Finding twinless strong bridges and articulation points in linear time

A Thesis

submitted to the designated

by the General Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Evangelos Kosinas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION IN DATA SCIENCE AND ENGINEERING

University of Ioannina

July 2020

Examining Committee:

- **Loukas Georgiadis**, Assoc. Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Leonidas Palios**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Papadopoulos Charis**, Assoc. Professor, Department of Mathematics, University of Ioannina

Acknowledgements

Ευχαριστώ τον κύριο Λουκά Γεωργιάδη που δέχτηκε να είναι ο επιβλέπων για την εκπόνηση της μεταπτυχιακής μου διατριβής, που μου πρότεινε τα προβλήματα με τα οποία ασχολήθηκα και που με παρέπεμψε σε ορισμένες πολύ καλές εργασίες στην πορεία της έρευνάς μου. Ευχαριστώ επίσης τα μέλη της εξεταστικής επιτροπής, τους κυρίους Παληό και Παπαδόπουλο, για τα χρήσιμα σχόλια και τις παρατηρήσεις τους.

Table of Contents

List of Figures **iii**

List of Algorithms **iv**

Abstract **v**

Εκτεταμένη Περίληψη **vi**

1 Introduction **1**

1.1 Basic concepts of graph theory 1

1.1.1 Graphs 1

1.1.2 Connectivity in graphs 2

1.2 Twinless strong connectivity 3

1.3 Our contribution 5

1.4 Overview of this work 6

2 Computing twinless strong bridges and articulation points **7**

2.1 An equivalent characterization of twinless strong connectivity 7

2.2 Finding twinless strong bridges, and counting TSCCs 8

2.3 Finding twinless strong articulation points, and counting TSCCs 10

3 Information extracted from a DFS tree **14**

3.1 Depth-first search 14

3.2 Computing all $high(v)$ and $high_p(v)$ in linear time 15

3.3 Computing all $M(v)$ and $M_p(v)$ in linear time 19

4 Counting cut-pairs in linear time **23**

4.1 Introduction 23

4.2 The case back-edge - tree-edge 24

4.3 The case where both edges are tree-edges 26

5 Counting vertex-edge cut-pairs in linear time 31

5.1 Introduction 31

5.2 The case where e is a back-edge 32

5.3 The case where e lies on the simple tree path $T[v, r]$ 33

5.3.1 The case $M(u) = v$ 33

5.3.2 The case where $M(u)$ is a proper descendant of v 35

5.4 The case where e lies in $T(v)$ 38

5.4.1 The case $high(u) = v$ 38

5.4.2 The case $high(u) < v$ 40

6 Open problems 47

Bibliography 49

List of Figures

1.1 An illustration of some concepts related to graph connectivity. 4
1.2 Example of a twinless strongly connected digraph. 5

2.1 Types of twinless strong bridges. 8
2.2 Types of twinless strong articulation points. 11
2.3 The problem of finding vertex-edge cut-pairs is related to 3-connectivity. 13

3.1 Concepts defined on the DFS tree. 16

4.1 Example of cut-pairs of tree-edges. 25
4.2 The necessary and sufficient conditions for two tree-edges to form a cut-pair. 27

5.1 An illustration of Proposition 5.2 (for the case $M(u) = v$). 35
5.2 An illustration of Proposition 5.3 (for the case that $M(u)$ is a proper descendant of v). 37
5.3 An illustration of Proposition 5.4 (for the case that $high(u) = v$). 39
5.4 An illustration of Proposition 5.5 (for the case that $high(u) < v$). 41

List of Algorithms

3.1 SimpleHigh 17

3.2 FastHigh 18

3.3 Procedure FindM 21

4.1 Counting cut-pairs of tree-edges 30

5.1 Calculating $\mathit{up}(c)$ and $\mathit{down}(v, c)$ 33

5.4 The case $\mathit{high}(u) = v$ 40

5.2 The case $M(u) = v$ 44

5.3 The case $M(u) > v$ 45

5.5 The case $\mathit{high}(u) < v$ 46

Abstract

Evangelos Kosinas, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, July 2020.
Finding twinless strong bridges and articulation points in linear time.
Advisor: Loukas Georgiadis, Associate Professor.

A directed graph $G = (V, E)$ is called twinless strongly connected if it contains a strongly connected subgraph $H = (V, E')$ without any pair of antiparallel (twin) edges. The following problem naturally suggests itself: given a twinless strongly connected graph G, find all edges e (resp. all vertices v), such that $G \backslash e$ (resp. $G \backslash v$) is not twinless strongly connected. Every such edge (resp. vertex) is called twinless strong bridge (resp. articulation point). In this thesis, we show how to compute these elements in linear time. For this purpose, we have developed new linear-time algorithms that solve problems related to 3-connectivity in undirected graphs. Although there already exist algorithms that can solve these problems in asymptotically optimal time, they are somewhat involved. Our approach is conceptually simple and thus leads to algorithms that are both easy to implement and likely to perform better in practice.

Εκτεταμενη Περιληψη

Ευάγγελος Κοσίνας, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2020.
Αλγόριθμοι γραμμικού χρόνου για τον υπολογισμό των twinless ισχυρών γεφυρών και των κόμβων twinless ισχυρής άρθρωσης.
Επιβλέπων: Λουκάς Γεωργιάδης, Αναπληρωτής Καθηγητής.

Σε αυτήν την εργασία ασχολούμαστε με μία έννοια συνεκτικότητας σε κατευθυνόμενα γραφήματα που ονομάζεται twinless ισχυρή συνεκτικότητα (twinless = *χωρίς αντιπαράλληλες ακμές*) και αποτελεί μια ισχυρότερη συνθήκη από την ισχυρή συνεκτικότητα. Συγκεκριμένα, λέμε ότι ένα κατευθυνόμενο γράφημα $G = (V, E)$ είναι twinless ισχυρά συνεκτικό αν υπάρχει (τουλάχιστον) ένα υπογράφημά του $H = (V, E')$ το οποίο είναι ισχυρά συνεκτικό και δεν περιέχει αντιπαράλληλες ακμές (twins). Αυτό είναι ένα είδος συνεκτικότητας που ορίστηκε το 2006, και έχει εφαρμογές σε προβλήματα τηλεπικοινωνιακών δικτύων και αρχιτεκτονικής.

Ένα ερώτημα που τίθεται εύλογα όταν ορίζει κανείς μια έννοια συνεκτικότητας σε γραφήματα, είναι το πώς μπορούν να εντοπιστούν, με αποδοτικό τρόπο, όλα τα στοιχεία του (κόμβοι ή ακμές), τα οποία, με την αφαίρεσή τους, καταστρέφουν αυτήν την ιδιότητα στο γράφημα. Έτσι μπορεί κανείς να έχει μια εικόνα π.χ. για το πόσο ανθεκτικό είναι ένα δίκτυο στην απώλεια κόμβων ή ακμών. Εμείς εξετάζουμε ακριβώς το πρόβλημα του εντοπισμού των ακμών (αντ. των κόμβων) σ' ένα twinless ισχυρά συνεκτικό γράφημα, οι οποίες (αντ. οι οποίοι) με την αφαίρεσή τους καταστρέφουν την twinless ισχυρή συνεκτικότητα σε αυτό. Οι πρώτες λέγονται twinless ισχυρές γέφυρες (twinless strong bridges), ενώ οι δεύτεροι κόμβοι twinless ισχυρής άρθρωσης (twinless strong articulation points). Αυτό το πρόβλημα εξετάστηκε από έναν ερευνητή το 2019, οποίος έδωσε αλγορίθμους πολυπλοκότητας $O(nm)$ για τον υπολογισμό αυτών των στοιχείων, όπου n είναι το πλήθος των κόμβων και

m το πλήθος των ακμών στο γράφημα εισόδου. Εμείς βελτιώνουμε αυτό το αποτέλεσμα, δίνοντας αλγορίθμους ασυμπτωτικά βέλτιστης (δηλαδή, στην προκειμένη περίπτωση, γραμμικής) πολυπλοκότητας για το ίδιο πρόβλημα.

Η εργασία μας στηρίζεται σ' έναν ισοδύναμο χαρακτηρισμό της twinless ισχυρής συνεκτικότητας που δόθηκε το 2006 απ' τον ερευνητή που εισήγαγε αυτήν την έννοια. Αυτός απέδειξε ότι ένα κατευθυνόμενο γράφημα είναι twinless ισχυρά συνεκτικό αν και μόνο αν είναι ισχυρά συνεκτικό και το μη-κατευθυνόμενο γράφημα που αντιστοιχεί σε αυτό είναι 2-συνεκτικό ως προς τις ακμές (2-edge-connected). Έτσι φαίνεται ξεκάθαρα πως η έννοια της twinless ισχυρής συνεκτικότητας αποτελεί ένα μίγμα συνεκτικοτήτων, στο αρχικό κατευθυνόμενο γράφημα και το αντίστοιχο μη-κατευθυνόμενο. Είναι λοιπόν εύλογο να υποθέσει κανείς ότι ορισμένα προβλήματα που αφορούν αυτήν την έννοια συνεκτικότητας μπορούν να λυθούν με την παράλληλη αντιμετώπισή τους στα δύο επίπεδα: το κατευθυνόμενο και το μη-κατευθυνόμενο.

Πράγματι, εμείς καταφέρνουμε να λύσουμε το βασικό πρόβλημα της εργασίας μας αναπτύσσοντας αλγορίθμους που επιλύουν ορισμένα προβλήματα που αφορούν την 3-συνεκτικότητα σε μη-κατευθυνόμενα γραφήματα (εφόσον τα προβλήματα που αφορούν το επίπεδο της ισχυρής συνεκτικότητας υπάρχουν ήδη γνωστοί αλγόριθμοι που τα επιλύουν). Συγκεκριμένα, λύνουμε σε γραμμικό χρόνο τα εξής προβλήματα:

- *Δοθέντος ενός μη-κατευθυνόμενου γραφήματος G που είναι 2-συνεκτικό ως προς τις ακμές, να βρεθούν όλα τα $count(e)$, για όλες τις ακμές e του γραφήματος, όπου $count(e)$ είναι το πλήθος των ακμών με τις οποίες η e σχηματίζει ζευγάρι τομής*, και

- *Δοθέντος ενός μη-κατευθυνόμενου γραφήματος G που είναι 2-συνεκτικό ως προς τους κόμβους, να βρεθούν όλα τα $count(v)$, για όλους τους κόμβους v του γραφήματος, όπου $count(v)$ είναι το πλήθος των ακμών με τις οποίες ο v σχηματίζει ζευγάρι τομής κόμβου-ακμής*,

τα οποία έχουν εφαρμογές και στο πρόβλημα του υπολογισμού των twinless ισχυρά συνεκτικών συνιστωστών (TSCCs) που σχηματίζονται μετά την αφαίρεση μιας ακμής (αντ. ενός κόμβου) που αποτελεί twinless ισχυρή γέφυρα αλλά όχι ισχυρή γέφυρα (αντ. κόμβο twinless ισχυρής άρθρωσης αλλά όχι ισχυρής άρθρωσης). Αν και τα προβλήματα που επιλύουν οι αλγόριθμοί μας μπορούν να λυθούν με γνωστές μεθόδους σε ασυμπτωτικά βέλτιστο χρόνο, οι αλγόριθμοι που προτείνουμε εμείς είναι απλούστεροι στην σύλληψή τους και ευκολότεροι στην υλοποίησή τους, και ενδέχεται λοιπόν να είναι και ταχύτεροι στην πράξη. Επιπλέον, οι έννοιες στις οποίες βασίζονται οι αλγόριθμοί μας, καθώς και οι τεχνικές που αξιοποιούν, είναι ουσιαστικά οι ίδιες και στις δύο περιπτώσεις, και φαίνεται πως μας δίνουν ένα ενιαίο πλαίσιο για την αντιμετώπιση προβλημάτων συνεκτικότητας σε μη-κατευθυνόμενα γραφήματα.

Chapter 1

Introduction

1.1 Basic concepts of graph theory
1.2 Twinless strong connectivity
1.3 Our contribution
1.4 Overview of this work

1.1 Basic concepts of graph theory

Although our exposition is self-sufficient, we refer the reader to Diestel [1] for a good introduction to the basic concepts of graph theory that are needed for our purposes (in particular, chapters 1 and 3).

1.1.1 Graphs

A graph is a collection of vertices and edges connecting those vertices. Formally, it is defined as a pair of sets (V, E), where V is the set of vertices and E consists of edge of the form $\{u, v\}$ or (u, v), depending on whether the graph is undirected or directed, respectively. A directed graph is also called a digraph. The word "graph", if unspecified, might mean either an undirected or a directed graph. If $\{u, v\}$ (resp. (u, v)) is an edge in an undirected (resp. in a directed) graph, then u, v are called the ends of the edge. A pair of edges (x, y), (y, x) in a directed graph is called a pair of antiparallel edges. If $G = (V, E)$ is a digraph, the underlying undirected graph of G, denoted by G^u, is the graph $(V, \{\{x, y\} \mid (x, y) \in E\})$.

Let $G = (V, E)$ be a graph. A graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$ is called a subgraph of G. If $C \subseteq V$ is a set of vertices, then the induced subgraph of C, denoted by $G[C]$, is the maximal subgraph (C, E') of G, w.r.t. E'. In other words, $G[C] := (C, \{e \in E \mid \text{both ends of } E \text{ are in } C\})$. Let x be either a vertex or an edge of G. $G \setminus \{x\}$ denotes the graph that remains after the removal of x from G. Formally, if v is a vertex of G, $G \setminus \{v\} := (V \setminus \{v\}, \{e \in E \mid \text{both ends of e are in } V \setminus \{v\}\})$; and if e is an edge of G, $G \setminus \{e\} := (V, E \setminus \{e\})$. If $X = \{x_1, \ldots, x_k\}$ is a collection of vertices and edges, we define $G \setminus X$ recursively as $G \setminus X := (G \setminus \{x_1\}) \setminus \{x_2, \ldots, x_k\}$. In the case that $X = \{x\}$ (i.e. X is a singleton), we may also write $G \setminus X$ as $G \setminus x$.

1.1.2 Connectivity in graphs

The concept of connectivity is one of the most fundamental in the theory of graphs. It can be most easily grasped by introducing it through the notion of paths. So let G be a graph. A path P in G is an alternating sequence of vertices and edges $v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k$, where every v_i (for $i = 1, \ldots, k$) is a vertex and every e_i (for $i = 1, \ldots, k-1$) has the form $\{v_i, v_{i+1}\}$ if G is undirected, or (v_i, v_{i+1}) if G is directed. We say that P starts from v_1 and ends in v_k. If $k = 2$, we do not distinguish the path P from the edge e_1. P is called simple if all vertices $v_1, \ldots, v_k$ are distinct. A graph (resp. a digraph) G is called connected (resp. strongly connected) if for every pair of vertices $\{u, v\}$ there is a path starting from u and ending in v. Equivalently, a (di)graph G is (strongly) connected if for every pair of vertices $\{u, v\}$ there is a *simple* path starting from u and ending in v.

One is interested to know how resilient is the connectivity of a graph to the removal of vertices or edges. Thus we get some stronger conditions than those of (strong) connectivity. More specifically, an undirected (resp. a directed) graph G is called k-edge-connected, if, for every set X of $k-1$ edges, $G \setminus X$ is connected (resp. strongly connected). In other words, in order to disconnect G, at least k edges have to be removed. The notion of k-vertex-connectivity is defined in an analogous manner: an undirected (resp. a directed) graph G is called k-vertex-connected, if it contains at least $k+1$ vertices and, for every set X of $k-1$ vertices, $G \setminus X$ is connected (resp. strongly connected). That is, in order to disconnect G, at least k vertices have to be removed. A graph that is k-vertex-connected is also k-edge-connected, but the reverse is not necessarily true - unless $k = 1$, in which case the notions of k-vertex and k-edge connectivity coincide with that of (strong) connectivity. If e (resp. v) is an edge (resp. a vertex) in a (strongly) connected (di)graph G and $G\setminus e$ (resp. $G\setminus v$) is not (strongly) connected, then e is called a (strong) bridge (resp. a (strong) articulation point). More generally, if G is k-edge-connected (resp. k-vertex-connected), a set X of k edges (resp. vertices) whose removal disconnects G is called a cut (resp. a separation set). A connected undirected graph in which every edge is a bridge is called a tree. A set C of vertices from a (di)graph G maximal w.r.t. the property that every two vertices in C remain in the same (strongly) connected component of G after the removal of any set of $k-1$ edges, is called a k-edge-connected component of G. Similarly, a set C of vertices from a (di)graph G maximal w.r.t. the property that every u, v in C remain in the same (strongly) connected component of G after the removal of any set of $k-1$ vertices, none of which is u or v, is called a k-vertex-connected component of G. Figure 1.1 is an illustration of some of these concepts and contains a few observations which are important for our purposes.

1.2 Twinless strong connectivity

In 2006, S. Raghavan [11] introduced the notion of twinless strong connectivity in directed graphs. A digraph $G = (V, E)$ is called twinless strongly connected if there exists a subgraph (V, E') of G which is strongly connected and contains no pair of antiparallel (or twin) edges. It is obvious that a graph is twinless strongly connected only if it is strongly connected (but the reverse is not necessarily true). (See Figure 1.2.) Twinless strong connectivity is thus a stronger condition than that of strong connectivity.

The notions of strongly connected components and strong bridges and articulation points have their analogues in the context of twinless strong connectivity. To be more specific, let G be a digraph. A maximal (w.r.t. their vertices) subgraph of G with the property of being twinless strongly connected is called a twinless strongly connected component (TSCC) of G. Furthermore, if G is twinless strongly connected, an edge e (resp. a vertex v) such that $G \setminus \{e\}$ (resp. $G \setminus \{v\}$) is not twinless strongly connected, is called a twinless strong bridge (resp. a twinless strong articulation point).

In order to determine how resilient a twinless strongly connected digraph is to the loss of vertices or edges, one is naturally interested in finding all twinless strong

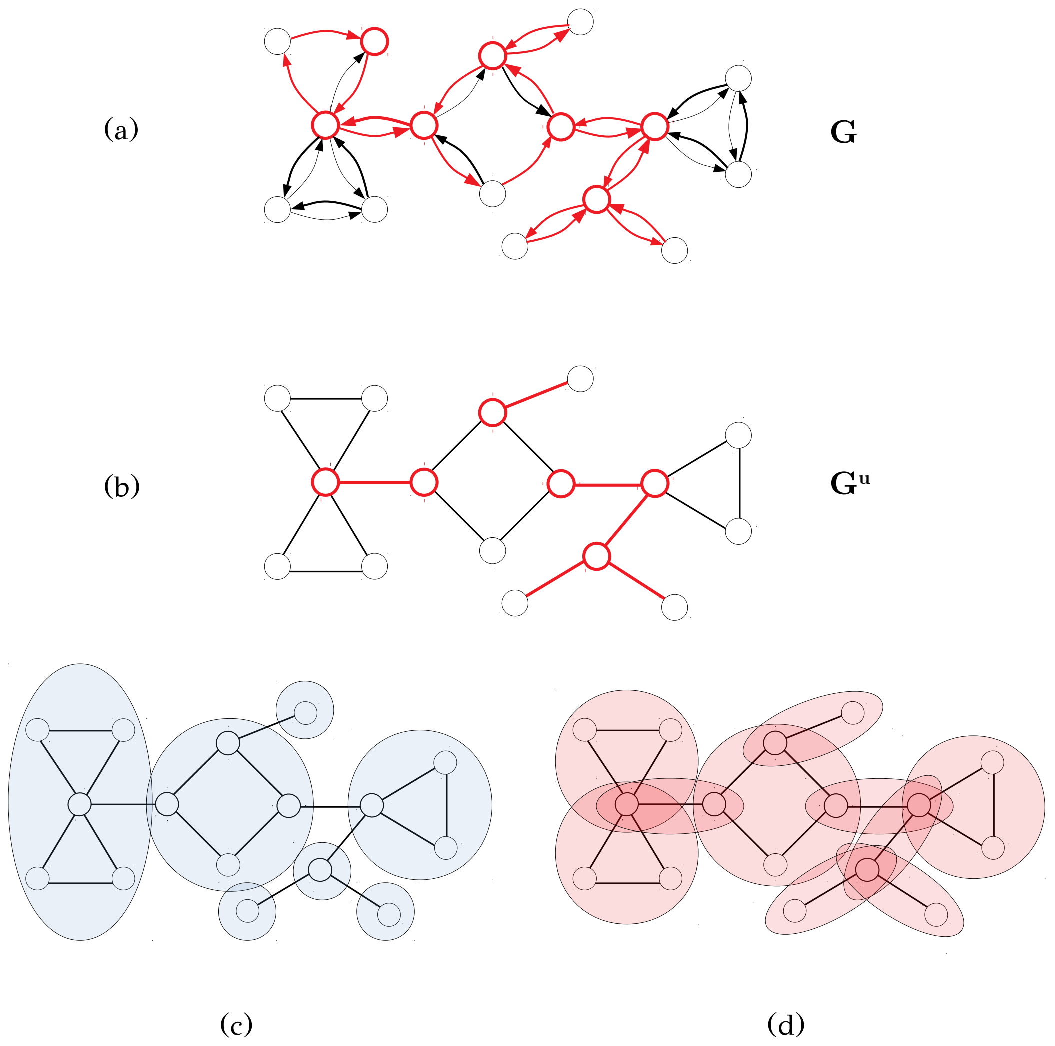
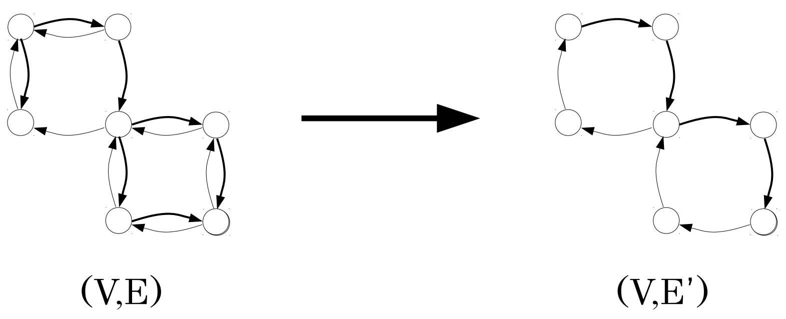
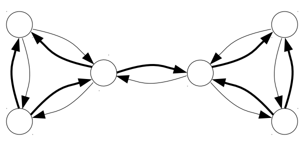


Figure 1.1: (a) A strongly connected digraph G with strong bridges and articulation points shown in red. (b) The underlying undirected graph G^u of G with bridges and articulation points shown in red. Every bridge (resp. every articulation point) of G^u corresponds to a strong bridge (resp. strong articulation point) of G, but the reverse is not true. (c) The 2-edge-connected components of G^u. They partition the vertices of the graph, and their induced subgraphs are connected with the bridges of G^u in a tree structure. (d) The 2-vertex-connected components of G^u. A vertex of G^u is an articulation point if and only if it belongs to at least two 2-vertex-connected components. The induced subgraphs of the 2-vertex-connected components partition the edges of the graph.

bridges and articulation points. In 2019 Jaberi [10] gave algorithms of $O(nm)$ complexity for the computation of all twinless strong bridges and articulation points in



(a)



(b)

Figure 1.2: (a) An example of a twinless strongly connected digraph. (b) is strongly connected but not twinless strongly connected.

a twinless strongly connected digraph with n vertices and m edges, and left as an open question whether these elements can be computed in linear time. This is a reasonable hypothesis, since there exist linear-time algorithms that solve the analogous problems for connectivity and strong connectivity (i.e. the computation of bridges and articulation points in undirected graphs and the computation of strong bridges and articulation points in digraphs) [12], [7].

1.3 Our contribution

In this thesis we provide an algorithm which computes all twinless strong bridges (resp. all twinless strong articulation points) of a twinless strongly connected digraph G in linear time. Furthermore, we show how, after a linear-time preprocessing of the graph, we can answer queries of the form "*Given a twinless strong bridge e (resp. a twinless strong articulation point v) which is not a strong bridge (resp. a strong articulation* *point), report the number of TSCCs of $G \setminus \{e\}$ (resp. $G \setminus \{v\}$)"*, in time $O(1)$.

To do this, we have developed linear-time algorithms which solve the following problems in undirected graphs:

- *Given a 2-edge-connected undirected graph $G = (V, E)$, find all count(e), for every $e \in E$, where count$(e) := \#\{e' \in E \mid \{e, e'\}$ is a cut-pair$\}$.*
- *Given a 2-vertex-connected undirected graph $G = (V, E)$, find all count(v), for every $v \in V$, where count$(v) := \#\{e \in E \mid \{v, e\}$ is a vertex-edge cut-pair$\}$.*

There are linear-time algorithms that can solve these problems ([13], [5]), but they are somewhat involved (especially [5], which constructs SPQR-trees, and relies on the algorithm of Hopcroft and Tarjan for 3-connectivity [6]). Our approach is conceptually simple and leads to algoritmhs that are easy to implement.

1.4 Overview of this work

In Chapter 2 we describe linear-time algorithms for the computation of all twinless strong bridges and articulation points in a twinless strongly connected digraph and the number of TSCCs in the graph that remains after the removal of a twinless strong bridge which is not a strong bridge (resp. a twinless strong articulation point which is not a strong articulation point). We show how, in order to solve these problems, we can rely on algorithms which solve some problems related to 3-connectivity in undirected graphs. These algorithms are described in Chapters 4 and 5. In Chapter 3 we define and show how to compute the parameters that are essential to our algorithms, and are defined on the structure provided by a depth-first scan of an undirected graph. Finally, in Chapter 6 we discuss some open problems.

Chapter 2

Computing twinless strong bridges and articulation points

2.1 An equivalent characterization of twinless strong connectivity
2.2 Finding twinless strong bridges, and counting TSCCs
2.3 Finding twinless strong articulation points, and counting TSCCs

2.1 An equivalent characterization of twinless strong connectivity

The starting point of our work is the following equivalent characterization of twinless strong connectivity given by S. Raghavan:

Theorem 2.1. *([11]) A digraph G is twinless strongly connected if and only if it is strongly connected and its underlying undirected graph G^u is 2-edge-connected.*

This shows clearly how this notion of connectivity in digraphs is related to a type of connectivity in undirected graphs. As a corollary, we have the following characterization of the TSCCs of a strongly connected digraph:

Corollary 2.1. *([11]) The TSCCs of a strongly connected digraph G correspond to the 2-edge-connected components of its underlying undirected graph G^u.*

2.2 Finding twinless strong bridges, and counting TSCCs

An immediate consequence of Theorem 2.1 is that a twinless strong bridge in a twinless strongly connected digraph is either (1) a strong bridge or (2) an edge whose removal destroys the 2-edge connectivity in the underlying graph. A twinless strong bridge can be an edge of both type (1) and (2) (see Figure 2.1). Now, since all strong bridges can be found in linear time [7], in order to find all twinless strong bridges it is sufficient to focus our attention on computing the edges of type (2). To do this, we only have to find all the edges of the underlying graph whose removal destroys the 2-edge-connectivity; in other words: all the edges of the underlying graph which belong to a cut-pair.

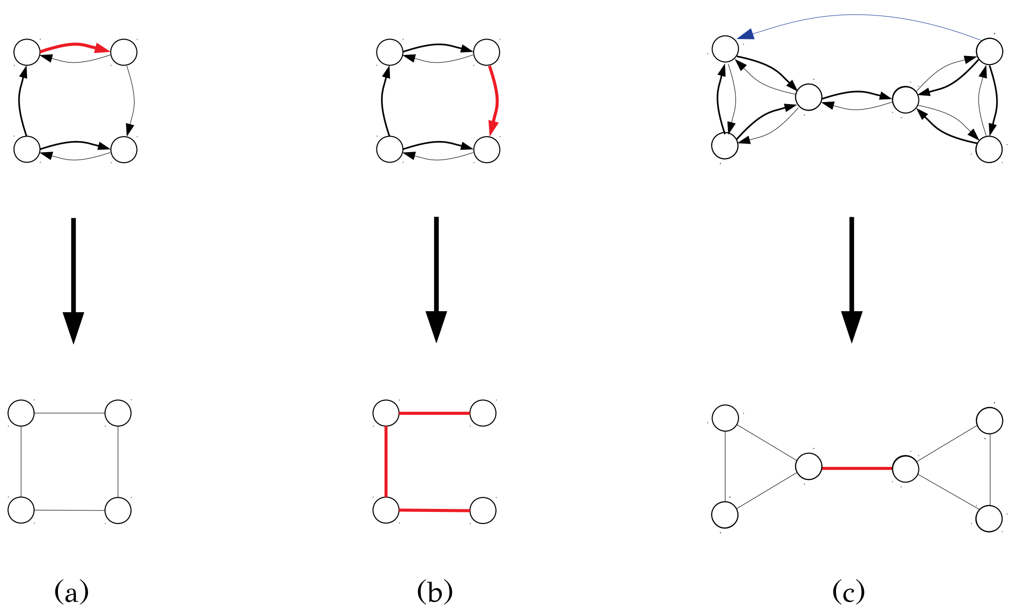


Figure 2.1: (a) A twinless strong bridge of type (1) but not (2). (b) A twinless strong bridge of type (1) and (2). (c) A twinless strong bridge of type (2) but not (1). Below each graph is depicted the underlying undirected graph that corresponds to the above after the removal of the twinless strong bridge (which is the coloured edge in the directed graph). With red are shown the bridges.

Lemma 2.1. *An edge $e = (x, y)$ in a twinless strongly connected digraph G is a twinless* *strong bridge but not a strong bridge if and only if its twin* (y, x) *is not an edge of* G*, and* $(G \setminus e)^u$ *is not* 2*-edge-connected.*

Proof. Let $e = (x, y)$ be an edge in a twinless strongly connected digraph G which is a twinless strong bridge but not a strong bridge. Theorem 2.1 implies, that the removal of this edge leaves us with a graph $H = G \setminus e$ whose underlying undirected graph H^u is not 2-edge-connected. Now, since the initial graph is twinless strongly connected, its underlying undirected graph G^u is 2-edge-connected, therefore the twin (y, x) of e is not an edge of G (otherwise, the removal of e would leave the underlying graph unchanged). The converse is an immediate consequence of Theorem 2.1. □

This suggests the following algorithm for the computation of all twinless strong bridges. Firstly, we mark all edges that are strong bridges. Then we find all edges of the underlying graph which belong to a cut-pair, and mark those that correspond to an edge in the initial graph whose twin is missing from it. All the marked edges are precisely the twinless strong bridges.

Now, although there are known algorithms which compute, in linear time, all the edges which belong to a cut-pair in a 2-edge-connected undirected graph G (see e.g. Tsin [13]), in Chapter 4 we describe our own algorithm, which is conceptually simple and easy to implement in practice. It relies on some parameters extracted by a depth-first search (DFS) on the graph, which we'll describe in the next Chapter. Furthermore, our algorithm counts, for every edge e, the number of edges e' such that $\{e, e'\}$ is a cut-pair; we call this number *count*(e). This is useful for counting the TSCCs after the removal of a twinless strong bridge which is not a strong bridge in a twinless strongly connected digraph, as suggested by the following Lemma:

Lemma 2.2. *Let* G *be a twinless strongly connected digraph, and let* e *be a twinless strong bridge of* G *which is not a strong bridge. Then* *count*$(\tilde{e}) + 1$*, where* $\tilde{e}$ *is the edge in* G^u *corresponding to* e*, is the number of TSCCs of* $G \setminus e$*.*

Proof. Since e is not a strong bridge, $G \setminus e$ is strongly connected. By Corollary 2.1, the TSCCs of a strongly connected digraph correspond to the 2-edge-connected components of its underlying undirected graph. Now, the number of the 2-edge-connected components of $(G \setminus e)^u = G^u \setminus \tilde{e}$ equals the number of its bridges + 1. (This is due to the tree structure of the 2-edge-connected components of a connected graph; see [1] and Figure 1.1.) By definition, this number of those bridges is equal to *count*$(\tilde{e})$. □

Thus, if e is twinless strong bridge which is not a strong bridge in twinless strongly connected digraph G, then the number of TSCCs of $G \setminus e$ equals *count*$(\tilde{e}) + 1$, where $\tilde{e}$ is the edge of G^u that corresponds to e.

2.3 Finding twinless strong articulation points, and counting TSCCs

It is an immediate consequence of Theorem 2.1 that a twinless strong articulation point in a twinless strongly connected digraph G is either (1) a strong articulation point or (2) a vertex whose removal destroys the 2-edge connectivity in the underlying undirected graph G^u. A twinless strong articulation point can be a vertex of both type (1) and (2) (see Figure 2.2). Now, since all strong articulation points can be computed in linear time [7], it remains to find all vertices of type (2). Note that such a vertex x either I entirely destroys the connectivity of the underlying graph G^u with its removal, or II, upon removal, it leaves us with a graph $G^u \setminus x$ that is connected but not 2-edge-connected (see Figure 2.2). Clearly, the set of vertices with property I are a subset of the set of strong articulation points. Therefore, it suffices to find all vertices with property II. To that end, we may process each 2-vertex-connected component of G^u separately, as the next lemma suggests.

Lemma 2.3. *Let H be a 2-edge-connected undirected graph. Let v be a vertex that is not an articulation point, and let C be its 2-vertex-connected component (2VCC). For any edge e, $H \setminus \{v, e\}$ is not connected if and only if e belongs to $H[C]$ and $H[C] \setminus \{v, e\}$ is not connected.*

Proof. $(\Rightarrow)$ Since H is 2-edge-connected, it contains no bridges. Therefore, every 2VCC of H contains at least three vertices, and thus its induced subgraph is a 2-vertex-connected subgraph of H (see Figure 1.1). Now, let e be an edge such that the graph $H \setminus \{v, e\}$ is not connected, and let C' be the 2VCC of H whose induced subgraph contains e (we recall that the 2VCCs of H partition its edges; see Figure 1.1). Suppose, for contradiction, that $C' \neq C$. Since $H[C']$ is 2-vertex-connected, is must also be 2-edge-connected, and therefore it contains no bridges. Moreover, v is not an articulation point, so it is contained in only one 2VCC of H, hence, $C' \setminus \{v\} = C'$. This means that e is not a bridge in $H[C' \setminus \{v\}] = H[C']$, and therefore not a bridge in $H \setminus v$ - a contradiction.

$(\Leftarrow)$ Recall the block graph representation of H [1]: Let T be the graph whose vertices

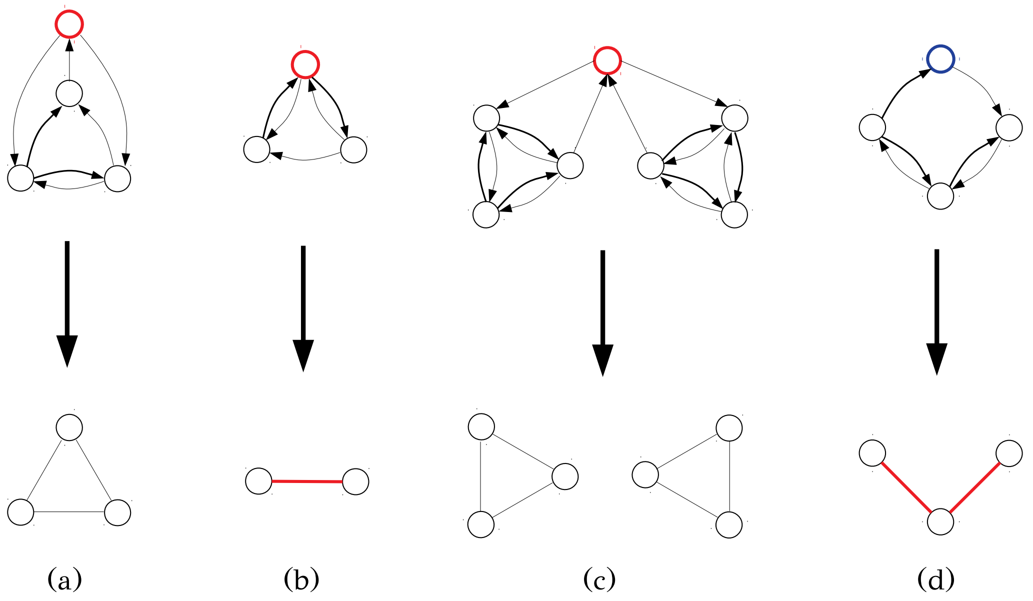


Figure 2.2: (a) A twinless strong articulation point (sap) of type (1) but not (2). (b) A twinless sap of type (1) and (2) (with property II). (c) A twinless sap of type (1) and (2) (with property I). (d) A twinless sap of type (2) but not (1) (it necessarily has property II). Below each graph is depicted the underlying undirected graph that corresponds to the above after the removal of the twinless sap (which is the coloured vertex in the directed graph). The red edges are the bridges.

are the 2VCCs and the articulation points of H, and which contains an edge e if and only if e connects a 2VCC C' with an articulation point $x \in C'$; then T is a tree. Now, suppose that there exists an edge $e = (x, y)$ in C such that $H[C] \setminus \{v, e\}$ is not connected, but $H \setminus \{v, e\}$ is connected. This means that there exists a simple path P in $H \setminus \{v, e\}$ connecting x and y. Since x and y are not connected in $H[C] \setminus \{v, e\}$, P must contain vertices from $H \setminus C$. So let z be the first vertex in P such that $z \in C$ but its successor in P is not (such a vertex exists, since $x \in C$). Since z is in C and has a neighbor that belongs to a different 2VCC, it is an articulation point. Now let w be the first vertex after z in P such that $w \in C$ (such a vertex exists, since $y \in C$). Due to the tree structure of the 2VCCs of H, we conclude that $w = z$. (In other words, when a path leaves a 2VCC through an articulation point, in order to return to this

2VCC it must pass again through the same articulation point.) But this contradicts the simplicity of P. □

So, in order to find all twinless strong articulation points, it is sufficient to solve the following problem: *Given a 2-vertex-connected undirected graph G, find all vertices v for which there exists an edge e such that $G \setminus \{v, e\}$ is not connected.* We call the sets $\{v, e\}$ with the property that $G\{v, e\}$ is not connected vertex-edge cut-pairs. In Chapter 5 we describe a linear-time algorithm for this problem. It relies on some parameters extracted by a DFS on the graph, which we'll describe in the next Chapter. Furthermore, our algorithm counts, for every vertex v, the number of edges e such that $\{v, e\}$ is a vertex-edge cut-pair; we call this number *count*(v). This is useful for counting the TSCCs after the removal of a twinless strong articulation point which is not a strong articulation point in a twinless strongly connected digraph, as suggested by the following Lemma:

Lemma 2.4. *Let G be a twinless strongly connected digraph, and let v be a twinless strong articulation point of G which is not a strong articulation point. Then count*$(v) + 1$ *(computed in the 2-vertex-connected component of v in G^u) is the number of twinless connected components of $G \setminus \{v\}$.*

Proof. Since v is not a strong articulation point, $G \setminus \{v\}$ is strongly connected. By Corollary 2.1, the twinless strongly connected components of a strongly connected graph correspond to the 2-edge-connected components of its underlying graph. Now, the number of the 2-edge-connected components of $G^u \setminus \{v\}$ equals the number of its bridges + 1 (this is due to the tree structure of the 2-edge-connected components of a connected graph). By Lemma 2.3, all these bridges lie in the 2-vertex-connected component of v in G^u. By definition, their number is *count*(v). □

We note that the problem of computing all vertices that belong to a vertex-edge cut-pair is related to 3-vertex connectivity (a fact which is illustrated in Figure 2.3), and it can be solved in linear-time by exploiting the structure of 3-vertex-connected (triconnected) components of the graph, represented by an SPQR tree. Our approach, however, avoids SPQR-trees (whose computation is somewhat involved), and is conceptually simple and easy to implement.

In summary, let us describe the algorithm for the computation of all twinless strong articulation points in a twinless strongly connected digraph G. First, we

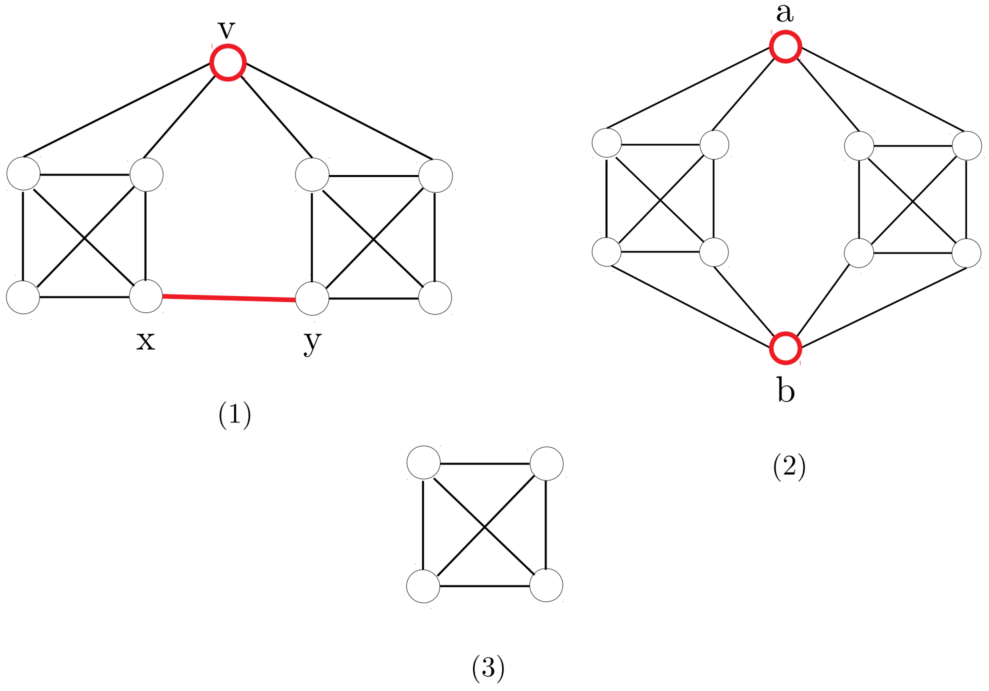


Figure 2.3: (1) A 2-vertex-connected graph with a vertex-edge cut-pair $\{v, (x, y)\}$. This implies that both $\{v, x\}$ and $\{v, y\}$ are separation pairs, and therefore the graph is not 3-vertex-connected. (2) A 2-vertex-connected graph with a separation pair $\{a, b\}$ but with no vertex-edge cut-pairs. (3) A 3-vertex-connected graph. Of course, it contains neither separation pairs nor vertex-edge cut-pairs.

mark all vertices that are strong articulation points. Then we compute the 2-vertex-connected components of the underlying graph G^u. (These can be computed in linear-time by an algorithm described in [12].) Finally, we process each 2-vertex-connected component C separately: We apply a linear-time algorithm (such as that described in Chapter 5) to find all vertices in C that belong to a vertex-edge cut-pair, and mark them. All the marked vertices are precisely the twinless strong articulation points of G.

In the next Chapter we will introduce some parameters that are needed for the algorithms described in Chapters 4 and 5, and are defined on the structure provided by a DFS on an undirected graph. We will also provide linear-time algorithms for their computation.

Chapter 3

Information extracted from a DFS tree

3.1 Depth-first search

3.2 Computing all *high*(v) and *high*$_p(v)$ in linear time

3.3 Computing all $M(v)$ and $M_p(v)$ in linear time

Depth-first search (DFS) is an extremely useful technique to explore a graph and collect data from it in the process. We refer the reader to the seminal paper of Tarjan [12] for a good exposition of DFS and of some of its applications.

3.1 Depth-first search

Let G be a connected undirected graph. We consider a DFS traversal of G, starting from an arbitrarily selected vertex r, and let T be the resulting DFS tree [12]. A vertex u is an ancestor of a vertex v (v is a descendant of u) if the simple tree path from r to v contains u. Thus, we consider a vertex to be an ancestor (and, consequently, a descendant) of itself. We let $p(v)$ denote the parent of a vertex v in T. If u is a descendant of v in T, we denote the set of vertices of the simple tree path from u to v as $T[u, v]$. The expressions $T[u, v)$ and $T(u, v]$ have the obvious meaning (i.e., the vertex on the side of the parenthesis is excluded from the tree path). Furthermore, we let $T(v)$ denote the subtree of T rooted at vertex v. We identify vertices in G by their DFS number, i.e., the order in which they were discovered by the search. Hence,

$u \leq v$ means that vertex u was discovered before v. The edges of the tree are called tree-edges, and the edges of G which are not tree-edges are called back-edges, as their endpoints have ancestor-descendant relation in T. When we write (u, v) to denote a back-edge, we always mean that $v \leq u$, i.e., u is an descendant of v in T.

The following is a list of some concepts that are defined on the structure given by the DFS and are essential to the algorithms we are going to describe in the next chapters. See Figure 3.1 for an illustration.

- *low*(v) := *min*$\{u \mid$ *there exists a back-edge* (x, u), *with* x *a descendant of* $v\}$
- *high*(v) := *max*$\{u \mid u$ *is a proper ancestor of* v *and there exists a back-edge* (x, u), *with* x *a descendant of* $v\}$
- *high*$_p(v)$:= *max*$\{u \mid u$ *is a proper ancestor of* $p(v)$ *and there exists a back-edge* (x, u), *with* x *a descendant of* $v\}$
- $M(v)$:= *the nearest common ancestor of all descendants of* v *which are connected to a proper ancestor of* v *with a back-edge.*
- $M_p(v)$:= *the nearest common ancestor of all descendants of* v *which are connected to a proper ancestor of* $p(v)$ *with a back-edge.*

low(v), *high*(v) and $M(v)$ are well-defined for every vertex $v \neq r$ if G is 2-edge-connected, and *high*$_p(v)$ and $M_p(v)$ are well-defined for every vertex v different from r and the child of r if G is 2-vertex-connected. The *low* points have been defined in [12], and can be computed easily in linear time with a recursive algorithm (in a bottom-up fashion). As for the other concepts, they are defined here for the first time, as far as I know. Thus, in the following two sections we will describe algorithms which compute all *high*, *high*$_p$, M and M_p (for all the vertices on which they are defined), in linear time.

3.2 Computing all *high*(v) and *high*$_p(v)$ in linear time

The basic idea to compute all *high*(v) (for $v \neq r$) is to do the following: We process the back-edges (u, v) in decreasing order with respect to their lower end v. When we process (u, v), we ascent the path $T[u, v]$, and for each visited vertex x such that

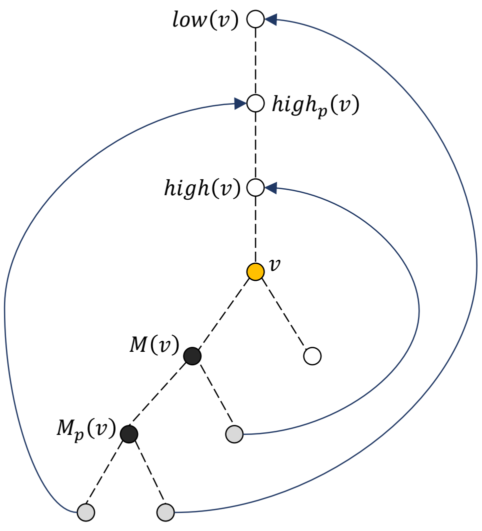


Figure 3.1: Concepts defined on the structure of the DFS tree that are essential to our algorithm. Dashed lines correspond to DFS tree paths. Back-edges are shown directed from descendant to ancestor.

high$[x]$ is still undefined, we set *high*$[x] \leftarrow v$. See Algorithm 3.1. It should be clear that this process, which forms the basis of our linear-time algorithm, computes all *high*(v), for $v \neq r$, correctly.

In order to achieve linear running time, we have to be able, when we consider a back-edge (u, v), to bypass all vertices on the path $T[u, v]$ whose *high* value has been computed. To that end, it suffices to know, for every vertex x in $T[u, v]$, the nearest ancestor of x whose *high* value is still null. We can achieve this by applying a disjoint-set-union (DSU) structure [2].

Specifically, we maintain a forest F that is a subgraph of T, subject to the following operations:

link(x, y): Adds the edge (x, y) into the forest F.

find(x): Return the root of the tree in F that contains x.

Let F_x denote the tree of F that contains a vertex x. Initially, F contains no edges, so x is the unique vertex in F_x. In our algorithm, the link operation always adds some tree edge $(u, p(u))$ to F, so the invariant that F is a subgraph of T is maintained. This is implemented by uniting the corresponding sets of u and $p(u)$ in the underlying

Algorithm 3.1 SimpleHigh

```
1: for all vertices $v \neq r$ do
2:     sort the back-edges $(u, v)$ in decreasing order w.r.t. to their lower end $v$
3: end for
4: for all back-edges $(u, v)$ do
5:     if $high[u] = null$ then
6:         $high[u] \leftarrow v$
7:     end if
8:     $u \leftarrow p(u)$
9: end for
```

DSU structure, and setting the root of of $F_{p(u)}$ as the representative of the resulting set. Then, *find*(u) returns the root of F_u, which will be the nearest ancestor of u in T whose *high* value is still null. Algorithm 3.2 gives a fast algorithm for computing *high*(v), for every vertex $v \neq r$.

The next lemma summarizes the properties of Algorithm 3.2.

Lemma 3.1. *Algorithm 3.2 is correct. Furthermore, it will perform $n-1$ link and $2m-n+1$ find operations on a 2-vertex-connected graph with n vertices and m edges.*

Proof. Let B be the sorted list of the back-edges. (Notice that B contains $m - n + 1$ edges.) We will prove the theorem inductively by showing that, for every t in $\{0, \ldots, m - n\}$: **if**, after having run the algorithm for the first t back-edges, we now have that, (1) for every vertex x, *find*(x) returns the nearest ancestor of x whose high value is still null, (2) for every back-edge (u, v) in $B[1, t]$, *high*$[x]$ has been computed correctly for every x in $T[u, v)$, and the *high* value of every other vertex, which does not belong to such a set, is still null, and (3) every set that has been formed due to the *link* operations that have been performed is a subtree of T, of whose members only its root has its $high$ value still set to null, **then**, if we run the algorithm once more for the $t+1$ back-edge, properties (1), (2) (for $t+1$), and (3) will still hold true.

For the basis of our induction, let us note that the premise of the inductive proposition for $t = 0$ is trivially true: Before we have begun traversing B, the set containing x is a singleton, *find*$(x) = x$, and *high*$[x]$ is null, for every vertex x. Now, suppose the premise of the inductive proposition is true for some t in $\{0, \ldots, m-n\}$, and let (u, v) be the $t + 1$ back-edge. Let $x_1, \ldots, x_k$, in decreasing order, be the vertices in $T[u, v]$ whose *high* value is still null. (Note that, since B is sorted in decreasing order w.r.t

Algorithm 3.2 FastHigh

```
1: initialize a forest $F$ with $V(F) = V(T)$ and $E(F) = \emptyset$
2: for all vertices $v \neq r$ do
3:     set $high[v] \leftarrow null$
4: end for
5: sort the back-edges $(u, v)$ in decreasing order w.r.t. to their lower end $v$
6: for all back-edges $(u, v)$ do
7:     $u \leftarrow find(u)$
8:     while $u > v$ do
9:         $high[u] \leftarrow v$
10:        $next \leftarrow find(p(u))$
11:        $link(u, p(u))$
12:        $u \leftarrow next$
13:    end while
14: end for
```

the lower end-point of its elements, we have $x_k = v$.) We observe two facts. First, by (1), we have that $x_1 = find(u)$, and $x_i = find(p(x_{i-1}))$, for $i = 2, \ldots, k$. Second, (2) implies that the correct *high* value of x_i, for every $i = 1, \ldots, k-1$, is v (although now it is still set to null). From these two facts we can see that, in order to prove that our algorithm is going to correctly compute the values $high[x_i]$, for $i = 1, \ldots, k-1$, and not mess with those that have already been computed, it is sufficient to show that the function $find(p(x))$, in line 10, will return, every time it is invoked, the closest ancestor of $p(x)$ whose *high* value is still set to null - despite all the *link* operations which might have been performed in the meantime. To see this, observe that (1) and (3) imply that, for every $i = 1, \ldots, k-1$, x_i and $p(x_i)$ belong to different sets (since x_i, having its *high* value still set to null, is the root of the set it belongs to). From this we conclude, that after linking x_i with $p(x_i)$, x_j and $p(x_j)$ still belong to different sets, for every $j = i+1, \ldots, k-1$. It should be clear now that, by executing our algorithm for the $t+1$ back-edge, only the *high* values of $x_1, \ldots, x_{k-1}$ are going to be affected (and computed correctly). This shows that (2) (for $t+1$) still holds true. We also see that all the sets that have been formed due to the *link* operations that have been performed are still subtrees of T, since every such operation is linking a vertex with its parent. Now, let x be a vertex that belongs to one of the sets that have been affected by the *link* operations that have been performed during the execution of the algorithm for the $t+1$ back-edge. By (1), this means that, before running the algorithm for this back-edge, *find*$(x) = x_i$, for some $i = 1, \dots, k$. We conclude, that after running the algorithm for the $t+1$ back-edge, the closest ancestor of x that has its *high* value still set to null is v (since, now, every vertex in $T[u, v)$ has its *high* value computed). This shows that (1) still holds true. Furthermore, this also shows that every vertex in $T[x, v]$ is part of the same set. We conclude that the root of the set which contains x is v. Thus we have shown that (3) still holds true. (We do not have to consider the vertices whose set has not been affected by the *link* operations.)

Thus we have proved that, since the premise of the inductive proposition for $t = 0$ is true, (2) in the conclusion of the inductive proposition for $t = m - n$ is also true. In other words, our algorithm computes correctly the *high* value of every x which belongs to a set of the form $T[u, v)$, for some back-edge (u, v). Since the graph is 2-vertex-connected, every vertex $x \neq r$ belongs to such a set. Furthermore, after the execution of the algorithm, precisely $n-1$ *link* operations (one for every vertex $x \neq r$), and $2m - n - 1$ *find* operations (one for every end-point of every back-edge, and one for every vertex $x \neq r$) will have been performed. □

Since all the *link* operations we perform are of the type *link*$(u, p(u))$, and the total number of *link* and *find* operations performed is $O(m+n)$, we may use the static tree DSU data structure of Gabow and Tarjan [2] to achieve linear running time.

Finally, we note that the algorithm for computing all *high*$_p(v)$ is almost identical to Algorithm 3.2. The only difference is in line 8, where we have to replace "**while** $u > v$" with "**while** $p(u) > v$". The proof of correctness and linearity is essentially the same.

3.3 Computing all $M(v)$ and $M_p(v)$ in linear time

Recall that $M(v)$ is the nearest common ancestor of all descendants of v that are connected with a back-edge to a proper ancestor of v, while $M_p(v)$ is the nearest common ancestor of all descendants of v that are connected with a back-edge to a proper ancestor of $p(v)$.

Before we describe our algorithm for the computation of $M(v)$ (and $M_p(v)$), we state a lemma that will be useful in what follows.

Lemma 3.2. *Let u and v be such that v is an ancestor of u and $M(v)$ is a descendant of u. Then $M(v)$ is a descendant of $M(u)$.*

Proof. Let $e = (x, y)$ be a back-edge with x a descendant of v and y a proper ancestor of v. Since v is an ancestor of u, y is a proper ancestor of u. And since $M(v)$ is a descendant of u, x is a descendant of u. Now, it is an immediate consequence of the definition of $M(u)$, that $M(u)$ is an ancestor of x. Since e was chosen arbitrarily, we conclude that $M(u)$ is an ancestor of $M(v)$. □

Note 3.1. We note that the lemma still holds if we replace $M(v)$ with $M_p(v)$.

Our algorithm for the computation of $M(v)$ works recursively on the children of v. So, let v be a vertex (different from r). We define $l(v) = min\{\{v\} \cup \{u \mid \text{there exists a back-edge } (v, u)\}\}$. (Of course, we have $low(v) \leq l(v)$.) Now, if $l(v) < v$, we have $M(v) = v$. (Let us note here that, if v is a leaf, then $l(v) < v$ is necessarily true, since the graph is 2-vertex-connected, and therefore we may set $M(v) = v$ for all vertices v that are leaves.) Furthermore, if there exist two children c, c' of v such that $low(c) < v$ and $low(c') < v$, then, again, $M(v) = v$. The difficulty arises when there is only one child c of v with the property $low(c) < v$ (one such child of v must of necessity exist, since the graph is 2-vertex-connected), in which case $M(v)$ is a descendant of c, and, therefore, $M(v)$ is a descendant of $M(c)$ by Lemma 3.2. In this case, we repeat the same process in $M(c)$: we shall test whether $l(M(c)) < v$ or whether there exists only one child d of $M(c)$ such that $low(d) < v$, in which case we repeat the same process in $M(d)$, and so on.

Now, we claim that a careful implementation of the above procedure yields a linear-time algorithm for the computation of $M(v)$, for all vertices $v \neq r$. To that end, it suffices to store, for every vertex v that is not a leaf of T, two pointers, $L(v)$ and $R(v)$, on the list of the children of v. Initially, $L(v)$ points to the first child c of v that has $low(c) < v$, and $R(v)$ points to the last child c' of v that has $low(c') < v$. Our algorithm works in a bottom-up fashion. Provided we have computed $M(u)$ for every descendant u of v, we execute Procedure 3.3.

Lemma 3.3. *By executing Procedure 3.3, for all vertices $v \neq r$, in bottom-up fashion of T, we can compute all $M(v)$ in linear-time.*

Proof. To prove correctness, it is sufficient show that, for the computation of $M(v)$, if $M(v)$ lies in $T(m)$, for some descendant m of v, and $l(m) \geq v$, then every back-edge

Algorithm 3.3 Procedure FindM

```
1: if $l(v) < v$ then
2:     return $v$
3: end if
4: if $L[v] \neq R[v]$ then
5:     return $v$
6: end if
7: $m \leftarrow M[L[v]]$
8: if $l(m) < v$ then
9:     return $m$
10: end if
11: while $low(L[m]) \geq v$ do
12:     $L[m] \leftarrow$ next child of $m$
13: end while
14: while $low(R[m]) \geq v$ do
15:     $R[m] \leftarrow$ previous child of $m$
16: end while
17: if $L[m] \neq R[m]$ then
18:     return $m$
19: end if
20: $m \leftarrow M[L[m]]$
21: goto line 8
```

that starts from $T(v)$ and ends in a proper ancestor of v has its starting-point in a subtree of the form $T(c)$, where c is a child of m between $L[m]$ and $R[m]$. It's easy to see this inductively: that is, let v be a vertex, all of whose descendants had this property as the algorithm was running. Now, suppose that m is a descendant of v such that $M(v)$ lies in $T(m)$. If $L[m]$ points to the first child of m and $R[m]$ to the last child of m, then there is nothing to prove. But if one of these two pointers was moved (in lines 12 or 15) during the execution of the algorithm, this means, thanks to the inductive hypothesis, that for an ancestor x of m which is also a proper descendant of v it is true that every back-edge that starts from $T(x)$ and ends in a proper ancestor of x has its starting-point in a subtree of the form $T(c)$, for some child c of m between $L[m]$ and $R[m]$. Now we see why every back-edge that starts from $T(v)$ and ends in a proper ancestor of v has its starting-point in a subtree of the same form: for if this is not the case, and there exists a back-edge that starts from $T(d)$, for some child d of v which is not between $L[m]$ and $R[m]$, and ends in a proper ancestor of v, then this is also a back-edge that starts from $T(x)$ and ends in a proper ancestor of x - a contradiction.

Now, to prove linearity, we note that the only way our algorithm could be making an excessive amount of steps, would be by visiting some vertices a lot of times, when it recursively descends to the descendants of some vertices, in order to compute their M value. So we define, for every vertex v, the (possibly empty) list $S(v) = \{m_1, \dots, m_{k_v}\}$ of the proper descendants of v that the algorithm had to visit in order to compute $M(v)$, sorted by the order of visit (i.e. ordered increasingly). We will prove linearity by showing that two such distinct lists can meet only in their last element. Equivalently, we may show that a non-last member m of such a list (let us call it: an intermediary member), can appear only in that same list. So, let m be an intermediary member of a list $S(v)$, for some vertex v, and let v be the first vertex in whose list m appears as an intermediary member (that is, there is no proper descendant of v in whose list m appears as an intermediary member). We note, that, since m is an intermediary member of $S(v)$, $M(v)$ is a proper descendant of m. Now, suppose that there exists a proper ancestor u of v such that m is a member of $S(u)$, and let u be the closest proper ancestor of v that has this property. Then we have $l(u) = u$, and there is a unique child c of u with the property $low(c) < u$. Furthermore, $M(c)$ (the first member of $S(u)$) belongs to $T[m, c]$. But $M(c)$ does not belong to $T[m, v]$: for otherwise, since c is an ancestor of v, Lemma 3.2 implies that $M(c)$ is a descendant of $M(v)$, which is a proper descendant of m. We conclude, that $M(c)$, the first member of $S(u)$, is an ancestor of v. Now, continuing in this fashion, (i.e. considering the unique child d of $M(c)$ that has the property $low(d) < u$, so that $M(d)$ is the second member of $S(u)$), we see that the members of $S(u)$ are either ancestors of v or descendants of $M(v)$. A contradiction. $\square$

We use a similar algorithm in order to compute all $M_p(v)$. The only change we have to make in Procedure 3.3 is to replace every comparison to v with a comparison to $p(v)$. The proof of correctness and linearity is essentially the same.

Chapter 4

Counting cut-pairs in linear time

4.1 Introduction

4.2 The case back-edge - tree-edge

4.3 The case where both edges are tree-edges

4.1 Introduction

Let $G = (V, E)$ be a 2-edge-connected undirected graph. For every e in E we define *count*$(e) := \{e' \in E \mid \{e, e'\}$ is a cut-pair $\}$. Thus, an edge e belong to a cut-pair if and only if *count*$(e) > 0$. In this Chapter we show how to compute all *count*(e) in linear time. Although Tsin [13] and other researchers have developed linear-time algorithms to solve the problem of determinind the edges that belong to a cut-pair in a 2-edge-connected undirected graph (and an extension of their algorithms can also solve the counting problem), our approach is conceptually simple and easy to implement in practise. In section 2.2 we saw how we can apply such an algorithm to find all twinless strong bridges in a twinless strongly connected digraph in linear time. Furthermore, by Lemma 2.2, we know how to use the parameters *count*(e) to compute in linear time the number of TSCCs of a twinless strongly connected digraph after the removal of a twinless strong bridge which is not a strong bridge.

To compute all *count*(e), we will work on the tree structure T, with root r, provided by a DFS on G. Then, if $\{e, e'\}$ is a cut-pair of edges, either one of them is a back-edge and the other one is a tree-edge, or both of them are tree-edges. (It cannot be the case that both of them are back-edges, since then their removal would not disconnect the graph.) Furthermore, in the case that both of them are tree-edges, we have the following:

Lemma 4.1. *If $\{e, e'\}$ is a cut-pair such that both e and e' are tree-edges, then they both lie on the simple tree path $T[u, r]$, for some vertex u. (See Figure 4.1.)*

Proof. Since both e and e' are tree-edges, there exist vertices u and v, such that $e = (u, p(u))$ and $e' = (v, p(v))$. Since the graph is 2-edge-connected, u is distinct from v, and let's assume, without loss of generality, that $u > v$. Now, suppose that e and e' are not part of the simple tree path $T[u, r]$. Then v is not an ancestor of u. Furthermore, since $u > v$, v is not a descendant of u either. Now, remove $\{e, e'\}$ from the graph. We note three facts. First, since v is not a descendant of u, $T(u)$ remains connected. Second, since the graph is 2-edge-connected, there exists a back-edge connecting a vertex from $T(u)$ with a proper ancestor x of u. Third, since v is not an ancestor of u, the vertices on the simple tree path $T[p(u), x]$ remain connected. These three facts imply that u remains connected with $p(u)$, and therefore $\{e, e'\}$ is not a cut-pair. A contradiction. □

Thus we have two distinct cases to consider, and we will compute *count*(e) by counting the number of cut-pairs $\{e, e'\}$ in each case. We will handle these cases separately, by providing a specific algorithm for each one of them. We shall begin with the case that one of those edges is a back-edge, since this is the easiest to handle. We suppose that all *count*(e) have been initialized to zero.

4.2 The case back-edge - tree-edge

Our algorithm for this case is based on the following observation:

Proposition 4.1. *Let e be a back-edge and u a vertex distinct from r. Then the pair $\{e, (u, p(u))\}$ is a cut-pair if and only if e starts from $T(u)$, ends in a proper ancestor of u, and is the only back-edge with this property.*

Proof. ($\Rightarrow$) Since the graph is 2-edge-connected, there exists at least one back-edge e'' with the property that e'' connects a descendant of u with a proper ancestor of u. Supposing $e'' \neq e$, we see that $\{e, (u, p(u))\}$ cannot be a cut-pair: since, in this case,

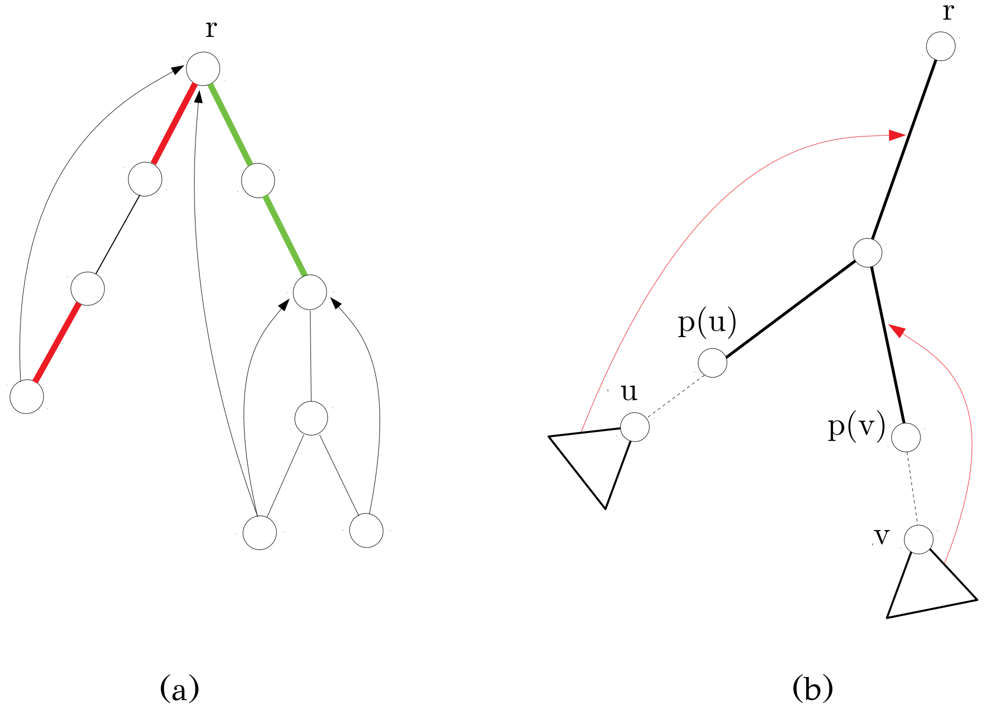


Figure 4.1: (a) An example of a DFS tree containing two cut-pairs of tree-edges (shown in red and green). (b) An illustration of the argument in 4.1: since the graph is 2-edge-connected, there exist a back-edge connecting the subtree $T(u)$ with $T(u, r]$, and a back-edge connecting the subtree $T(v)$ with $T(v, r]$. The pair of tree-edges $\{(u, p(u)), (v, p(v))\}$ is not a cut-pair.

by removing $\{e, (u, p(u))\}$, u remains connected with $p(u)$. That's a contradiction.
($\Leftarrow$) By removing the edge $(u, p(u))$, all the paths that connect u with $p(u)$ must necessarily use a back-edge connecting a vertex from $T(u)$ with a proper ancestor of u. If e is the only back-edge with this property, then $\{e, (u, p(u))\}$ is a cut-pair. $\square$

This implies that, for every vertex u, there exists at most one cut-pair of the form $\{e, (u, p(u))\}$, where e is a back-edge, and it immediately suggests an algorithm for determining whether such a cut-pair exists. We only have to count, for every vertex u ($\neq r$), the number $b\_count(u)$:= #{back-edges that start from $T(u)$ and end in a proper ancestor of u}. To do this, we define, for every vertex u, $up(u)$:= #{back-edges that start from u and end in an ancestor of u}, and $down(u)$:= #{back-edges that start from $T(u)$ and end in u}. All $up(u)$ and $down(u)$ can be computed easily in linear time. Now $b\_count(u)$ can be computed recursively: if $c_1, \ldots, c_k$ are the children of u, then $b\_count(u) = up(u) + b\_count(c_1) + \ldots + b\_count(c_k) - down(u)$; and if u is childless,

$b\_count(u) = up(u)$.

Now, for every vertex u that has $b\_count(u) = 1$, we set $count[(u, p(u))] \leftarrow count[(u, p(u))]$ $+1$. Furthermore, in this case, there exists only one back-edge (x, y) such that x is a descendant of u and y is a proper ancestor of u, and so we have $x = M(u)$ and $y = low(u)$. Thus, we also set $count[(M(u), low(u))] \leftarrow count[(M(u), low(u))] + 1$.

4.3 The case where both edges are tree-edges

Our algorithm for this case is based on the following observation:

Proposition 4.2. *Let u, v be two vertices such that v is an ancestor of u. Then $\{(u, p(u)),$ $(v, p(v))\}$ is a cut-pair if and only if v is a proper ancestor of u with $M(u) = M(v)$ and $high(u) < v$. (See Figure 4.2.)*

Proof. ($\Rightarrow$) Since the graph is 2-edge-connected, the removal of one edge is not sufficient to disconnect the graph, and therefore v must be a proper ancestor of u. Now, let (x, y) be a back-edge such that x is a descendant of u and y is a proper ancestor of u. Since u is a descendant of v, x is also a descendant of v. Furthermore, we notice that, since $\{(u, p(u)), (v, p(v))\}$ is a cut-pair, y is a proper ancestor of v. (For otherwise, by removing $(u, p(u))$ and $(v, p(v))$, $T(u)$ remains connected (since v is an ancestor of u), the vertices in the simple tree path $T[p(u), y]$ remain connected (since v is an ancestor of y), and, therefore, the existence of the back-edge (x, y) implies that u remains connected with $p(u)$.) This shows that $M(v)$ is an ancestor of $M(u)$, and $high(u)$ is a proper ancestor of v. Conversely, let (x, y) be a back-edge such that x is a descendant of v and y is a proper ancestor of v. We observe that, since $\{(u, p(u)), (v, p(v))\}$ is a cut-pair, x must be a descendant of u. (For otherwise, by removing $(u, p(u))$ and $(v, p(v))$, the simple tree paths $T[x, v]$ and $T[p(v), y]$ have not been affected (since u is a descendant of x), and, therefore, the existence of the back-edge (x, y) implies that v remains connected with $p(v)$.) Furthermore, since v is an ancestor of u, y is a proper ancestor of u. This shows that $M(u)$ is an ancestor of $M(v)$. We conclude that $M(u) = M(v)$.

($\Leftarrow$) Remove the edges $(u, p(u))$ and $(v, p(v))$. Now, if there exists a path connecting u with $p(u)$, this path must use a back-edge (x, y) such that either (1) x is in $T(u)$ and y in $T[p(u), v]$, or (2) x is a descendant of a vertex in $T[p(u), v]$, but not a descendant of u, and y is a proper ancestor of v. (1) cannot be true, since $high(u) < v$. (2) cannot be true, since $M(u) = M(v)$, and therefore $M(v)$ is a descendant of u, and therefore x is a descendant of u. We conclude that the pair $\{(u, p(u)), (v, p(v))\}$ is a cut-pair. $\square$

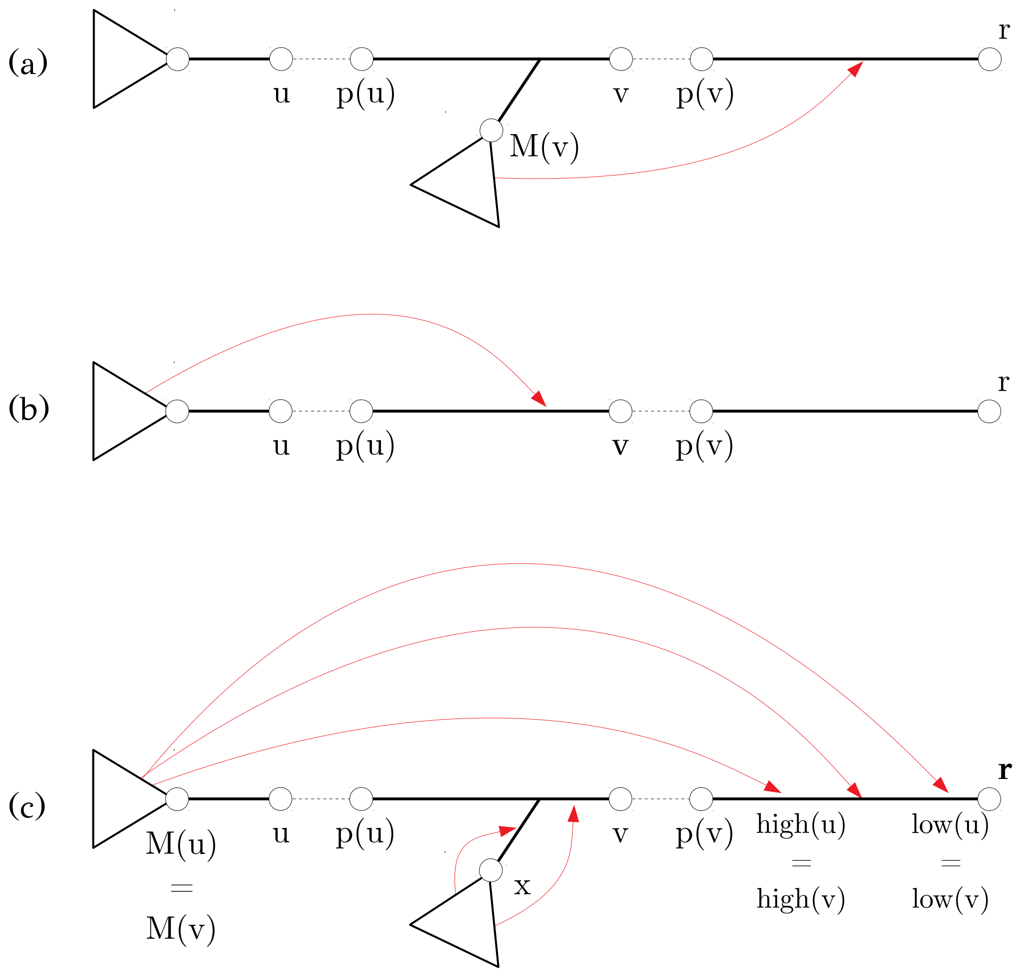


Figure 4.2: This is an illustration of the necessary and sufficient conditions for two tree-edges (that belong to a simple tree path starting from the root) to form a cut-pair. From (a) it is obvious that $M(v)$ must be in $T(u)$. From (b) its is obvious that $high(u)$ must be a proper ancestor of v. In (c) we have that $M(v) = M(u)$ and $high(u) < v$. Then $\{(u, p(u)), (v, p(v))\}$ is a cut-pair. By Lemma 4.2, we have $high(u) = high(v)$. Since $M(v)$ is in $T(u)$, for every x which is a descendant of a vertex in $T(u, v]$, but not a descendant of u, we have $high(x) \geq v$.

Algorithm 4.1 describes how we can compute, for every vertex u, the number of cut-pairs of the form $\{(u, p(u)), (v, p(v))\}$. To prove correctness, we will need the following:

Lemma 4.2. *Let u, v be two vertices such that $M(u) = M(v)$, v is an ancestor of u, and $high(u) < v$. Then $high(u) = high(v)$.*

Proof. Let (x, y) be a back-edge such that x is a descendant of u and y is a proper ancestor of u. Then, since u is a descendant of v, x is also a descendant of v. Furthermore, $high(u) < v$ implies that y is a proper ancestor of v. This shows that $high(u) \leq high(v)$. Conversely, let (x, y) be a back-edge such that x is a descendant of v and y is a proper ancestor of v. Then, since $M(v) = M(u)$, $M(v)$ is a descendant of u, and therefore x is also a descendant of u. Furthermore, since v is an ancestor of u, y is a proper ancestor of u. This shows that $high(v) \leq high(u)$. We conclude that $high(u) = high(v)$. □

Now, we let $S(u)$, for every vertex $u \neq r$, denote the set $\{u\} \cup \{v \mid \{(u, p(u)), (v, p(v))\}$ is a cut-pair $\}$. Then we have the following:

Proposition 4.3. *For every $v \in S(u)$ we have $S(v) = S(u)$.*

Note 4.1. In other words, this Proposition says that the binary relation "e forms a cut-pair with e'", defined on the set of tree-edges, is transitive.

Proof. Let u be a vertex. We will show that all vertices in $S(u)$ have the same *high* point. So let v be a member of $S(u)$. By Lemma 4.1, we have that v is either an ancestor of u or a descendant of u. Suppose that v is an ancestor of u. Then, Proposition 4.2 implies that $high(u) < v$. By the same Proposition, we also have $M(u) = M(v)$. By Lemma 4.2, these two facts imply that $high(u) = high(v)$. Now, if v is a descendant of u, the same argument (with a reversal of the roles of u and v) shows that $high(u) = high(v)$.

Now let v be a member of $S(u)$ and w a member of $S(v)$. Since $high(w) = high(v)$ and $high(v) = high(u)$, we conclude that $high(w) = high(u)$, and therefore (by the definition of *high*) $high(w) < u$ and $high(u) < w$. By Proposition 4.2, we have $M(w) = M(v)$ and $M(v) = M(u)$, and thus we conclude that $M(w) = M(u)$. Now, since u and w have a common descendant (that is, $M(w)$), they are related as ancestor-descendant. If u is a proper ancestor or a proper descendant of w, then, since $M(u) = M(w)$ and $high(u) < w$ and $high(w) < u$, by Proposition 4.2 we have that w is in $S(u)$. Otherwise, $w = u$, and therefore w is in $S(u)$ (by definition). Thus we have $S(v) \subseteq S(u)$. The reverse inclusion is proved by a symmetric argument. □

Theorem 4.1. *Algorithm 4.1 is correct.*

Proof. According to Proposition 4.2, all cut-pairs of the form $\{(u,p(u)),(v,p(v))\}$ have $M(u)=M(v)$. Therefore, in order to count all these cut-pairs, it is sufficient to focus our attention on the lists $M^{-1}(m)$, for all vertices m ($\neq r$), to find therein vertices u and v such that $\{(u,p(u)),(v,p(v))\}$ is a cut-pair. Now, suppose that we have all these lists computed and their elements sorted in decreasing order, and let m be a vertex. Let u be an element of $M^{-1}(m)$ which is maximal in $S(u)$ (i.e. if there exists a v such that $\{(u,p(u)),(v,p(v))\}$ is a cut-pair, then v is a proper ancestor of u). Then, by Proposition 4.2, we have $S(u)=M^{-1}(m)\cap T[u,\textit{high}(u))$. Furthermore, by Proposition 4.3, we have that, for every v in $S(u)$, $S(v)=S(u)$. This explains why Algorithm 4.1 works. We start with the first element u of $M^{-1}(m)$, and we traverse the list $M^{-1}(m)$ until we reach a vertex v such that $v\leq \textit{high}(u)$ (or until we run out of elements). While doing that, we keep a counter *n_edges* of the elements in $M^{-1}(m)\cap T(u,\textit{high}(u))$. Then we traverse the segment $M^{-1}(m)\cap T[u,\textit{high}(u))$ of the list $M^{-1}(m)$ again, and, for every w in $M^{-1}(m)\cap T[u,\textit{high}(u))$, we set $\textit{count}[(w,p(w))] := \textit{count}[(w,p(w))] + \textit{n\_edges}$. Then we repeat the same process from v, until we reach the end of $M^{-1}(m)$. $\square$

Algorithm 4.1 Counting cut-pairs of tree-edges

```
1: calculate all lists $M^{-1}(m)$, for all vertices $m$, and bucket-sort their elements in
   decreasing order
2: for all vertices $m$ do
3:   $u \leftarrow$ first element of $M^{-1}(m)$
4:   while $u \neq \emptyset$ do
5:     $v \leftarrow$ successor of $u$ in $M^{-1}(m)$
6:     $n\_edges \leftarrow 0$
7:     while $v \neq \emptyset$ and $high(u) < v$ do
8:       $n\_edges \leftarrow n\_edges + 1$
9:       $v \leftarrow$ next element of $M^{-1}(m)$
10:    end while
11:    $v \leftarrow u$
12:    while $v \neq \emptyset$ and $high(u) < v$ do
13:      $count[(v, p(v))] \leftarrow count[(v, p(v))] + n\_edges$
14:      $v \leftarrow$ next element of $M^{-1}(m)$
15:    end while
16:    $u \leftarrow v$
17:  end while
18: end for
```

Chapter 5

Counting vertex-edge cut-pairs in linear time

5.1 Introduction

5.2 The case where e is a back-edge

5.3 The case where e lies on the simple tree path $T[v, r]$

5.4 The case where e lies in $T(v)$

5.1 Introduction

Let $G = (V, E)$ be a 2-vertex-connected undirected graph. For every v in V we define *count*$(v) := \{e \in E \mid \{v, e\}$ is a cut-pair $\}$. Thus, a vertex v belong to a vertex-edge cut-pair if and only if *count*$(v) > 0$. In this Chapter we show how to compute all *count*(v) in linear time. Although there are known methods to solve the problem of computing all *count*(v) in linear-time (by exploiting properties of SPQR-trees [5]), our approach is conceptually simple and easy to implement in practise. In section 2.3 we saw how we can apply such an algorithm to find all twinless strong articulation points in a twinless strongly connected digraph in linear time. Furthermore, by Lemma 2.4, we know how to use the parameters *count*(v) to compute in linear time the number of TSCCs of a twinless strongly connected digraph after the removal of a twinless strong articulation point which is not a strong articulation point.

Now, to compute all *count*(v), we will work on the tree structure T, with root r, provided by a DFS on G. Then, if $\{v, e\}$ is a vertex-edge cut-pair, e can either be a back-edge, or a tree-edge. Furthemore, in the case that e is a tree-edge, we have the following:

Lemma 5.1. *If $\{v, e\}$ is a cut-pair such that e is a tree-edge, then e either lies in $T(v)$ or on the simple tree path $T[v, r]$.*

Proof. Suppose that e is neither in $T(v)$ nor on the path $T[v, r]$. Since e is a tree-edge, it has the form $(u, p(u))$, for some vertex u. Since H is 2-vertex-connected, there exists a back-edge $e' = (x, y)$ joining a vertex x from $T(u)$ with a proper ancestor y of u. Now, remove v and e from the graph. Since e does not lie on the path $T[v, r]$, v is not a descendant of u, and therefore $T(u)$ remains connected. Furthermore, since e is not in $T(v)$, y remains connected with $p(u)$. The existence of e' implies that u remains connected with $p(u)$ - a contradiction. □

Thus we have three distinct cases in total, and we will compute *count*(v) by counting the cut-pairs $\{v, e\}$ in each case. We will handle these cases separately, by providing a specific algorithm for each one of them, based on some simple observations like Lemma 5.1. The linearity of these algorithms will be clear.

Now, we shall begin with the case where e is a back-edge, since this is the easiest to handle. We suppose that all *count*(v) have been initialized to zero.

5.2 The case where e is a back-edge

Proposition 5.1. *If $\{v, e\}$ is a cut-pair such that e is a back-edge, then e starts from the subtree $T(c)$ of a child c of v, ends in a proper ancestor of v, and is the only back-edge that starts from $T(c)$ and ends in a proper ancestor of v. Conversely, if e is such a back-edge, then $\{v, e\}$ is a cut-pair.*

This immediately suggests an algorithm for counting all such cut-pairs. We only have to count, for every vertex c ($\neq r$ or the child of r), the number *b_count*(c) := #{back-edges that start from $T(c)$ and end in a proper ancestor of $p(c)$}. To do this efficiently, we define, for every vertex v, *up*(v) := #{back-edges that start from v and end in an ancestor of v}, and, for every child c of v (if it has any), *down*(v, c) := #{back-edges that start from $T(c)$ and end in v}. See Algorithm 5.1.

Algorithm 5.1 Calculating $up(c)$ and $down(v, c)$

```
1: initialize all $up(v)$ and $down(v, c)$ to 0
2: sort the back-edges $(u, v)$ in increasing order w.r.t. their higher end $u$
3: sort the list of the children of every vertex in increasing order
4: for all vertices $v$ do
5:     if $v$ is not childless then
6:         $c_v \leftarrow$ first child of $v$
7:     end if
8: end for
9: for all back-edges $(u, v)$ do
10:    $up(u) \leftarrow up(u) + 1$
11:    while $c_v$ is not an ancestor of $u$ do
12:        $c_v \leftarrow$next child of $v$
13:    end while
14:    $down(v, c_v) \leftarrow down(v, c_v) + 1$
15: end for
```

Now, *b_count*(c) can be computed recursively: if $d_1, \ldots, d_k$ are the children of c, then *b_count*$(c) = up(c) +$ *b_count*$(d_1) + \ldots +$ *b_count*$(d_k) - down(p(c), c)$; and if c is childless, *b_count*$(c) = up(c)$. Finally, the number of vertex-edge cut-pairs $\{v, e\}$ where e is a back-edge, equals the number of children c of v that have *b_count*$(c) = 1$.

5.3 The case where e lies on the simple tree path $T[v, r]$

Let $\{v, e\}$ be a vertex-edge cut-pair such that e is part of the simple tree path $T[v, r]$. Then there exists a vertex u which is a proper ancestor of v and such that $e = (u, p(u))$. We observe that all back-edges that start from $T(u)$ and end in a proper ancestor of u must necessarily start from $T(v)$. In other words, $M(u)$ is a descendant of v. Here we further distinguish two cases, depending on whether $M(u)$ is a proper descendant of v.

5.3.1 The case $M(u) = v$

Our algorithm for this case is based on the following observation:

Proposition 5.2. *Let $c_1, \ldots, c_k$ be the children of v (if it has any), and let $\{v, (u, p(u))\}$ be a cut-pair such that u is an ancestor of v with $M(u) = v$. Then u does not belong in any set of the form $T[high_p(c_i), low(c_i))$, for $i = 1, \ldots, k$. Conversely, given that u is a proper ancestor of v such that $M(u) = v$, and given also that u does not belong in any set of the form $T[high_p(c_i), low(c_i))$, for $i = 1, \ldots, k$, we may conclude that the pair $\{v, (u, p(u))\}$ is a cut-pair. (See Figure 5.1.)*

Proof. ($\Rightarrow$) Suppose that u belongs to $T[high_p(c), low(c))$, for some child c of v. Then $high_p(c)$ is a descendant of u, $low(c)$ is an ancestor of $p(c)$, and both of these vertices are proper ancestors of v. Now, there exists a back-edge $(x, high_p(c))$, with x in $T(c)$. There also exists a back edge $(x', low(c))$, with x' in $T(c)$. Therefore, it should be clear that the removal of both v and $(u, p(u))$ does not disconnect u from $p(u)$: for, even after this removal, both u and $p(u)$ remain connected with the subtree $T(c)$. This contradicts the fact that $\{v, (u, p(u))\}$ is a cut-pair.
($\Leftarrow$) Remove the vertex v and the edge $(u, p(u))$. $M(u) = v$ means that there are no back-edges (x, y) with x being a descendant of a vertex in $T(v, u]$, but not a descendant of v, and y an ancestor of $p(u)$. Therefore, if u remains connected with $p(u)$, they must both be connected with a subtree $T(c)$, for some child c of v. Furthermore, if such is the case, there must exist two back-edges, (x, y) and (x', y'), and a child c of v, such that both x and x' are in $T(c)$, y is proper ancestor of v and a descendant of u, and y' is an ancestor of $p(u)$. But this means that $u \leq high_p(c)$ and $low(c) \leq p(u)$. In other words: u is in $T[high_p(c), low(c))$ - a contradiction. $\square$

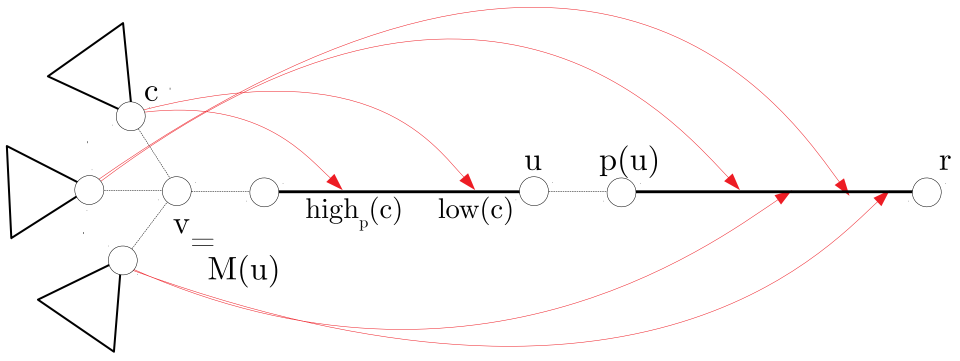


Figure 5.1: $M(u) = v$, and v forms a vertex-edge cut-pair with $(u, p(u))$. Observe that, by removing v and $(u, p(u))$, $T(v, u]$ gets disconnected from $T(u, r]$, since $M(u)$ is in $T(v)$ and there is no child c of v with the property that $high_p(c) \geq u$ and $low(c) \leq p(u)$.

Algorithm 5.2 describes how we can count, for every vertex v, all cut-pairs of the form $\{v, (u, p(u))\}$, where u is a proper ancestor of v with $M(u) = v$.

Theorem 5.1. *Algorithm 5.2 is correct.*

Proof. By Proposition 5.2, we only have to count, for every vertex v, the vertices u in $T(v, r)$ that have $M(u) = v$ and are not contained in any set of the form $T[high_p(c), low(c))$, for any child c of v. We do this by finding, in a sense, all maximal subsets of $T(v, r)$ of the form $T[x, y]$, which do not meet any set $T[high_p(c), low(c))$, for any child c of v, and we count all elements of $M^{-1}(v) \cap T[x, y]$. If $c_1, \dots, c_k$ is the list of the children of v sorted in decreasing order w.r.t. their $high_p$ point, then the first such set is $T(v, high_p(c_1))$, the last one is $T[low(c), r)$, where c is a child of v with low minimal among the children of v, and all intermediary sets have the form $T[low(c'), high_p(c''))$, for some children c', c'' of v. If v is childless, we only have to count the elements of $M^{-1}(v) \cap T(v, r)$. □

5.3.2 The case where $M(u)$ is a proper descendant of v

In this case, $M(u)$ belongs to $T(c)$, for a child c of v, and so we have that $\{p(c), (u, p(u))\}$ is a cut-pair. We base our algorithm for this case on the following observation:

Proposition 5.3. *Let $\{p(c), (u, p(u))\}$ be a cut-pair, such that u is an ancestor of $p(c)$ and $M(u)$ is in $T(c)$. Then $M_p(c) = M(u)$ and $high_p(c) < u$. Conversely, if u is a proper ancestor of $p(c)$ such that $M_p(c) = M(u)$ and $high_p(c) < u$, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair. (See Figure 5.2.)*

Proof. ($\Rightarrow$) Let (x, y) be a back-edge such that x is in $T(u)$ and y is a proper ancestor of u. Since $M(u)$ is in $T(c)$, x is in $T(c)$. Furthermore, since u is an ancestor of $p(c)$, y is a proper ancestor of $p(c)$. This shows that $M_p(c)$ is an ancestor of $M(u)$. Conversely, let (x, y) be a back-edge such that x is in $T(c)$ and y is a proper ancestor of $p(c)$. Since c is a descendant of u, x is in $T(u)$. Furthermore, since $\{p(c), (u, p(u))\}$ is a cut-pair, y must be a proper ancestor of u. (For otherwise, we can easily see that, by removing the vertex $p(c)$ and the edge $(u, p(u))$, u remains connected with $p(u)$, since there exists a back-edge connecting a vertex from $T(M(u))$ (which is a subtree of $T(c)$) with $low(u)$, which is an ancestor of $p(u)$.) This means that x is a descendant of $M(u)$, and this shows that $M_p(c)$ is a descendant of $M(u)$. We conclude that $M_p(c) = M(u)$. Finally, since $\{p(c), (u, p(u))\}$ is a cut-pair, it should be clear that $high_p(c)$ must be a proper ancestor of u (the argument is the same as in the parenthesis).
($\Leftarrow$) Let's remove the vertex $p(c)$ and the edge $(u, p(u))$. Now, if there exists a path connecting u to $p(u)$, this path should contain at least one back-edge (x, y) such that either (1) x is in $T(c)$ and y is in $T(p(c), u]$, or (2) x is a descendant of some vertex in $T(p(c), u]$, but not a descendant of $p(c)$, and y is an ancestor of $p(u)$. (1) cannot be true, since $high_p(c) < u$. (2) cannot be true, since $M(u)$ is in $T(c)$. We conclude that u has been disconnected from $p(u)$. $\square$

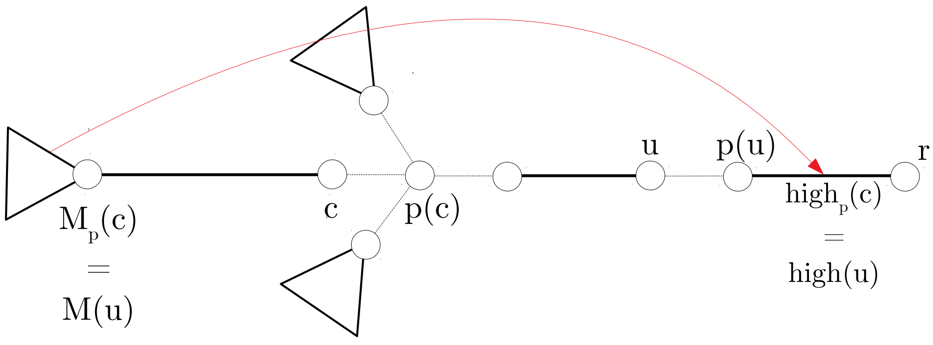


Figure 5.2: $M(u)$ is in $T(c)$ and $p(c)$ forms a vertex-edge cut-pair with $(u, p(u))$. This implies that $M(u) = M_p(c)$ and $high_p(c) < u$. In fact, in this case we have $high_p(c) = high(u)$.

Algorithm 5.3 describes how we can compute, for every vertex v, the number of cut-pairs of the form $\{v, (u, p(u))\}$, where u is a proper ancestor of v with $M(u)$ in $T(c)$ for a child c of v.

Theorem 5.2. *Algorithm 5.3 is correct.*

Proof. According to Proposition 5.3, for every cut-pair of the form $\{p(c), (u, p(u))\}$ such that u is an ancestor of $p(c)$ with $M(u)$ in $T(c)$, we have $M(u) = M_p(c)$. Therefore, we may search for these cut-pairs by scanning the lists $M^{-1}(m)$, $M_p^{-1}(m)$, for every vertex m. Suppose we have calculated these lists and have their elements sorted in decreasing order. Now, let c be the first element of $M_p^{-1}(m)$ for which there exists a u in $M^{-1}(m)$ such that u is an ancestor of $p(c)$ and the pair $\{p(c), (u, p(u))\}$ is a cut-pair. Proposition 5.3 implies that $high_p(c) < u$. Furthermore, (and, again, as a consequence of the same Proposition), we have that, for every u' in $M^{-1}(m) \cap T(p(c), high_p(c))$, $\{p(c), (u', p(u'))\}$ is a cut-pair, and these are all the elements in $M^{-1}(m)$ for which this is true. Let U denote the collection of these elements. Now, if c' is in $M_p^{-1}(m) \cap T[c, high_p(c))$, then $high_p(c') = high_p(c)$. By Proposition 5.3, this means that all the elements u' of $M^{-1}(m)$ with the property that u' is a proper ancestor of $p(c')$ and $\{p(c'), (u', p(u'))\}$ is a cut-pair, are precisely the members of $U \cap T(p(c'), high_p(c))$. This explains the counting procedure of Algorithm 5.3. Then, after we have updated $count[p(c')]$ for all c' in $M_p^{-1}(m) \cap T[c, high_p(c))$, we repeat the same process for the greatest element c' of $M_p^{-1}(m)$ which is smaller than (i.e. an ancestor of) $high_p(c)$, and has the property that there exists an element u' in $M^{-1}(m)$ such that u' is an ancestor of $p(c')$ and $\{p(c'), (u', p(u'))\}$ is a cut-pair - and keep repeating, until we have traversed $M_p^{-1}(m)$ (or $M^{-1}(m)$) entirely. □

Notice that in Sections 5.2 and 5.3.1, we were able to count specific types of cut-pairs by detecting all of them explicitly. Here, on the other hand, we count cut-pairs in an indirect manner. Of course, we were bound to perform the counting indirectly at some point: since we claim a linear-time algorithm for the computation of all *count*(v), we cannot explicitly find all vertex-edge cut-pairs, as there can be too many of those. (Consider, for instance, a cycle with n vertices; every vertex v forms precisely $n-2$ vertex-edge cut-pairs, i.e., with all the edges not incident to v.) We will perform the counting in an indirect manner again in Section 5.4.2. In the next section we basically find all cut-pairs explicitly.

5.4 The case where e lies in $T(v)$

Let $\{v, (u, p(u))\}$ be a cut-pair with u being a descendant of v. Then u is a proper descendant of a child c of v. Now, we observe that all back-edges that start from $T(u)$ and end in a proper ancestor of u must necessarily end in an ancestor of $p(c)$. In other words, $high(u) \leq v$. Here we distinguish two cases, depending on whether $high(u)$ is a proper ancestor of v.

5.4.1 The case $high(u) = v$

Our algorithm for this case is based on the following observation:

Proposition 5.4. *Let $\{v, (u, p(u))\}$ be a cut-pair such that v is a proper ancestor of u with $high(u) = v$, and let c be the child of v of which u is a descendant. Then, either (1) $low(u) = p(c)$, or (2) $low(u) < p(c)$ and $u \leq M_p(c)$. Conversely, if c is a proper ancestor of u such that $high(u) = p(c)$ and either (1) or (2) holds, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair. (See Figure 5.3.)*

Proof. (⇒) Suppose that $low(u) \neq p(c)$. Then, since $low(u) \leq high(u)$ and $high(u) = p(c)$, we have $low(u) < p(c)$. Furthermore, let e be a back-edge that starts from $T(c)$ and ends in a proper ancestor of $p(c)$. We claim that e starts from $T(u)$. For otherwise, the removal of both $p(c)$ and $(u, p(u))$ would not result in the disconnection of u from $p(u)$. Since, in this case, we could start from u, traverse the subtree $T(u)$ until we reach a vertex from which we can land with a back-edge on $low(u)$, then follow the tree path to the end of e which is a proper ancestor of $p(c)$, and, after we land on the other end of e, which is a descendant of a proper ancestor of u which is also a descendant of c, we can reach $p(u)$ through a path in $T(c)$. This shows that $M_p(c)$ is in $T(u)$, and therefore we have $u \leq M_p(c)$.

($\Leftarrow$) If (1) holds, then all back-edges that start from $T(u)$ and end in a proper ancestor of u end precisely in $p(c)$. In this case, the removal of the pair $\{p(c), (u, p(u))\}$ disconnects the vertices u and $p(u)$. If (2) holds, we claim that $M_p(c)$ is in $T(u)$. Indeed: since $low(u) < p(c)$, there exists a back-edge that starts from $T(u)$ and ends in a proper ancestor of $p(c)$. This implies that $M_p(c)$ is an ancestor of a descendant of u. But, since $u \leq M_p(c)$, $M_p(c)$ is not a proper ancestor of u. Therefore, it must be a descendant of u. Now, since $M_p(c)$ is in $T(u)$ and $high(u) = p(c)$, it is easy to see that the removal of the pair $\{p(c), (u, p(u))\}$ results in the disconnection of u from $p(u)$. $\square$

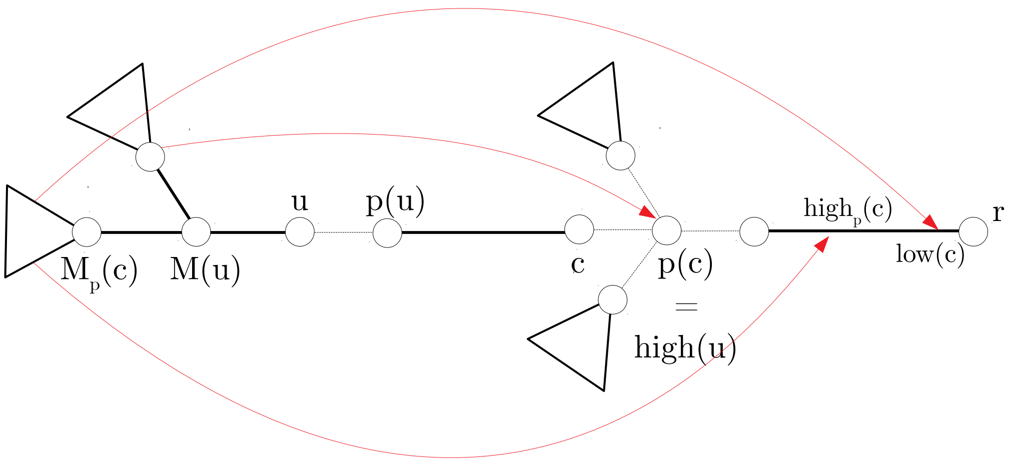


Figure 5.3: $high(u) = p(c)$ and $p(c)$ forms a vertex-edge cut-pair with $(u, p(u))$. In this case we have that $M_p(c)$ is a descendant of $M(u)$. Observe that, after removing $p(c)$ and $(u, p(u))$, the tree path $T(u, c]$ gets disconnected from the root, since all back-edges that start from $T(c)$ and end in a proper ancestor of $p(c)$ start from $T(u)$.

It is an immediate application of Proposition 5.4 that Algorithm 5.4 correctly computes, for every vertex v, the number of cut-pairs $\{v, (u, p(u))\}$ with the property that u is a descendant of v with $high(u) = v$.

Algorithm 5.4 The case $high(u) = v$

```
1: calculate all lists $high^{-1}(v)$, for all vertices $v$, and have their elements sorted in
   increasing order
2: sort the list of the children of every vertex in increasing order
3: for all vertices $v$ do
4:    $u \leftarrow$ first element of $high^{-1}(v)$
5:    $c \leftarrow$ first child of $v$
6:    while $u \neq \emptyset$ do
7:       while $c$ is not an ancestor of $u$ do
8:          $c \leftarrow$ next child of $v$
9:       end while
10:      if $low(u) = v$ or ($u \leq M_p(c)$ and $u \neq c$) then
11:         $count(v) \leftarrow count(v) + 1$
12:      end if
13:      $u \leftarrow$ next element of $high^{-1}(v)$
14:   end while
15: end for
```

5.4.2 The case $high(u) < v$

Our algorithm for this case is based on the following observation:

Proposition 5.5. *Let $\{p(c), (u, p(u))\}$ be a cut-pair such that u is a descendant of c with $high(u) < p(c)$. Then $M(u) = M_p(c)$. Conversely, if u is a proper descendant of c such that $M(u) = M_p(c)$ and $high(u) < p(c)$, then the pair $\{p(c), (u, p(u))\}$ is a cut-pair. (See Figure 5.4.)*

Proof. ($\Rightarrow$) Let (x, y) be a back-edge such that x is in $T(u)$ and y is a proper ancestor of u. Then, since u is a descendant of c, x is in $T(c)$, and, since $high(u) < p(c)$, y is a proper ancestor of $p(c)$. This shows that $M_p(c)$ is an ancestor of $M(u)$. Conversely, let e be a back-edge that starts from $T(c)$ and ends in a proper ancestor of $p(c)$. Then it is easy to see that e must start from $T(u)$ (for otherwise, since $high(u) < p(c)$, the pair

$\{p(c),(u,p(u))\}$ would not be a cut-pair, for u and $p(u)$ would still be connected with $high(u)$). Furthermore, since $p(c)$ is an ancestor of u, e ends in a proper ancestor of u. This shows that $M(u)$ is an ancestor of $M_p(c)$. Thus we conclude that $M(u) = M_p(c)$.
($\Leftarrow$) Remove the vertex $p(c)$ and the edge $(u,p(u))$. Now, if it were possible to reach $p(u)$ from u through a path in the remaining graph, such a path would have to include a back-edge that starts from $T(u)$. But such a back-edge will lead us to a proper ancestor of $p(c)$ (since $high(u) < p(c)$), and then the only way to get back to $T(c)$ (in which we must return, for this is where $p(u)$ lies) is to use a back-edge that starts from $T(c)$ and ends in a proper ancestor of $p(c)$. But such a back-edge must start from $T(u)$ (since $M_p(c)$ lies in $T(u)$). This shows that $p(u)$ cannot be reached from u. ☐

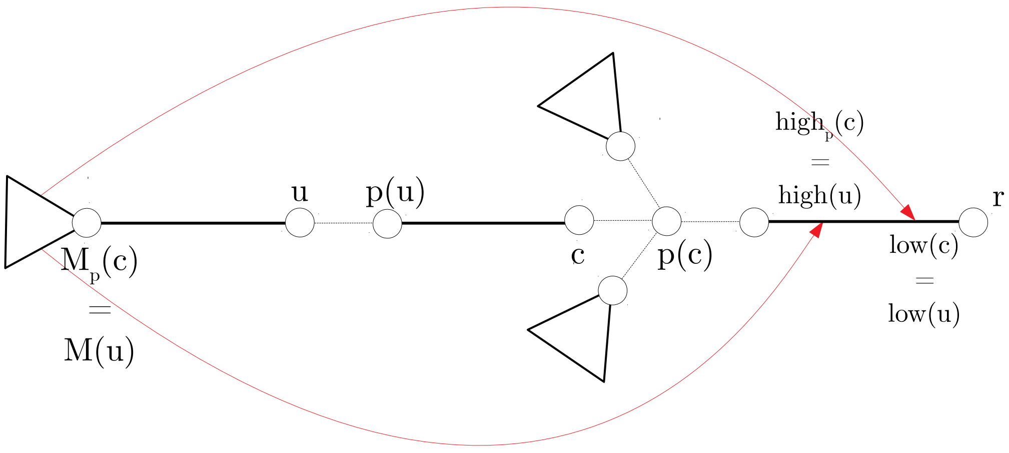


Figure 5.4: $high(u) < p(c)$ and $p(c)$ forms a vertex-edge cut-pair with $(u,p(u))$. In this case we have $M_p(c) = M(u)$.

Algorithm 5.5 describes how we can compute, for every vertex v, the number of cut-pairs of the form $\{v,(u,p(u))\}$, where u is a descendant of v with $high(u) < v$.

Theorem 5.3. *Algorithm 5.5 is correct.*

Proof. According to Proposition 5.5, for every cut-pair of the form $\{p(c),(u,p(u))\}$ with c an ancestor of u and $high(u) < p(c)$, we have $M^{-1}(u) = M_p^{-1}(c)$. Therefore, in order to count all these pairs, it is sufficient to focus our attention, for every vertex

m, on the lists $M^{-1}(m)$ and $M_p^{-1}(m)$. Now, fix a vertex m and let $U(c)$, for a vertex c in $M_p^{-1}(c)$, denote the (possibly empty) set of all u in $M^{-1}(m)$ with the property that u is a (proper) descendant of c such that $\{p(c),(u,p(u))\}$ is a cut-pair. Let c be a vertex in $M_p^{-1}(m)$ such that $U(c)$ is not empty, and let u be the greatest element in $U(c)$. Proposition 5.5 implies that $\mathit{high}(u) < p(c)$. Now let c' be a vertex in $M_p^{-1}(m)$ such that $c' \leq c$ and $\mathit{high}(u) < p(c')$. Since every u' in $M^{-1}(m) \cap T[u, \mathit{high}(u))$ has $\mathit{high}(u') = \mathit{high}(u)$, Proposition 5.5 implies that every u' in $U(c)$ is also in $U(c')$ (since such a u' is also a proper descendant of c'). Furthermore, no u' strictly greater than u can be in $U(c')$: since u is the greatest element in $M^{-1}(m)$ that is a descendant of c such that $\{p(c),(u,p(u))\}$, for every u' in $M^{-1}(m)$ strictly greater than u we must have $\mathit{high}(u') \geq c$, and therefore $\mathit{high}(u') \geq c'$. We conclude that $U(c') = U(c) \cap T[c, c')$. Therefore, if $\#U(c)$ is known, in order to find $\#U(c')$ it is sufficient to find the elements of $U(c')$ in $M^{-1}(m) \cap T[c, c')$ - call their collection C - and then $\#U(c') = \#U(c) + \#C$. This explains the counting procedure in Algorithm 5.5. Now, suppose that we have all the lists $M^{-1}(m)$ and $M_p^{-1}(m)$ computed and their elements sorted in decreasing order. Algorithm 5.5 works by finding the first u in $M^{-1}(m)$ with the property that there exists a c in $M_p^{-1}(m)$ such that c is an ancestor of u and $\{p(c),(u,p(u))\}$ is a cut-pair. Now, thanks to what we said above, we can easily calculate $\#U(c)$, for every c in $M_p^{-1}(m)$ such that $\mathit{high}(u) < p(c)$. Then, after we have properly updated all $\mathit{count}[p(c)]$, for every such c, we only have to repeat the same process for the greatest element u' in $M^{-1}(m)$ which is lower than $\mathit{high}(u)$ and such that there exists a c in $M_p^{-1}(m)$ such that c is an ancestor of u' and $\{p(c),(u',p(u'))\}$ is a cut-pair - and keep repeating, until we reach the end of $M^{-1}(m)$ (or $M_p^{-1}(m)$). □

Finally, let us briefly explain why Algorithms 5.2, 5.3, 5.4, and 5.5, run in linear time. All the required sorted lists can be computed in linear time by bucket sorting. For example, we can sort the list of children of v, for all vertices v, in increasing order w.r.t. the high_p values (as needed in Algorithm 5.2), as follows. First, we initialize all lists $\mathit{high}_p^{-1}(x)$ to $\emptyset$. Then, for every vertex c ($\neq r$ or the child of r), we insert into the list $\mathit{high}_p^{-1}(\mathit{high}_p(c))$ the element c. Now we initialize the list of children of every vertex v to $\emptyset$. We process all vertices in increasing order, and for every vertex x we do the following: we traverse the list $\mathit{high}_p^{-1}(x)$, and for every c in $\mathit{high}_p^{-1}(x)$ we insert into the list of children of $p(c)$ the element c. The computation of all $M^{-1}(m)$, $M_p^{-1}(m)$, $\mathit{high}^{-1}(x)$, etc., is performed in a similar manner. In Line 7 of Algorithm 5.4 we need to check whether c is an ancestor of u; this can be done easily in constant time (in this particular case, thanks to the way we have sorted *high*$^{-1}(v)$ and the list of the children of v, and the way the algorithm proceeds, we may simply check whether c is the last child of v, or $c \leq u$ and $c' > u$, where c' is the next child of v). Now, the key observation to see why the main part of Algorithms 5.2, 5.3, 5.4, and 5.5, runs in linear time, is that the final step in every **while** loop always moves forward to the next element of the list (and never moves backwards).

Algorithm 5.2 The case $M(u) = v$

```
1: calculate all lists $M^{-1}(v)$, for all vertices $v$, and have their elements sorted in decreasing order
2: sort the list of the children of every vertex in decreasing order w.r.t. the $high_p$ value of its elements
3: for all vertices $v$ do
4:   if $M^{-1}(v) = \emptyset$ then
5:     continue
6:   end if
7:   $u \leftarrow$second element of $M^{-1}(v)$  // the first element of $M^{-1}(v)$ is $v$
8:   $c \leftarrow$first child of $v$
9:   $min \leftarrow v$
10:  while $u \neq \emptyset$ and $c \neq \emptyset$ do
11:    $min \leftarrow high_p(c)$
12:    while $u \neq \emptyset$ and $u > min$ do
13:      $count[v] \leftarrow count[v] + 1$
14:      $u \leftarrow$next element of $M^{-1}(v)$
15:    end while
16:    $min \leftarrow low(c)$
17:    $c \leftarrow$next child of $v$
18:    while $c \neq \emptyset$ and $high_p(c) \geq min$ do
19:      if $low(c) < min$ then
20:        $min \leftarrow low(c)$
21:      end if
22:      $c \leftarrow$next child of $v$
23:    end while
24:    while $u \neq \emptyset$ and $u > min$ do
25:      $u \leftarrow$next element of $M^{-1}(v)$
26:    end while
27:  end while
28:  while $u \neq \emptyset$ do
29:    if $u \leq min$ then
30:      $count[v] \leftarrow count[v] + 1$
31:    end if
32:    $u \leftarrow$next element of $M^{-1}(v)$
33:  end while
34: end for
```

Algorithm 5.3 The case $M(u) > v$

```
1: calculate all lists $M^{-1}(m)$ and $M_p^{-1}(m)$, for all vertices $m$, and have their elements sorted in decreasing order
2: for all vertices $m$ do
3:   $c \leftarrow$ first element of $M_p^{-1}(m)$
4:   $u \leftarrow$ first element of $M^{-1}(m)$
5:   while $c \neq \emptyset$ and $u \neq \emptyset$ do
6:     while $u \neq \emptyset$ and $u \geq p(c)$ do
7:       $u \leftarrow$ next element of $M^{-1}(m)$
8:     end while
9:     if $u = \emptyset$ then
10:      break
11:    end if
12:    if $high_p(c) < u$ then
13:      $n\_edges \leftarrow 0$
14:      $first \leftarrow u$
15:      while $u \neq \emptyset$ and $high_p(c) < u$ do
16:        $n\_edges \leftarrow n\_edges + 1$
17:        $u \leftarrow$ next element of $M^{-1}(m)$
18:      end while
19:      $last \leftarrow$ predecessor of $u$ in $M^{-1}(m)$
20:      $count[p(c)] \leftarrow count[p(c)] + n\_edges$
21:      $c \leftarrow$ next element of $M_p^{-1}(m)$
22:      while $c \neq \emptyset$ and $p(c) > last$ do
23:        while $first \geq p(c)$ do
24:          $n\_edges \leftarrow n\_edges - 1$
25:          $first \leftarrow$ next element of $M^{-1}(m)$
26:        end while
27:        $count[p(c)] \leftarrow count[p(c)] + n\_edges$
28:        $c \leftarrow$ next element of $M_p^{-1}(m)$
29:      end while
30:    else
31:      $c \leftarrow$ next element of $M_p^{-1}(m)$
32:    end if
33:  end while
34: end for
```

Algorithm 5.5 The case *high*$(u) < v$

```
1: calculate all lists $M^{-1}(m)$ and $M_p^{-1}(m)$, for all vertices $m$, and have their elements
   sorted in decreasing order
2: for all vertices $m$ do
3:   $u \leftarrow$ first element of $M^{-1}(m)$
4:   $c \leftarrow$ first element of $M_p^{-1}(m)$
5:   while $u \neq \emptyset$ and $c \neq \emptyset$ do
6:     while $c \neq \emptyset$ and $c \geq u$ do
7:       $c \leftarrow$ next element of $M_p^{-1}(m)$
8:     end while
9:     if $c = \emptyset$ then
10:      break
11:    end if
12:    if $high(u) < p(c)$ then
13:      $n\_edges \leftarrow 0$
14:      $h \leftarrow high(u)$
15:      while $c \neq \emptyset$ and $h < p(c)$ do
16:        while $u \neq \emptyset$ and $c < u$ do
17:          $n\_edges \leftarrow n\_edges + 1$
18:          $u \leftarrow$ next element of $M^{-1}(m)$
19:        end while
20:        $count[p(c)] \leftarrow count[p(c)] + n\_edges$
21:        $c \leftarrow$ next element of $M_p^{-1}(m)$
22:      end while
23:    else
24:      $u \leftarrow$ next element of $M^{-1}(m)$
25:    end if
26:  end while
27: end for
```

Chapter 6

Open problems

In this Chapter we discuss some open problems whose solution would constitute a natural extension to our work.

Firstly, let us recall that in Chapter 2 we showed how to compute, in linear time, the number of TSCCs of $G \setminus e$ (resp. $G \setminus v$), for all twinless strong bridges e (resp. all twinless strong articulation points v) which are not strong bridges (resp. strong articulation points), for a twinless strongly connected digraph G. Now, one may ask whether we can have a similar result for all edges (resp. all vertices). (Of course, we are interested only in those edges (resp. vertices) that are twinless strong bridges (resp. twinless strong articulation points)). This is a reasonable question, considering that the analogous problem for strong connectivity has been solved: i.e., there is a linear-time algorithm that computes the number of strongly connected components of $G \setminus e$ (resp. $G \setminus v$), for all edges e (resp. all vertices v), where G is a strongly connected digraph (see Georgiadis et al [4]).

Another open problem concerns the 2-edge-twinless and 2-vertex-twinless blocks. A 2-edge(resp. vertex)-twinless block is a maximal set B of vertices in a twinless strongly connected digraph G with the property that, for every two vertices u, v in B, u and v remain in the same TSCC of $G \setminus e$ (resp. $G \setminus w$), for every edge e (resp. every vertex $w \notin \{u, v\}$). The concepts of 2-edge-twinless and 2-vertex-twinless blocks have been introduced by Jaberi in [8] and [9], where he provided algorithms of complexity $O((b_t - b_s + n)m)$ and $O(n^3)$, respectively, for their computation, where b_t is the number of twinless strong bridges, b_s is the number of strong bridges, n the number of vertices and m the number of edges in a twinless strongly connected digraph G. These concepts have their analogues in the context of strong connectivity (the 2-edge and 2-vertex blocks), and they can be computed in linear-time (see [4]). Jaberi left as an open question whether the 2-edge-(resp. vertex)-twinless blocks can also be computed in linear time.

Finally, although 3-connectivity in undirected and 2-connectivity in directed graphs can be tested in linear time (for 3-connectivity in undirected graphs see [13] and [6], as well as our own algorithm in Chapter 4 (for 3-edge connectivity); for 2-connectivity in digraphs see Georgiadis [3] (for 2-vertex connectivity) and [7], which computes all strong bridges and articulation points), we are not aware of any linear-time algorithms that test higher connectivities in the directed or undirected setting, and thus it is natural to ask whether such algorithms exist.

Bibliography

[1] R. Diestel. *Graph Theory*. Springer-Verlag, New York, second edition, 2000.

[2] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.

[3] L. Georgiadis. Testing 2-Vertex Connectivity and Computing Pairs of Vertex-Disjoint s-t Paths in Digraphs. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010) A*, pages 738-749.

[4] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *SODA*, pages 1880–1899, 2017.

[5] C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In Joe Marks, editor, *Graph Drawing*, pages 77–90, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[7] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. URL: `http://www.sciencedirect.com/science/article/pii/S0304397511009303`, `doi:10.1016/j.tcs.2011.11.011`.

[8] R. Jaberi. 2-edge-twinless blocks, 2019. `arXiv:1912.13347`.

[9] R. Jaberi. Computing 2-twinless blocks, 2019. `arXiv:1912.12790`.

[10] R. Jaberi. Twinless articulation points and some related problems, 2019. `arXiv:1912.11799`.

[11] S. Raghavan. Twinless strongly connected components. In F. B. Alt, M. C. Fu, and B. L. Golden, editors, *Perspectives in Operations Research: Papers in Honor of Saul Gass' 80th Birthday*, pages 285–304. Springer US, Boston, MA, 2006. doi:10.1007/978-0-387-39934-8_17.

[12] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[13] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130 – 146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP). URL: http://www.sciencedirect.com/science/article/pii/S1570866708000415, doi:https://doi.org/10.1016/j.jda.2008.04.003.

Short Biography

I hold a BSc degree (2017) in Mathematics from the Department of Mathematics of the University of Ioannina in Greece. In the last few months I became interested in algorithmic problems on graph connectivity.