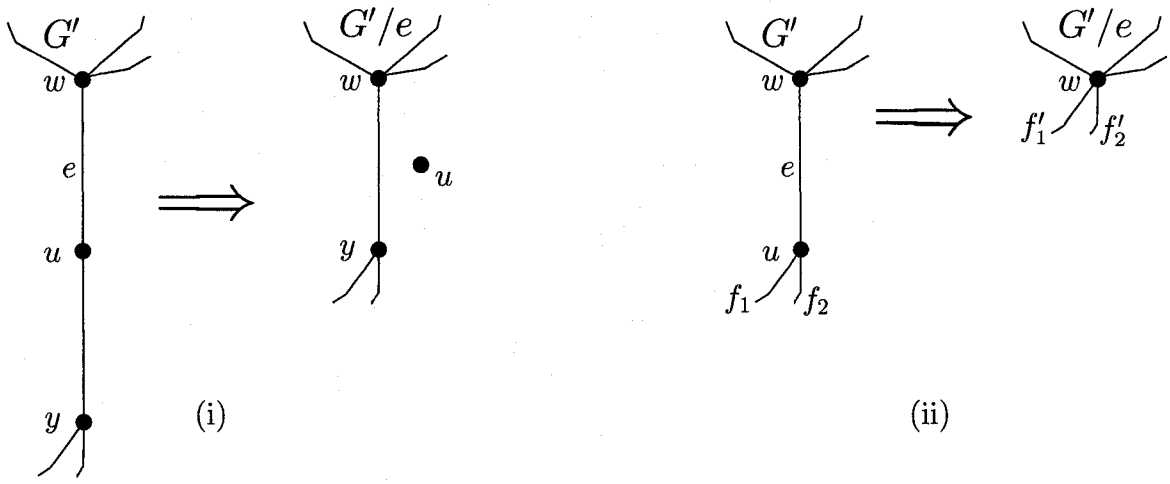


Figure 4.3: (i)$deg'_G(u) = 2$ and e is a cut-edge. (ii)e is a not a cut-edge

be any vertex in V. For each tree-edge in T, during the DFS the absorb-eject operation is applied to either the tree-edge or an embodiment of the tree-edge. Each time the absorb-eject operation is applied, either condition (i) or condition (ii) is satisfied. The purpose of condition (ii) is to collect vertices belonging to the same 3-edge connected component under one vertex because the two end-points of edge e which is not a cut-edge, are in the same 3-edge connected component. When all the vertices in a 3-edge connected component have been gathered under a vertex u, $deg_{G'}(u)$ becomes 2, where G' is the graph to which G has been transformed. When the DFS backtracks to the parent of vertex u, Condition (i) is satisfied. Therefore, the absorb-eject operation is applied at the parent, to absorb the tree-edge e and isolate u from the graph. This is because the two edges incident on u form a cut-pair such that $\sigma(u)$ is a 3-edge connected component of G.

When the DFS backtracks from vertex w to its parent, the subtree at w in T, has been transformed into a graph consisting of a set of isolated vertices and a tree-path P_w associating with some back-edges, called the $w - path$. Each back-edge associated with P_w has one of its end-point on the w-path and the other on the $r - w$ tree-path. The w-path is denoted by $P_w : (w =)w_0 - w_1 - w_2 - \ldots - w_k$, where $w_0, w_1 \ldots w_k$ are the vertices on the path and w_0, w_k are the end vertices of the path. Furthermore, there exists a back-edge

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

$f = (w_k, x)$ where $pre(x) = lowpt(w)$, and there is no back-edge connecting any two vertices on the w-path. Each isolated vertex corresponds to a 3-edge connected component of G. The tree-edge (w, w_1) on the w-path is a candidate in order to form a cut-pair with an edge lying on the $r - w$ tree-path.

The determination of the w-path is as the following. When the DFS visits vertex w for the first time, the $w - path$ is initialized to a null path. Whenever the DFS backtracks from a child u such that $lowpt(u)$ is smaller than the current value of $lowpt(w)$ or encounters a back-edge (w, u) with $dfs(u)$ smaller than the current value of $lowpt(w)$, then the w-path is updated. In the former case, the w-path becomes the path consisting of the tree-edge (w, u) and the u-path. In the latter case, it becomes the null path. The absorb-eject operation is then applied at vertex w to absorb all the edges on the previous w-path because none of the edges on the path can form a cut-pair with any edge on $r - w$ path. In other words, all the vertices on it including w belong to the same 3-edge connected component.

In Figure 4.4(a) a graph G is shown with solid lines denoting tree-edges and dotted arrows denoting back-edges. In Figure 4.4(b) the graph to which G has been transformed when the DFS backtracks from vertex u to vertex w is shown. Note that the current w-path is $w - w_1 - w_2 - w_3$ and the u-path is $u - a - b$. In Figure 4.4(c) the current w-path is updated to u-path $w - a - b$ while the previous w-path $w - w_1 - w_2 - w_3$ is absorbed by vertex w; this is because $lowpt(u) < lowpt(w)$. Moreover, since $deg(u) = 2$, an absorb-eject operation was applied on the edge e at w (Figure 4.4(b)). When the DFS encounters an incoming back-edge $f' = (w, u)$ that is not a self-loop at vertex w, the back-edge f' must be an embodiment of an incoming back-edge $f = (w, u')$ of w in T. Since u' is a descendant of w in T, u' must be located in a subtree of at a child of w in T. The DFS has traversed the subtree and consequently the subtree must have been transformed into a tree-path and a set of isolated vertices such that vertex u has absorbed vertex u'. Vertex u cannot be any of the isolated

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

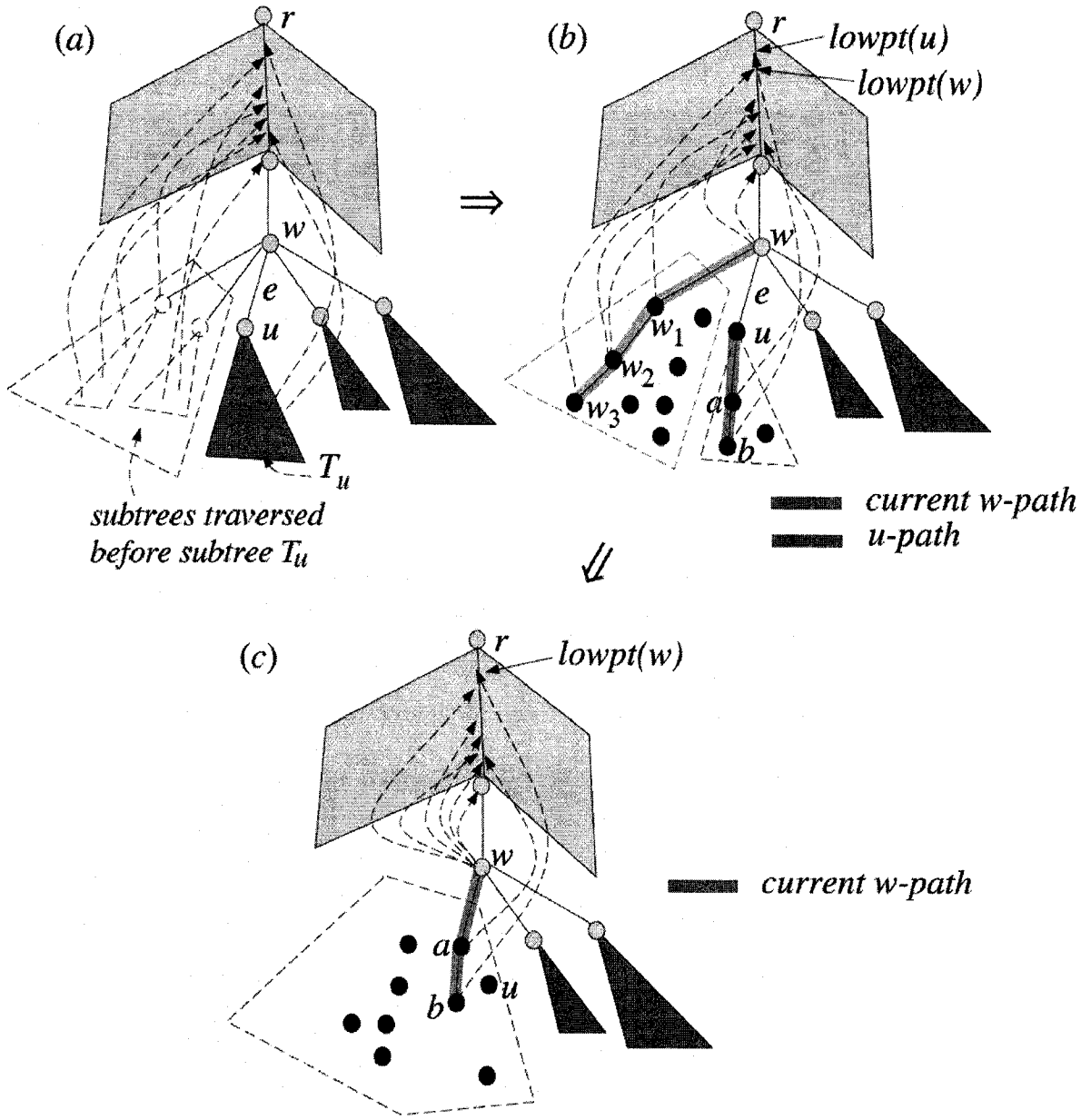


Figure 4.4: When DFS backtracks from vertex u to vertex w. (Extracted from [15])

vertices because of the existence of the edge f'. Hence, It must be located on the tree-path. The tree-path either is the current w-path or must have been absorbed by vertex w earlier. In the latter case $u = w$ which means f' is a self-loop. In the former case, vertex u must be a vertex located on the current (non-null) w-path. The absorb-eject operation is then applied at w to absorb the section of the current w-path from w to u (see Figure 4.5). This

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

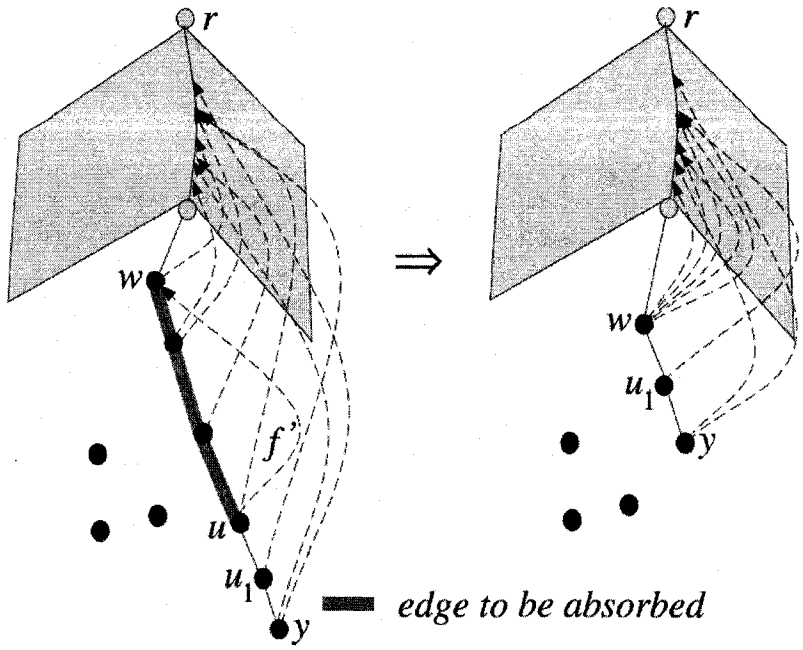


Figure 4.5: When an incoming back-edge is encountered. (Extracted from [15])

is because no edge lying on this section could generate new cut-pairs. Note that at this step the operation is applied to absorb each tree-edge (w, x) on P_w if it lies on $P[w..u]$ (Lemma 9 of [15]). The edge lies on $P[w..u]$ if x is an ancestor of u' in T. The final w-path is determined when the search backtracks to the parent of vertex w.

The DFS eventually backtracks to the root r from a vertex u, which is the last child of r. At this time, the graph G has been transformed to a graph consisting of a set of isolated vertices and the path $r + P_u$, where $+$ is the concatenation operator. If $deg(u) = 2$, an absorb-eject operation is performed on the tree-edge (r, u) at r. Then r absorbs the path $r + P_u$ because $lowpt(r) = 1$. This results a set of isolated vertices each represents a 3-edge connected component of G.

To determine whether a given graph is 3-edge connected, the DFS stops once it finds the first vertex with degree 2, and the graph is reported as non-3-edge connected (i.e. a 'no' instance). On the other hand, if the algorithm does not find any vertex with degree 2, then

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

the DFS would run to its completion, and the graph is reported as 3-edge connected (i.e. a 'yes' instance).

Lemma 3: [Lemma 6(ii) of [15]] *Let $u \in V - \{r\}$. At the time the DFS backtracks from u to its parent w, $P_u : (u =)u_0 - u_1 - u_2 - \ldots - u_k$ such that for each back-edge $f = (u_i, x)$, $0 \leq i \leq k$, x lies on the $r - w$ tree-path.*

During the execution, when the DFS backtracks from w the absorb-eject operation has been applied on n_w tree-edges, where $0 \leq n_w \leq |E_{T_w}|$. Note that each of the tree-edges is either a tree-edge in T_w or an embodiment of a tree-edge in T_w. Each time the operation is applied we have a set $a_i, 1 \leq i \leq n_w$, consisting of all the parallel back-edges between w and one of its proper descendants, $v_i, 1 \leq i \leq n_w$. We define $A_w = \bigcup_{i=1}^{n_w} a_i$.

Lemma 4: *The number of incomming back-edges of w is $|A_w|$.*

Proof: The case where $n_w = 0$ is obvious.

Let $n_w \geq 1$. Suppose when the DFS backtracks from w, each incoming back-edge (v_i, w) of vertex w has the end-point v_i lying on the same P_x path, where P_x is $w(= x_0) - x_1 - x_2 - \ldots - x_{n_w}$. Regardless of whether P_x is P_w or $w + P_u$, the absorb-eject operation absorbs the vertices x_i along the path. Let b_i be the set of parallel back-edges between w and v_i. By Lemma 3, any back-edge with one end-point x_i has the other end-point on the $r - w$ path. Therefore no back-edge can have v_i and v_j as its end-points, $1 \leq i, j \leq n_w$ and $i \neq j$. Consequently, no incoming back-edge of w can be added by applying the absorb-eject operation. As a result, $b_i \cap b_j = \emptyset$. It follows that $|\bigcup_{i=1}^{n_w} b_i| = |\bigcup_{i=1}^{n_w} a_i| = |A_w|$. Hence, $|A_w|$ is the number of incoming back-edges of w (whose other end-points lies on P_x).

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

Since each incoming back-edge of w is an edge (v_i, w) such that v_i lies on a $P_{w_j}, 1 \leq j \leq k$, where $k \leq n_w$ (note that P_{w_j} is either P_w or $w + P_u$), it follows that the set of all incoming back-edges of w can be partitioned into a collection of disjoint subsets $\mathcal{C}_1 = \{S_j | 1 \leq j \leq k\}$ such that there is a one-to-one correspondence between $\mathcal{C}_1$ and the collection $\{\{(v_i, w) | v_i \textit{ lies on } P_{w_j}\} | 1 \leq j \leq k\}$. From the preceding paragraph, for each $P_{w_j}, 1 \leq j \leq k$, $|A_{w_j}|$ is the number of incoming back-edges of w whose other end-points lies on P_{w_j}. It follows that $|A_w| = \sum_{j=1}^{k} |A_{w_j}| = \sum_{j=1}^{k} |S_j|$. As $\sum_{j=1}^{k} |S_j|$ is the total number of incoming back-edges of w, the lemma thus follows. □

Lemma 5: *Let $u, w \in V$ such w is the parent of u and G' be the graph to which G has been transformed before the absorb-eject operation is applied on the tree edge $e = (w, u)$ at w. Then after the operation, $deg[w] = deg'[w] + deg[u] - 2 - 2k$, where $deg'[w]$ is the degree of w before the operation is performed and k is the number of parallel back-edges between w and u in G'.*

Proof: Let k be the number of parallel edges between w and u. It is easily verified that $0 \leq k \leq |B|$. Let k' be the number of outgoing back-edges whose other end-point is not w. Clearly $k' = deg[u] - k - 1$. As the absorb-eject operation is performed, these k' edges become incident outing back-edges of w. So k' must be considered in updating $deg[w]$ in G'/e. However $deg[w]$ must also be reduced by both 1 and k because w has lost the tree-edge e and those k edges had become self-loops which should be disregarded. Therefore, $deg[w]$ becomes $deg[w] + k' - 1 - k$. Hence $deg[w]$ becomes $deg[w] + deg[u] - 2k - 2$. The lemma thus follows. □

Corollary 3: When the DFS backtracks from vertex w to its parent, $deg[w] = deg'[w] + \sum_{i=1}^{n_w} deg_{G_i}(v_i) - 2\sum_{i=1}^{n_w} k_i - 2n_w$, where $deg'[w]$ is the degree of w before the first adsorb-eject operation is applied to it, G_i is the graph to which G has been transformed when the

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

adsorb-eject operation is performed on (w, v_i) and k_i is the number of parallel back-edges between w and v_i in G_i.

Proof: Immediate from Lemma 5. □

Now, instead of spending $O(|E|)$ time and space by using the way mentioned in Lemma 11 of [15] for determining if $deg[u] = 2$ in Section 1.1 of the algorithm, we can spend $O(|V|)$ time and space on the degree determination if we first initialize the degree of all vertices to zero and then add 3 statements (Each takes $O(1)$ time) to the algorithm as explained below:

- Firstly, in Section 1 of the algorithm, we insert the statement $deg[w] \leftarrow deg[w] + 1$. So that when the DFS visits vertex w for the first time the statement is also executed. This is to take every edge incident on w into account. Note that $deg[w]$ is initialized to zero.

  Secondly, we add the statement $deg[w] \leftarrow deg[w] + deg[u] - 2$ in Procedure Absorb-path within the **for** loop. So that any time the absorb-eject operation is applied by calling the Procedure, the statement is also executed. Since the absorb-eject operation is applied n_w times in total for vertex w, a total of $\sum_{i=1}^{n_w} deg_{G_i}(v_i)$ is added and a total of $2n_w$ is subtracted in computing the final value of $deg[w]$. This takes care of two of the terms in the formula of Corollary 3.

  Lastly, we add the statement $deg[w] \leftarrow deg[w]-2$ in the **then** part of Statement 1.6.0. of the algorithm. So that each time the DFS encounters an incoming back-edge the statement is also executed. From the preceding paragraph, it remains to deduct $2\sum_{i=1}^{n} k_i$ from $deg[w]$ in order to calculate the final value of $deg[w]$. By Lemma 4, the size of A_w is the number of incoming back-edges of w. Since $\sum_{i=1}^{n_w} k_i = \sum_{i=1}^{n_w} |a_i| = |A_w|$, we thus have $\sum_{i=1}^{n_w} k_i$ is the number of incoming back-edges of w. As the newly added statement is executed whenever an incoming back-edge is encountered, a total of $2\sum_{i=1}^{n_w} k_i$ is thus deducted in computing the final value of $deg(w)$. Hence, by Corollary 3, $deg[w]$

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

is correctly computed.

- It is unnecessarily to add the second statement mentioned above to the **then** part of Statement 1.1 of the algorithm; although the absorb-eject operation is applied. This is if the statement was added, then if $deg[u] = 2$ when the DFS backtracks from u to w, $deg[w]$ would be updated to $deg[w] + deg[u] - 2 = deg[w] + 2 - 2$ which is $deg[w]$ itself! To be more specific, if P_u is not null, then $deg[w]$ needs no update because by applying the operation, vertex w looses the tree edge (w, u) but on the other hand gains the first tree-edge on P_u. If P_u is Null, then vertex w looses the tree-edge (w, u) but gains the only back-edge on u.

- We shall verify the spending time and space on the degree determination in Section 1.1 using this method. We note that we need only $|V|$ memory space for this purpose since we only need a simple array $deg[w]$ of size $|V|$. Furthermore, the logical **if** expression in Section 1.1 takes a total of $O(|V|)$ time to determine the degree for all the vertices in $V - \{r\}$; since only the value of $deg[w]$ must be checked for each w.

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

Chapter 5

Comparison

We performed three types of experiment to compare the execution times of the three algorithms. They are: testing for 3-edge connectivity, determining cut-pairs, and computing 3-edge connected components. The platform we used for the experiments is as below:

- Hardware:
  - Model: Dell 650 Precision Workstation
  - Processor: Pentium 4, 3.2GHz Intel/Xeon, 512KB L2 cache
  - Memory: 4GB
- Software:
  - Operating System: Linux, Fedora core 2.6.12
  - Programming Language: C

5.1 Data Set

In order to perform experiments and show execution times of each algorithm, we generated two different sets of graphs using a random graph generator. Since the algorithm of Taoka

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

et al. accepts only 2-edge connected graphs, all the graphs in the two sets are thus 2-edge connected. The first set S_1 consists of 305 sparse graphs such that $\forall G = (V, E) \in S_1, 12.5 \leq \frac{|E|}{|V|} \leq 16.5$. The input sizes ($|E| + |V|$) of the graphs in S_1 are distinct and are randomly chosen from the interval of $[600,000..60,000,000]$.

On the other hand, the second set S_2 consists of 301 dense graphs where $\forall G \in S_2, \frac{|E|}{|V|} \simeq \frac{|V|^2}{4}$. The input sizes of the graphs in S_2 are distinct and are randomly chosen from the interval of $[6,000,000..110,000,000]$.

Moreover, after performing an experiment for finding 3-edge connected components on the sets, it turned out that 180 and 201 graphs are 3-edge connected in the first set and second set, respectively. Note that since the graphs are randomly generated, we might not have enough number of 3-edge-connected graphs or non-3-edge connected graphs in our sets. In the following we will explain how to obtain the desired number of 3-edge-connected graphs. So we can run the experiment for determining 3-edge-connected graphs, while we have a fair number of Yes and No instances.

5.1.1 Generating the experimental input graphs

Starting with an edgeless graph $G = (V, E)$ and a number t (a threshold), we visit all vertices in V. Each time we visit a vertex v, we generate a random number *Enum*, where $1 \leq Enum \leq t$. The t parameter indicates that at most how many edges are added to the adjacency list of v while *Enum* is the actual number of edges added by the time we visit v. Having a larger value for t results in generating a graph with higher density. To generate each of the *Enum* edges for v, we generate a random vertex number, w, in the interval $[1..|V|]$. If $w = v$, we generate w again since we are not interested in self-loops. We then add an edge (v, w) to G by adding v to the adjacency list of w, and w to the adjacency list of v.

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

5.1.2 Generating connected graphs

After we create a random graph $G = (V, E)$, if the graph is not connected, we will turn it into a connected graph. This can be easily done by finding the connected components and then adding edges to make the graph connected. We find the connected components of G using the algorithm explained in [17]. The details are given below:

```
1   c = 0;
2   S = ∅;
3   for (i=1;i<=Vnum;i++)
4     visited[i] == 0
5   for (i=1;i<=Vnum;i++)
6     if (visited[i] == 0) {
7       c = c + 1;
8       s_c = ∅;
9       DFS_connect(i);//Procedure DFS_connect (Chapter 1.3) is called to compute s_c.
10      j = a random vertex in s_c;
11      S = S ∪ s_c
12      k = a random vertex in S;
13      if (i != 1) {  //there is more than one connected component.
14        add j to the list of k in a random position;
15        add k to the list of j in a random position;
16      }
17    }
```

5.1.3 Generating 2-edge connected graphs

After we generated a connected graph we have to turn it into a 2-edge connected one. We do this through a DFS procedure. Let $w, v \in V$. Each time the DFS backtracks from a child

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

w to its parent v, if $lowpt(w) = pre(w)$ then the edge (v, w) is a bridge. Therefore we add an edge $(low(v), node)$ where $node$ is either a child of w or w itself (in case w has no child). Note that $low(v)$ is its value at the time the DFS backtracks from w.

5.1.4 Generating non-3-edge connected graphs

We had observed that when the density is high, the graph we generated was always 3-edge connected. Obviously, it is desirable to generate non-3-edge connected graphs for the algorithms. In this case, we have the random graph generator to choose a number c and generate c different graphs, where the total number of vertices of the graphs is $|V|$. The set of vertices for each graph, which is 3-edge connected, is denoted by $s_i, 1 \leq i \leq c$. Then we connect the graphs to each other as below:

```
1   S = ∅;
2   for (i=1;i<=c;i++) {
3   S = S ∪ s_i;
4     for (j=1;j<=2;j++) {
5       v1 = a random vertex in s_i;
6       v2 = a random vertex in S;
7       add v2 to the list of v1 in a random position;
8       add v1 to the list of v2 in a random position;
9     {
10  {
```


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

5.2 Result

We performed experiments to show the execution time of each algorithm for roughly 600 random 2-edge connected graphs. The results are depicted in Figures 5.1 to 5.10. The plots in Figures 5.1 to 5.5 are results of the experiments on the first set S_1; the one whose graphs are of lower densities. The plots in the remaining figures are results for the second set S_2; the one whose graphs are of higher densities. Moreover, the plots in Figures 5.1 and 5.6 are results of the experiment that computes 3-edge connected components, and the plots in the remaining figures are results for finding cut-pairs and also determining whether the input graph is 3-edge connected.

Looking at the plots, we observe that Taoka has almost the longest execution time in the experiments. As we described the algorithm before, this was expected since the algorithm performs different depth-first searches in multiple phases. In some cases the algorithm determines *No* instances faster than the others (See Figure 5.4 and 5.9). This is because the algorithm is able to find type-1 cut-pairs in the first DFS, so it stops whenever it finds the first type-1 cut-pair. On the other hand in Figure 5.4 we observe that the algorithm is remarkably slow in some cases because in these cases there is no type-1 cut-pair so the algorithm eventually finds a type-2 cut-pair in the last phase. Overall, it is hard to say which algorithm is good in determining *No* instances, because it depends on when an algorithm can find the first cut-pair or 3-edge connected component.

Looking at Figure 5.6, after the input size goes beyond 6.6×10^7, the WR algorithm starts to collapse as it runs out of memory on the platform; while the other algorithms including the modified version of WR, which spends $O(|V|)$ time and space on degree determination of vertices, still run.

WR terminates successfully on the first four graphs after the input size goes beyond

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

6.6×10^7 (Figure 5.6). This is because these four graphs are not 3-edge connected and their sizes are close to 6.6×10^7. When a vertex is determined to be of degree 2, all the outgoing back-edges of all the vertices in the same 3-edge connected component with the vertex have become self-loops, with the exception of at most one of them. So the freed up memory space of the self-loops result in enough space to allow the algorithm to completely find the next 3-edge connected component. This is the case until all the 3-edge connected components are found and the algorithm terminates execution. However, the freed up memory space of the self-loops could not help the algorithm to run to its completion for the other graphs (69 out of 301) which have input sizes beyond 6.6×10^7. Hence, the modified version of WR increases the largest input graph size it could handle by 34% while being the fastest among all the algorithms.

As we can see in the figures in the chapter, the challenge is between WOR and WR. This is because Taoka performs different depth-first searches in multiple phases while the others each performs only one depth-first search. In computing 3-edge connected components WOR searches the adjacency lists 3 times while WR searches adjacency lists and back-edge lists each once. In Figure 5.1 we also see that WR is faster than WOR in computing 3-edge connected components.

In determining *Yes* instances (3-edge connected graphs) WOR does not have to search adjacency lists in order to find connected components. Therefore, its performance becomes better. This is clearly shown In Figure 5.3.

The modified version of WR has the best performance since it does not need to construct and search back-edge lists for degree determination of vertices. As explained in chapter 4.3 instead of spending $O(|E|)$ time and space on this part, the modified version of WR use a different method to do the same which takes a total of $O(|V|)$ time and space.

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

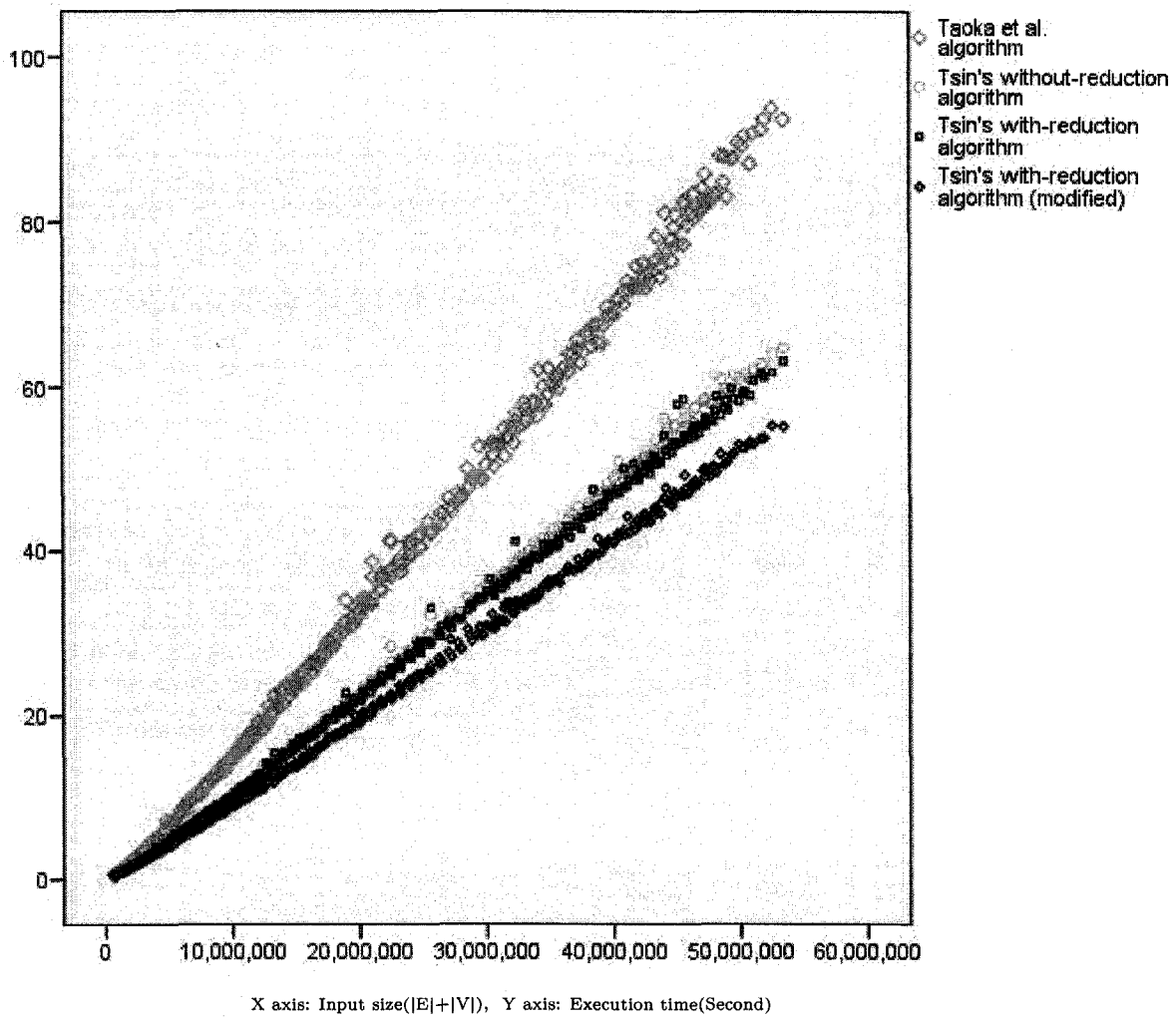


Figure 5.1: *Serach for* $3-$*edge connected components*; $12.5 \leq \frac{|E|}{|V|} \leq 16.5$

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

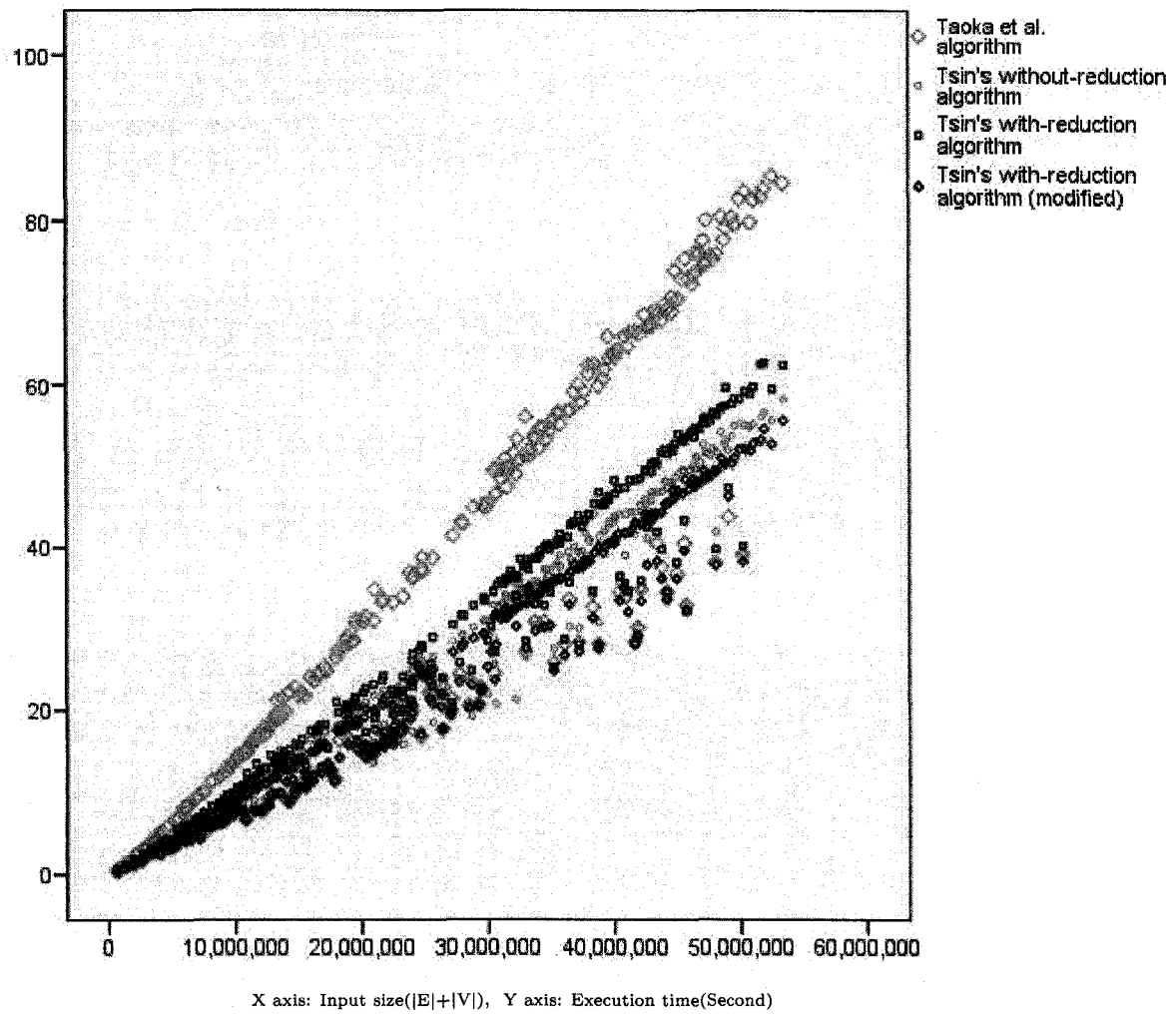


Figure 5.2: *Determining whether graphs are* $3-$*edge connected or NOT*; $12.5 \leq \frac{|E|}{|V|} \leq 16.5$

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

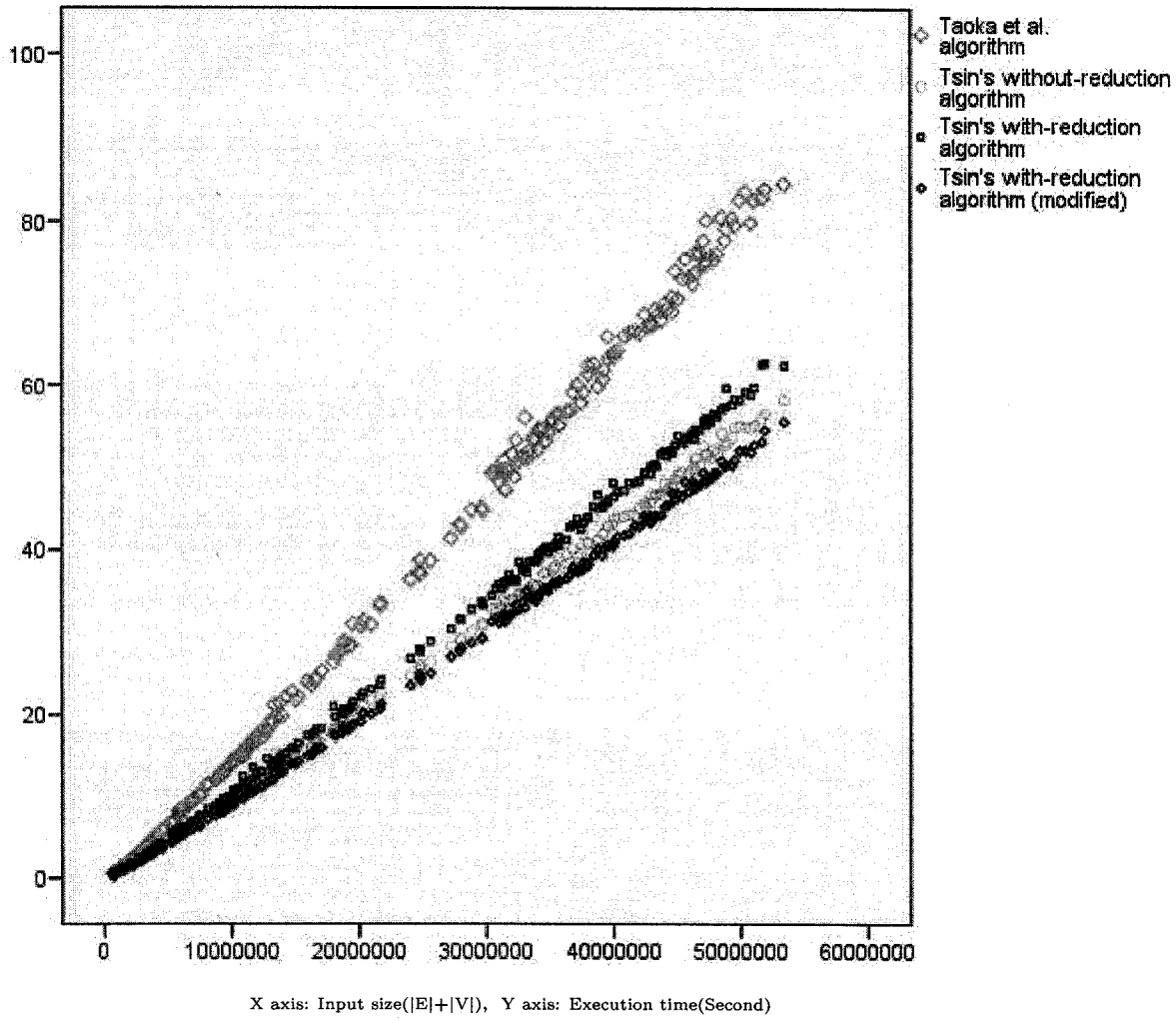


Figure 5.3: *Determining* $3-$ *edge connectivity*; *only Yes instances*; $12.5 \leq \frac{|E|}{|V|} \leq 16.5$

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

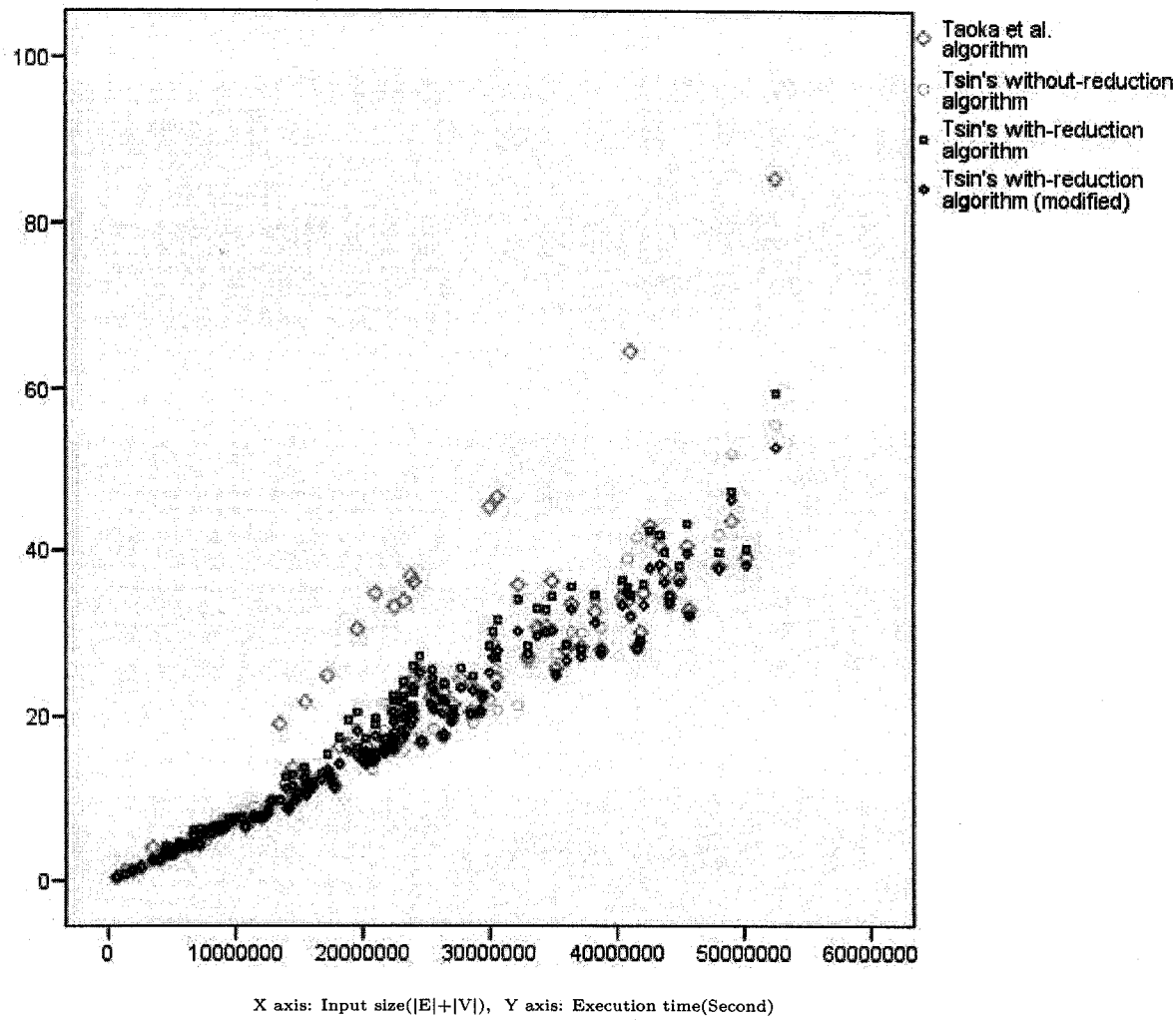


Figure 5.4: *Determining* $3-$ *edge connectivity*; *only No instances*; $12.5 \leq \frac{|E|}{|V|} \leq 16.5$

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

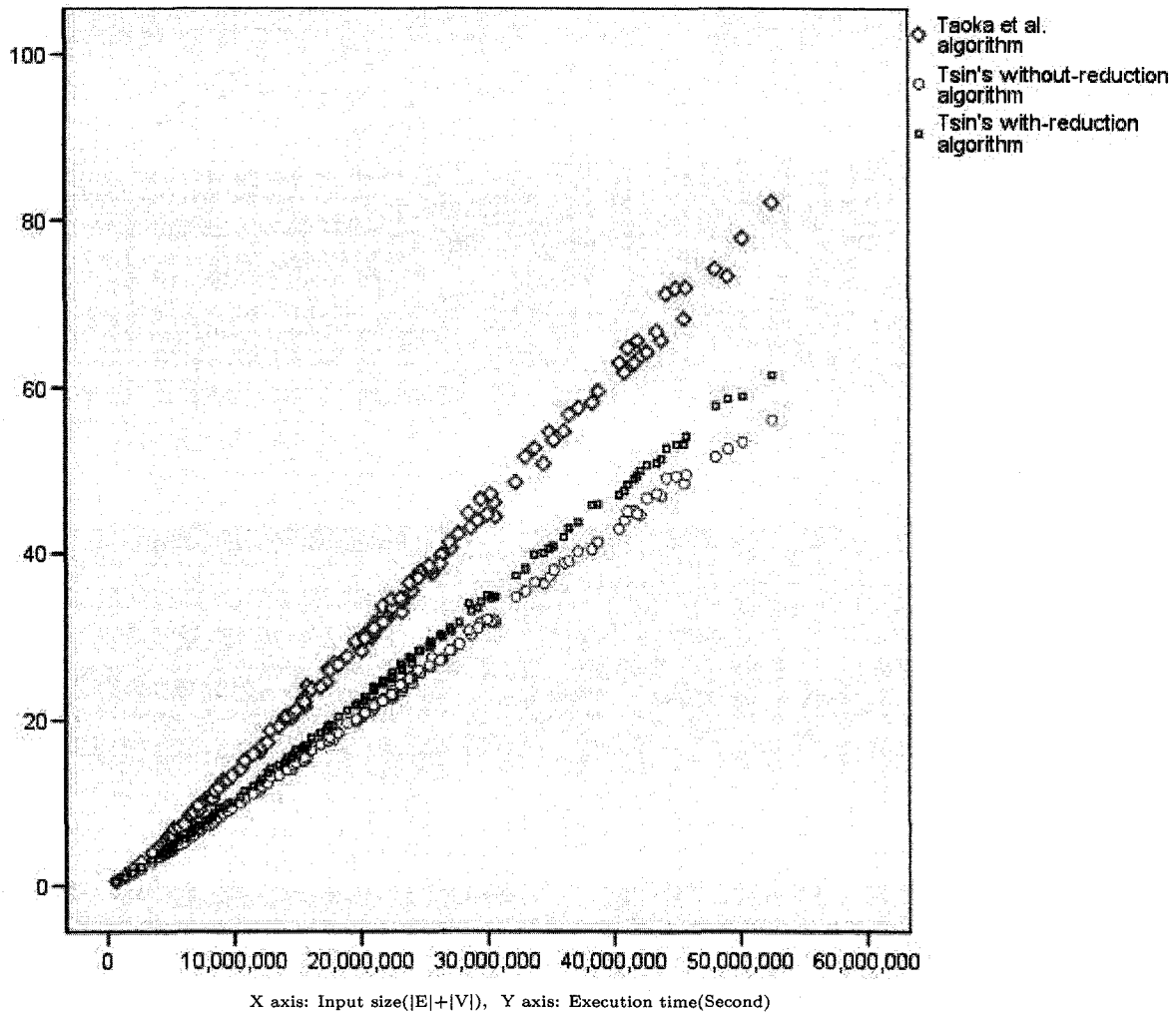
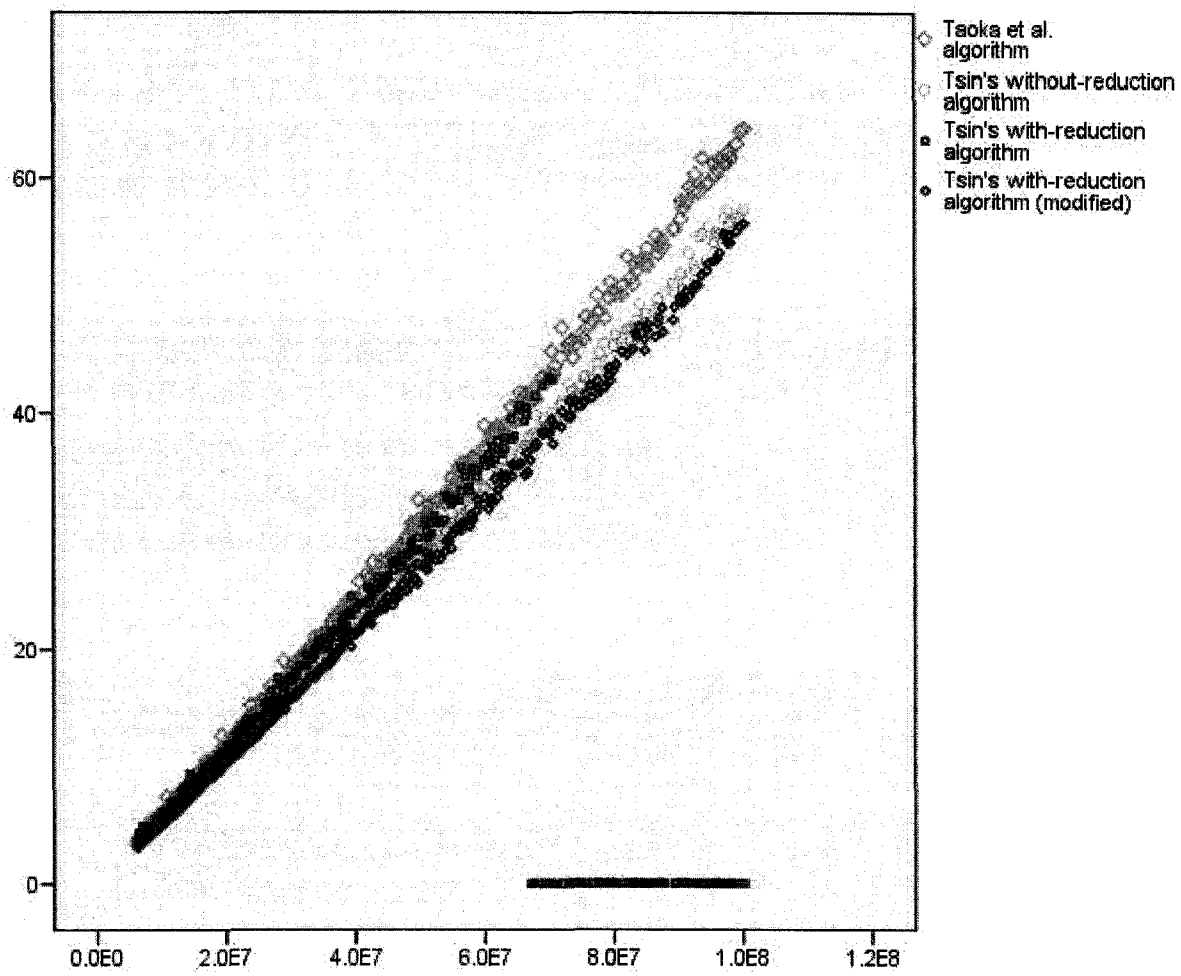


Figure 5.5: *Determining cut − pairs on No instances*; $12.5 \leq \frac{|E|}{|V|} \leq 16.5$

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



X axis: Input size($\simeq |E|$), Y axis: Execution time(Second)

Figure 5.6: *Serach for* $3-$*edge connected components*; $|E| \simeq \frac{|V|^2}{4}$

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

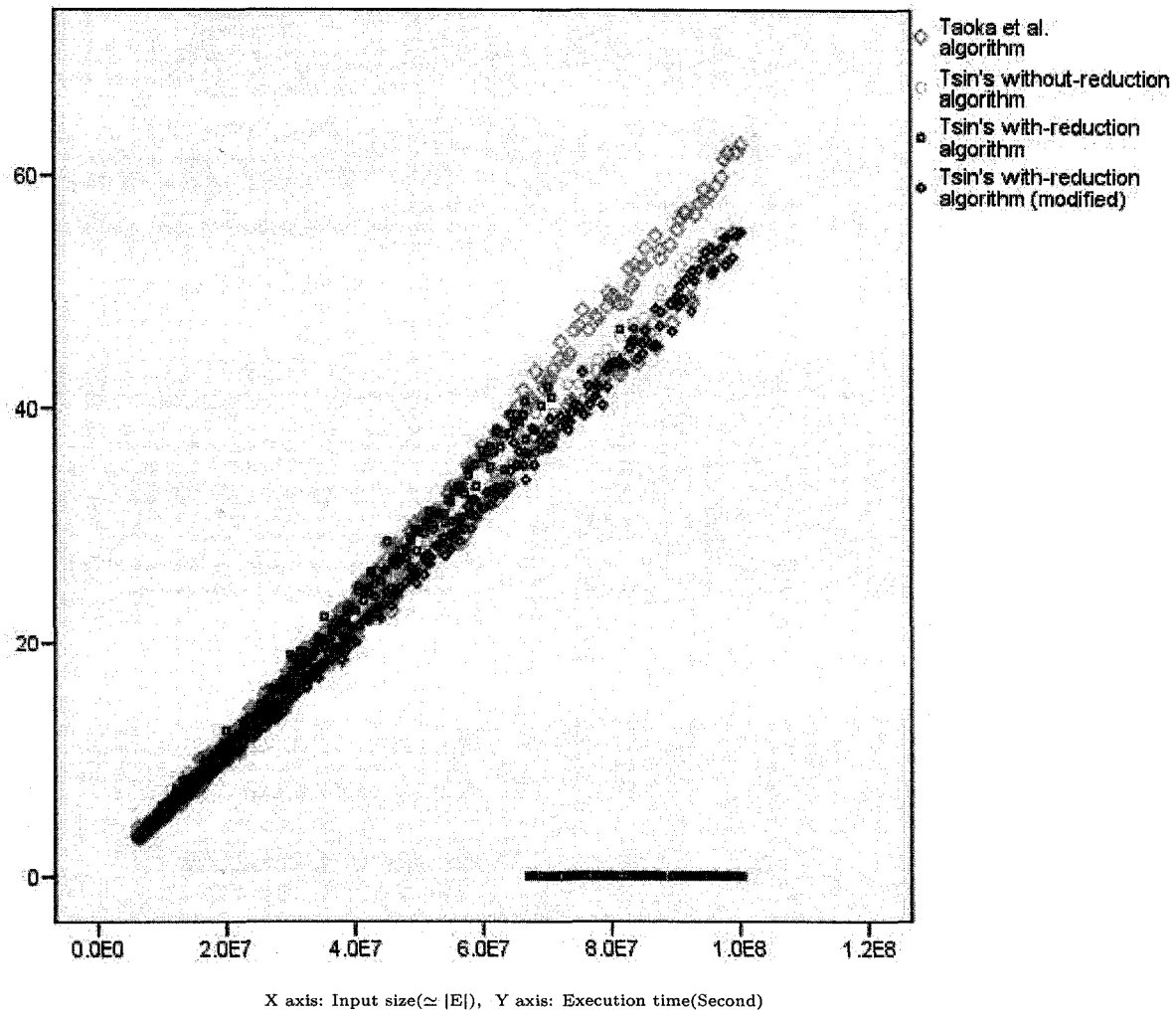


Figure 5.7: *Determining whether graphs are* $3-$*edge connected or NOT*; $|E| \simeq \frac{|V|^2}{4}$

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

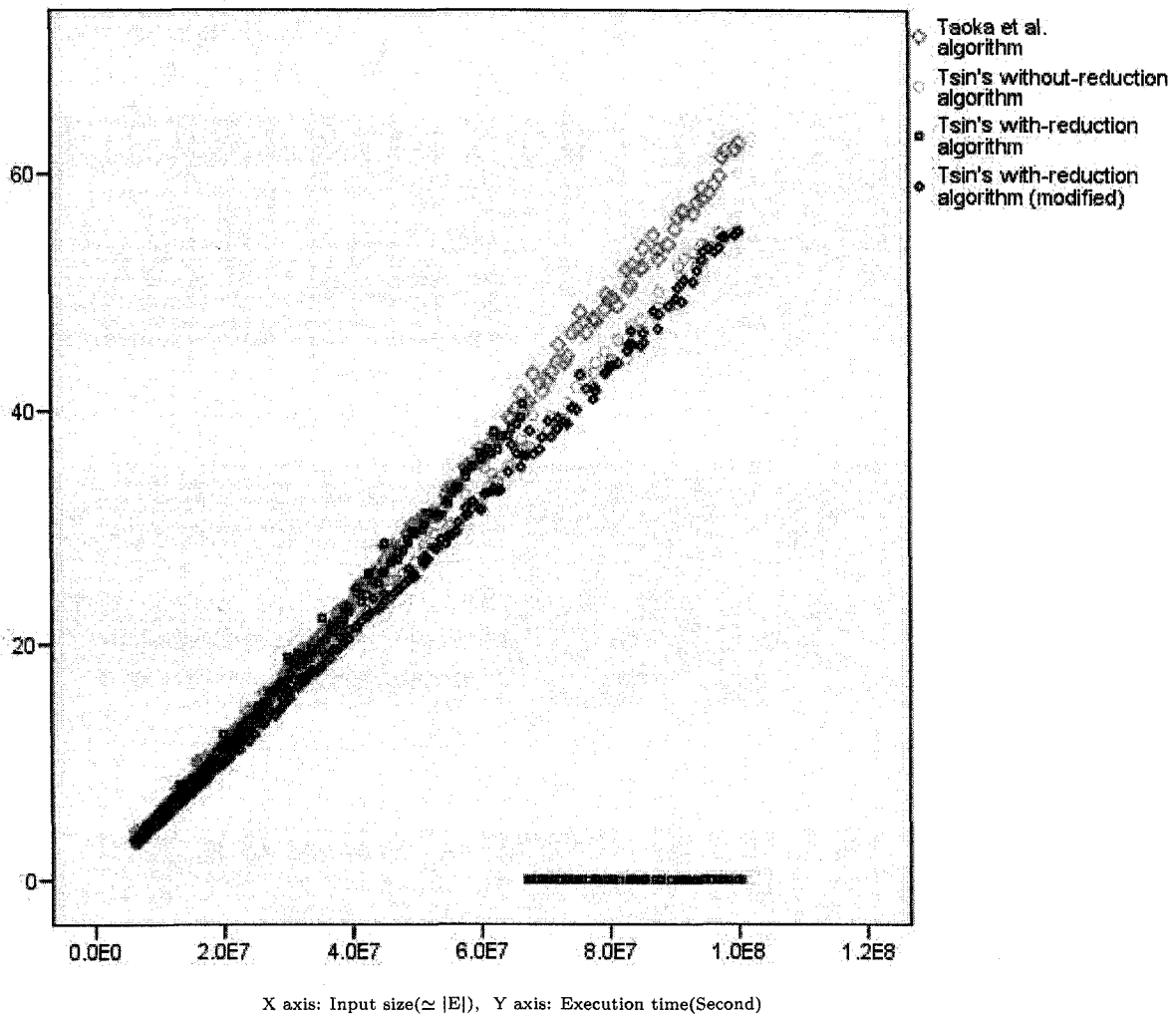


Figure 5.8: *Determining* $3-$ *edge connectivity*; *only Yes instances*; $|E| \simeq \frac{|V|^2}{4}$

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

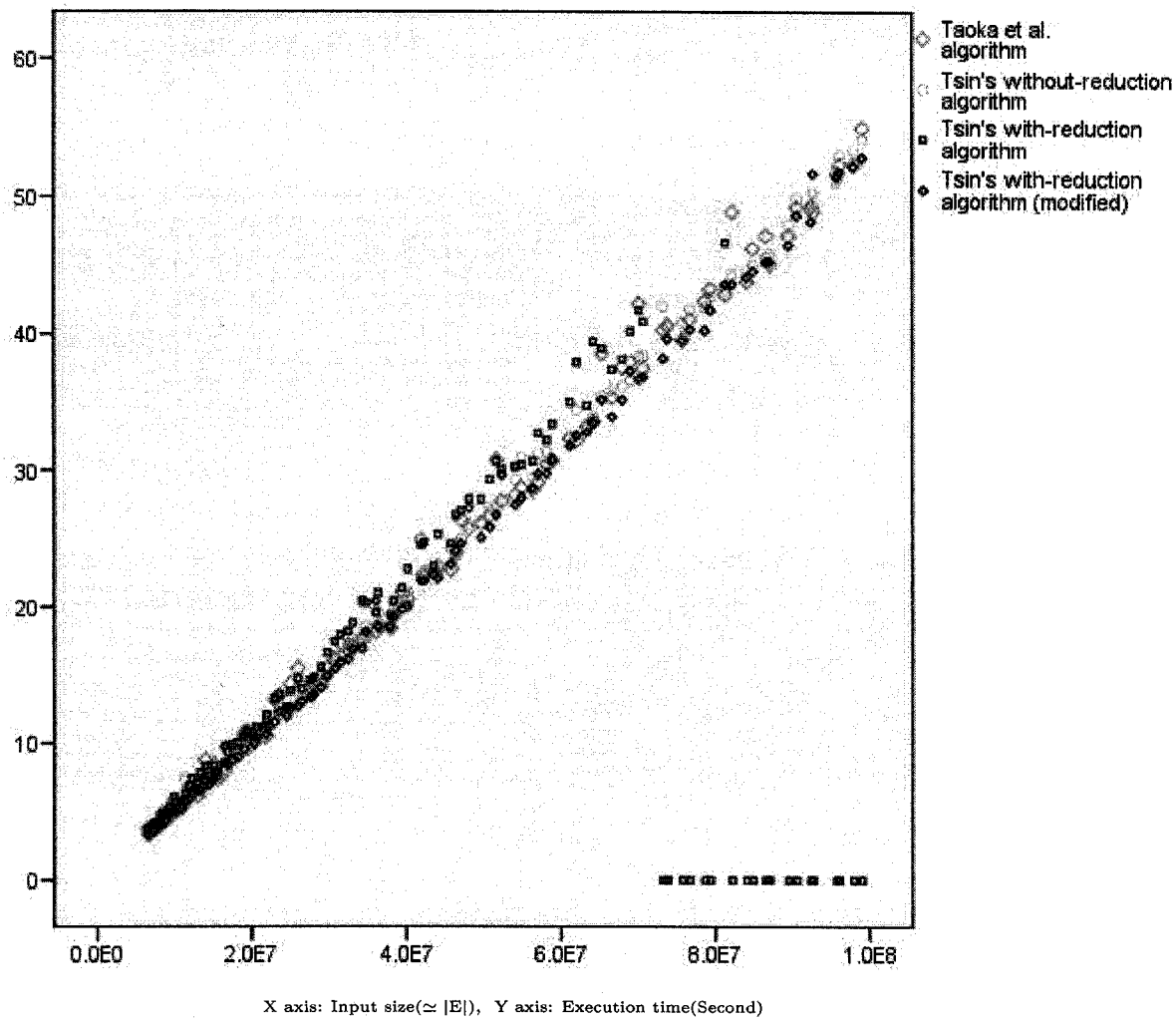


Figure 5.9: *Determining* $3-$ *edge connectivity*; *only No instances*; $|E| \simeq \frac{|V|^2}{4}$

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

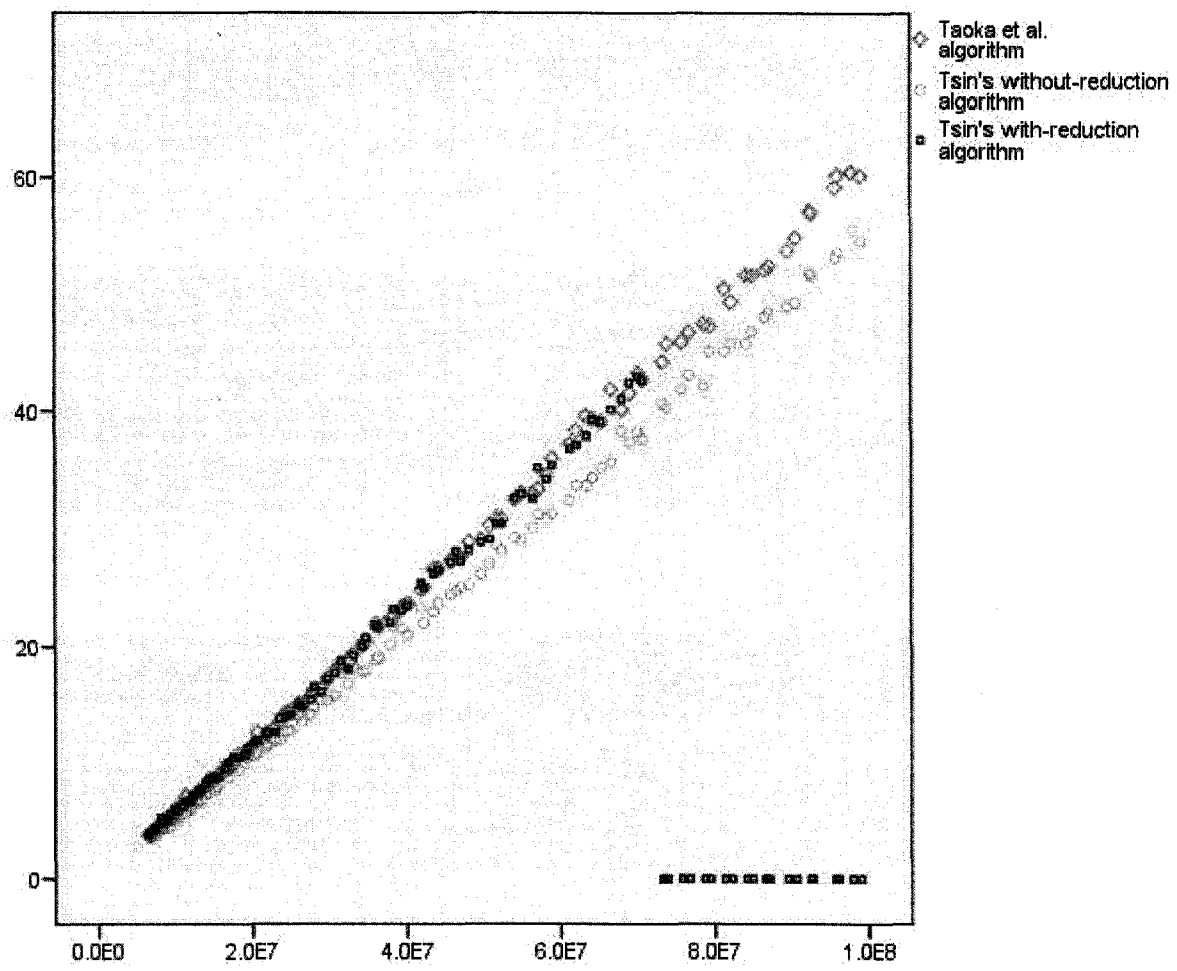


Figure 5.10: *Determining cut − pairs on No instances*; $|E| \simeq \frac{|V|^2}{4}$

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

Chapter 6

Conclusion and Future work

One of the important reasons that we were able to perform experiments on such large input sizes is because of the data structure model we proposed in Chapter 3 to deal with cut-pair deletions. Without this model, the largest input size we were able to handle would have been about half of the largest input size we are able to handle.

The modified version of WR has the best performance in computing 3-edge connected components and determining *Yes* instances. This is because the algorithm does not need to construct and search back-edge lists for degree determination of vertices. As a result, it is now able to determine if the degree of each vertex is 2, which is a key step in the algorithm, in O($|V|$) (rather than $O(|E|)$) time and space as explained in Chapter 4.3. This algorithm could be useful in resolving the irreducibility problem for the Feynman diagrams.

The WOR algorithm has the best performance in finding cut-pairs. This is because it does not have to search the adjacency lists again as it does not have to find the connected components while the WR algorithm has to construct and search in the back-edge lists. The WOR algorithm can be used in Bioinformatics where finding the cut-pairs of a graph is an important process. One disadvantage of the modified version of WR is that it cannot find

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

cut-pairs because it does not construct back-edge lists.

Our future work is to apply the algorithms to even much larger input sizes. One direction is to adapt them to the external memory model [18]. On such a model, the size of the input graph is larger than the size of the main memory of the computer so that a substantial portion of the input graph must be stored on the external memory.

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

Bibliography

[1] Chen Z., "Testing a Graph for 3-edge Connectivity", Master Thesis, The University of Georgia, Athens, Georgia, 2001.

[2] S. Coleman, "Aspects of Symmetry: Selected Erice Lectures", Cambridge University Press, ISBN 0521318270, 1988.

[3] Corcoran J.N., Schneider U. and Schttler H.-B., "Perfect Stochastic Summation in High Order Feynman Graph Expansions", Int. J. Mod. Phys. C, 2006, Vol. 17 (11), 1527-1549.

[4] Dehne F., Langston M. A., Luo X., Pitre S., Shaw P. and Zhang Y., "The Cluster Editing Problem: Implementations and Experiments", IWPEC 2006, Zuerich, September 2006.

[5] Galil Z. and Italiano G.F., "Reducing Edge Connectivity to Vertex Connectivity", SIGACT News 22, 57-61, 1991.

[6] Hopcroft J. and Tarjan R.E., "Dividing a Graph into Triconnected Components", SIAM J. Comput. 2, 135-158, 1973.

[7] Michael T. Goodrich, Roberto Tamassia, "Algorithm Design: Foundations, Analysis, and Internet Examples", ISBN 0471383651, Wiley, Paperback, c2002.

[8] Michael T. Goodrich, Roberto Tamassia, "Data Structures and Algorithms in Java" (3rd edition), ISBN: 0471469831, Hoboken, NJ : Wiley, c2004.

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

[9] Nagamochi H. and Ibaraki T., "A Linear-time Algorithm for Computing 3-edge-connected Components in a Multigraph", Japan J. Indust. Appl. Math. 9, 163-180, 1992.

[10] Shaw P., Department of Computer Science, University of Newcastle, Australia, Electronic Communication, 2006.

[11] Steven S. Skiena, "The Algorithm Design Manual", Springer, ISBN 0387948600, (July 31, 1998)

[12] Sun F., "Checking a Graph for 3-edge-connectivity", Master Thesis, The University of Georgia, Athens, Georgia, 2003.

[13] Taoka S., Watanabe T. and Onaga K., "A Linear-time Algorithm for Computing All 3-edge-connected Components of a Multi Graph", IEICE Trans. Fundamentals E75(3), 410-424, 1992.

[14] Trajan R., "Depth-first Search and Linear Graph Algorithms", SIAM J. COMPUT., Vol. 1, No. 2, June 1972.

[15] Tsin, Y.H., "A Simple 3-edge-connected Component Algorithm", Theory of Computing Systems, Vol 40 Number 2, pp 125-142, 2005.

[16] Tsin, Y.H., "An Efficient Distributed Algorithm for 3-edge-connectivity", International Journal of Foundations of Computer Science, Vol 17 Number 3, pp 677-701, 2006.

[17] Tsin, Y.H., "Lecture Notes for Graph Algorithms(60-592)", School of Computer Science, University of Windsor, 2004.

[18] Vitter, J., "External memory algorithms and data structures: dealing with massive data", ACM Computing Surveys, Vol 33 Number 2, pp 209-271, 2001.

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

Vita Auctoris

Nima Norouzi was born in 1977 in Tehran, Iran. He obtained his Bachelor of Arts degree in the field of software engineering in 2001 from Azad University, sought Tehran branch. In 2004, he went to the university of Windsor, Ontario, Canada, and graduated with a Master of Science degree in computer science in 2007.

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