GRAPH ALGORITHMS FOR NETWORK TOMOGRAPHY AND FAULT TOLERANCE

by

Abishek Gopalan

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2013
As members of the Final Examination Committee, we certify that we have read the
dissertation prepared by Abishek Gopalan
entitled Graph Algorithms for Network Tomography and Fault Tolerance
and recommend that it be accepted as fulfilling the dissertation requirement for the
Degree of Doctor of Philosophy.

Dr. Srinivasan Ramasubramanian

Date: 24 June 2013

Dr. Loukas Lazos

Date: 24 June 2013

Dr. Tamal Bose

Date: 24 June 2013

Dr. Alon Efrat

Date: 24 June 2013

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction
and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Dr. Srinivasan Ramasubramanian

Date: 24 June 2013
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Abishek Gopalan
I am deeply grateful to my dissertation advisor, Dr. Srinivasan Ramasubramanian, who has been a tremendous source of knowledge, passion and limitless energy. His willingness to discuss research and brainstorm ideas at any time have been instrumental in keeping me motivated and enthusiastic. He has also always been there to take care of my funding situation, which has helped me focus on research and for that, I will always be indebted to him. A good mentor is a crucial component in a PhD program and plays a vital role in the intellectual evolution of an individual. I am both humbled and lucky to have had him as my mentor and friend and look forward to having his insights and support for life.

I would like to thank Dr. Loukas Lazos, Dr. Alon Efrat and Dr. Marwan Krunz for imparting their knowledge on algorithm design and networking through courses I have taken under them. I also thank them for their valuable guidance and discussions throughout the course of my graduate program and for their participation during the networking seminars we have had. I am also thankful to them for having been on various exam committees of mine in the past and for providing valuable feedback.

I would like to thank Dr. Tamal Bose, Dr. Bane Vasic, Dr. Jonathan Sprinkle and Dr. Ivan Djordjevic, all of whom have been part of my various exam committees, and have provided valuable guidance through the course of my graduate program.

I would like to thank my lab mates Sangman, Ted, Sandeep, Sri Hari and Aditya for discussions on research, life and beyond and for making the lab, one’s home away from home a fun place to be and work in. I would like to thank all my friends I have made here in Tucson who are far too many to name in this single page but have made graduate school a very enjoyable and wholesome experience.

Inanimate as it may be, I am thankful to the city of Tucson for its small town charm, the night sky splendor that helped keep the amateur astronomer in me alive, for it’s mountains several of which I have enjoyed hiking, and for creating the necessity for me to learn cooking, an activity I have come to enjoy. Most importantly though, I am thankful because I will always cherish the memories of graduate school and almost all of them will be associated with this beautiful city and the friends and people I have come to know here.

Finally, but most importantly, my most profound thanks goes to my family - my mother Nirmala, my father Gopalan, my fiancee Swathi and my brother Arjun for their sacrifices and selfless love and affection. They have been a pillar of support and an unending source of emotional strength and patience without whom this dissertation would not have been possible.
DEDICATION

To my family and friends.
# TABLE OF CONTENTS

## LIST OF FIGURES

- List of figures: 9

## LIST OF TABLES

- List of tables: 13

## ABSTRACT

- Abstract: 14

## CHAPTER 1 INTRODUCTION

1.1 Organization: 24

## CHAPTER 2 NETWORK TOMOGRAPHY - PART I

2.1 Related work: 26
2.2 Problem Illustration: 28
2.3 Contributions: 30
2.4 Organization: 30
2.5 Identifying Metrics on Undirected Networks: 31
  2.5.1 Necessary and sufficient conditions: 31
  2.5.2 Constructing Three Link-Independent Trees: 35
  2.5.3 Counting Distinct Cycles: 40
  2.5.4 Algorithm complexity and correctness: 43
  2.5.5 Application to networks that are less than 3-edge connected: 45
2.6 Discussion: 46
  2.6.1 Measurements in IP networks: 46
  2.6.2 Identification Using Only Paths: 47
2.7 Identifying Metrics on Directed Links: 47

## CHAPTER 3 NETWORK TOMOGRAPHY - PART 2

3.1 Related Work: 53
3.2 Problem Illustration: 54
3.3 Contributions: 56
3.4 Organization: 57
3.5 Network Model: 58
  3.5.1 The intuition behind the solution: 59
3.6 Link Rank of a 2-Edge Connected Network: 60
  3.6.1 Analysis of an open chain: 62
  3.6.2 Analysis of the last ear: 64
  3.6.3 Putting it all together: 69
TABLE OF CONTENTS – Continued

3.6.4 Complexity Analysis ............................................. 70
3.6.5 An Example ......................................................... 71
3.7 Link Rank of a 1-Edge Connected Network ......................... 72
3.8 Discussion .......................................................... 73
3.9 Computing the linearly independent cycles ......................... 74
  3.9.1 Linearly independent cycles in the last ear .................. 74
  3.9.2 Linearly independent cycles through a type-1 component .... 76
  3.9.3 Linearly independent cycles in a type-2 component ......... 78
  3.9.4 Counting cycles across all components in the last ear: .... 81
  3.9.5 Cycles on the example network .............................. 81
3.10 Appendix ..................................................................... 83

CHAPTER 4 THREE EDGE INDEPENDENT SPANNING TREES - PRELUDE ............................. 95
  4.1 Background .................................................................. 95
  4.2 Counterexample .......................................................... 96
  4.3 Analysis ........................................................................ 97

CHAPTER 5 CONSTRUCTING THREE EDGE INDEPENDENT SPANNING TREES ............. 103
  5.1 Prior Work .................................................................... 103
  5.2 Contributions ............................................................ 104
  5.3 Organization .............................................................. 105
  5.4 Constructing Three Edge Independent Spanning Trees ........ 105
    5.4.1 Graph pruning and decomposition ............................. 106
    5.4.2 Overview of the construction procedure .................... 106
    5.4.3 Expansion to a cubic graph ................................... 107
    5.4.4 Constructing augmenting cycles/paths ...................... 110
    5.4.5 Computing segments ............................................ 112
    5.4.6 Computing red, blue, and green trees ....................... 115
    5.4.7 Correctness and complexity .................................... 116
  5.5 Appendix ...................................................................... 116
    5.5.1 A complete example for algorithm illustration ............ 120

CHAPTER 6 MULTIPATH ROUTING AND FAST RECOVERY IN IP NETWORKS ................. 124
  6.1 Organization ............................................................ 124
  6.2 Prior Work ............................................................... 125
  6.3 Network Model .......................................................... 126
  6.4 Routing with Three Link-Independent Trees ...................... 126
TABLE OF CONTENTS – Continued

6.4.1 Multipath Routing ....................................... 126
6.4.2 Fast Re-Routing ......................................... 127
6.5 Performance Evaluation ..................................... 130
  6.5.1 No Failures ........................................... 130
  6.5.2 One Link Failures ..................................... 131
  6.5.3 Two Link Failures ..................................... 132

CHAPTER 7 FAST RECOVERY IN ETHERNET NETWORKS ........................................ 137
  7.1 Fast Recovery in IP vs. Ethernet .......................... 137
  7.2 Related Work ............................................ 139
  7.3 Contributions ............................................ 141
  7.4 Organization ............................................ 142
  7.5 Fast Recovery with Multiple Spanning Trees – General Considerations 142
  7.6 3Trees Approach ......................................... 144
  7.7 2Trees Approach ......................................... 147
  7.8 1Tree Approach ......................................... 152
  7.9 Performance Evaluation ................................... 157
    7.9.1 No Failures ......................................... 159
    7.9.2 One Link Failures ................................... 160
    7.9.3 Root node selection ................................ 161
    7.9.4 Performance results ................................ 162
  7.10 Appendix ............................................... 167

CHAPTER 8 CONCLUSION ........................................ 173

REFERENCES .................................................. 176
LIST OF FIGURES

1.1 A computed tomography scan showing a cross-section of the brain (1). 15
1.2 An illustration to show that end-to-end measurements can be taken in a network to infer internal network behavior and performance. . . 16
1.3 Estimation of a non-linear metric such as bottleneck bandwidth (2) capacity (min $C_i$) by measuring the dispersion in time for a packet pair observed at the receiver $\Delta_R$. ................................. 17
1.4 An illustration of a metric that combines in an additive manner over a path. .................................................. 17
1.5 Inference from end-to-end resistance measurements on fabricated chips can help estimate individual link resistances internal to the chip. 20
1.6 An example of a power system (3). ................................. 21
1.7 Example network and three link independent spanning trees rooted at node B. ............................................. 23
2.1 Example network to illustrate the problem. ......................... 28
2.2 List of cycles (left) and the corresponding routing matrix (right). . . 29
2.3 A cutset in a two-edge-connected network involving two links $\ell_1$ and $\ell_2$. ..................................................... 32
2.4 Procedure to construct linearly independent cycles in a three edge connected network given the position of a monitoring station. . . . 33
2.5 Outline to construct three link-independent trees rooted at monitoring station $m$ in a minimally three-edge-connected graph. . . . . 36
2.6 An example of a two-vertex and minimally three-edge-connected network. ............................................. 36
2.7 Segments on the example network. ..................................... 37
2.8 Counting cycles on the example network ............................. 39
2.9 Example network and three link independent spanning trees rooted at node B. ............................................. 39
2.10 Cutset to show lack of identifiability in directed networks. ....... 48
3.1 An example network to illustrate some of the challenges in computing the link rank of a network. ............................... 55
3.2 Simplifying the problem of multiple monitors by merging the monitor nodes. ............................................. 59
3.3 A link set is transformed into a single link (of unknown metric). . . 60
3.4 An open chain consisting of \( z - 1 \) nodes and \( z \) link-sets is transformed into a single link (of unknown metric). ........................................... 61
3.5 Structure of the last ear in the 3-graph. A ear starts and ends at the same node in the 3-graph. ................................................................. 62
3.6 (a) Component \( C \) is connected to measurement node \( m \) using two path segments \( p_1 \) and \( p_2 \), whose metrics are assumed to be unknown. Component \( C \) is assumed to be three-edge-connected after the removal of segment \( p_1 - m - p_2 \). (b) As the metrics on all the links inside the component are uniquely identified, the network can be transformed to a simple two-node structure. ................................. 66
3.7 (a) Component \( C \) is 3-edge connected when the segment \( p_1 - m - p_2 \) is present and 2-edge connected when \( p_1 - m - p_2 \) is removed. The component \( C \) is decomposed into 3-edge connected sub-components (\( C_1 \) through \( C_k \)) after removing \( p_1 - m - p_2 \). (b) Transformed network. ......... 68
3.8 (a) Nodes in component \( C \) have 3-edge connectivity after removal of path segment \( p_1 - m - p_2 \). One of the disjoint paths between two nodes inside \( C \) may use \( p_3 - x - p_4 \). (b) Transformed network. ................................. 68
3.9 (a) Nodes in component \( C \) have 3-edge connectivity after removal of path segment \( p_1 - m - p_2 \). The disjoint paths between any two nodes in the component may traverse nodes segments \( p_3 - x - p_4 \) and/or \( p_5 - y - p_6 \). (b) Transformed network. ................................. 69
3.10 Steps to compute the link rank of a 2-edge connected network. ........ 70
3.11 Example network and its transformation into 3E-components, 3-graph, and last ear (open-chain transformations). ................................. 71
3.12 Steps to compute the link rank of a 1-edge connected network. ........ 73
3.13 The last ear with \( z \) link-sets ................................................................. 75
3.14 Steps to compute the linearly independent cycles in the last ear. ....... 76
3.15 (a) \( C \) is 3E-connected after removal of path segment \( p_1 - m - p_2 \). (b) Three trees constructed at a virtual node \( v \). .................................................. 77
3.16 (a) End component \( C_k \). (b) Trees being constructed in \( C_k \) .............. 78
3.17 (a) End component \( C_i \). (b) Trees being constructed in \( C_i \) .............. 79
3.18 Two common cycles across type-3 components .................................. 80
3.19 Cycles on the example network .......................................................... 82
3.20 (a) Trees in type-1 component, (b) Decomposition of the type-2 3E-component (b) Trees being constructed in \( C_2 \) ................................. 82
3.21 Identifying links in \( L_{rb} \) ................................................................. 88
3.22 Identifying metrics on attachment links \( l_1 \) and \( l_2 \) ............................. 90
3.23 Identifying links in \( L_g \) ................................................................. 90
### LIST OF FIGURES – Continued

3.24 Illustration for identifying link metrics for scenario in Figure 3.8, where $m_1 = x$ and $m_2 = y$. .................................................. 92

4.1 Example 3-edge connected graph. ........................................... 98
4.2 Transformed graph. ............................................................ 98
4.3 Vertex independent spanning tree $T'_1$. .................................. 99
4.4 Vertex independent spanning tree $T'_2$. ............................... 100
4.5 Vertex independent spanning tree $T'_3$. ............................... 101
4.6 Spanning tree $T_1$. ......................................................... 101
4.7 Spanning tree $T_2$. ......................................................... 102
4.8 Spanning tree $T_3$. ......................................................... 102

5.1 Algorithm to expand a given three-edge and two-vertex connected graph into a three-edge connected cubic graph. ......................... 108
5.2 Structure of an example graph when vertex $n$ is expanded. ....... 109
5.3 Algorithm to construct a sequence of paths in a three-connected cubic graph. ................................................................. 111
5.4 Algorithm to compute a segment in a given portion of the path. ... 113
5.5 Illustration to show segmentation of a path. ............................ 114
5.6 Procedure to construct the red and blue trees rooted at destination $d$ in $\mathcal{G}$ ............................................................. 115
5.7 Structure of the expanded graph when edge $\ell$ internal to vertex $n$ is removed. ................................................................. 118
5.8 An example two-vertex and minimally three-edge-connected graph and its expansion to a three-edge connected cubic graph. ........... 120
5.9 Cycles/Paths on the expanded graph. .................................... 120
5.10 The list of segments on the example graph. ............................ 122
5.11 The neighbors defined on the example graph .......................... 123
5.12 Example graph and three edge independent trees rooted at vertex B. 123

6.1 Example network illustrating the Red Tree First approach ......... 128
6.2 Networks considered for performance evaluation. .................... 131
6.3 The Average Path Lengths to each destination node in ARPANET under the four routing approaches ................................. 135
6.4 The Average Path Lengths to each destination node in NSFNET under the four routing approaches ................................. 135
6.5 The Average Path Lengths to each destination node in Node16 under the four routing approaches ................................. 136
6.6 The Average Path Lengths to each destination node in Mesh4x4 under the four routing approaches ................................. 136
LIST OF FIGURES – Continued

7.1 A comparison of network connectivity, number of VLANs required, and 802.1ah Mac-in-Mac encapsulation requirements for the different approaches developed. .................................................. 141
7.2 Construction of three spanning trees in an example three edge connected network. (a) Example network. (b)–(d) Three independent trees rooted at node A. (e)–(g) Three undirected spanning trees derived from the independent trees. The three undirected spanning trees have the property that for any link in the network, there exists a spanning tree that does not contain the link. ............................. 142
7.3 Forwarding procedure in 3Trees approach. ........................................ 145
7.4 Construction of two spanning trees in an example two edge connected network. (a) Example network. (b)–(c) Two undirected spanning trees derived from the independent trees, rooted at node A. (d)–(f) Three failure scenarios depicting the different backup forwarding mechanisms required. .................................................. 147
7.5 Forwarding procedure in 2Trees approach. ........................................ 149
7.6 Construction of a shortest path spanning tree in an example two edge connected network. (a) Example network. (b) Undirected shortest path spanning tree rooted at node A. ................................. 151
7.7 Construction of backup port assignments according to (4) on an example two edge connected network. (a) Tree rooted at node A. (b) Backup forwarding node assignments. (c) The red VLAN. (d) The blue VLAN. ................................................................. 153
7.8 Failure scenarios when the first vlan can be arbitrarily chosen. Two failure scenarios depicting the different backup forwarding mechanisms required on the example network. ......................... 155
7.9 Forwarding procedure in 1Tree approach. ........................................ 156
7.10 Networks considered for performance evaluation. .............................. 158
7.11 Impact of root node selection on single link failure performance of the three approaches. ................................................................. 162
7.12 Example of a directed cycle when ESCAP is employed (a) Example network. (b) Primary and backup arcs on the example network. .... 168
7.13 Illustration of $M$-ESCAP: Forcing the backup forwarding neighbor for an exit node whose backup has not yet been defined. ............... 169
7.14 Illustration of a scenario that cannot happen in ESCAP. Nodes $y$ and $z$ are both in $T(w)$. Node $z$ has no backup arc defined yet but node $y$ has its backup arc (and exit link) defined as $y \rightarrow z$. ............ 170
7.15 Illustration for the contradiction of the existence of a directed cycle in $M$-ESCAP. ................................................................. 171
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Average back up path lengths.</td>
<td>134</td>
</tr>
<tr>
<td>7.1</td>
<td>Average back up path lengths using the 1T approach.</td>
<td>164</td>
</tr>
<tr>
<td>7.2</td>
<td>Average back up path lengths using the 2T approach.</td>
<td>165</td>
</tr>
<tr>
<td>7.3</td>
<td>Average back up path lengths using the 3T approach.</td>
<td>166</td>
</tr>
</tbody>
</table>
ABSTRACT

The massive growth and proliferation of media, content, and services on the Internet are driving the need for more network capacity as well as larger networks. With increasing bandwidth and transmission speeds, even small disruptions in service can result in a significant loss of data. Thus, it is becoming increasingly important to monitor networks for their performance and to be able to handle failures effectively. Doing so is beneficial from a network design perspective as well as in being able to provide a richer experience to the users of such networks.

Network tomography refers to inference problems in large-scale networks wherein it is of interest to infer individual characteristics, such as link delays, through aggregate measurements, such as end-to-end path delays. In this dissertation, we establish a fundamental theory for a class of network tomography problems in which the link metrics of a network are modeled to be additive. We establish the necessary and sufficient conditions on the network topology, provide polynomial time graph algorithms that quantify the extent of identifiability, and algorithms to identify the unknown link metrics. We develop algorithms for all graph topologies classified on the basis of their connectivity. The solutions developed in this dissertation extend beyond networking and are applicable in areas such as nano-electronics and power systems.

We then develop graph algorithms to handle link failures effectively and to provide multipath routing capabilities in IP as well as Ethernet based networks. Our schemes guarantee recovery and are designed to pre-compute alternate next hops that can be taken upon link failures. This allows for fast re-routing as we avoid the need to wait for (control plane) re-computations.
Access to information in a timely, reliable, and secure manner is becoming increasingly critical for the information-centric lifestyle. As the network transmission speed increases, the bandwidth-delay (propagation delay) product increases, resulting in a large amount of data in transit at any given time. Therefore, any small service disruption, be it due to failures or intrusion, leads to a significant loss of data. Thus, techniques for achieving dependability (reliability, availability, security, and verifiability) must be proactive in nature. The components of a network may be constantly monitored for their performance in order to proactively re-route traffic that may be affected by a failure.

While it is desirable to monitor the links and nodes of a network, it is often impractical to make such direct measurements. A typical example is a large-scale network like the Internet where relying on internal nodes/routers to actively monitor and measure link quality is unreasonable. To minimize the co-operation required from the internal nodes, measurements are typically obtained from end-to-end paths.

Figure 1.1: A computed tomography scan showing a cross-section of the brain (1).
organs from external measurements taken from various angles without having to dissect the organs. Examples include X-rays and CT scans. A sample CT scan of a brain is shown in Figure 1.1. Similarly, it is of interest in large-scale networks to be able to infer certain individual characteristics of networks based on aggregate or end-to-end measurements. Figure 1.2 shows the equivalent setting in networks where we could have measurements taken from different vantage points in the interest of inferring internal network behavior.

![Network tomography](image)

Figure 1.2: An illustration to show that end-to-end measurements can be taken in a network to infer internal network behavior and performance.

Network tomography may be classified into passive or active tomography. In passive tomography, routers collect information on the normally forwarded traffic. Based on the collected information some network aspects, such as origin-destination traffic matrix, may be estimated. In active tomography, the network is specifically probed for information along one or more paths. Based on the path-level observations, individual link behaviors may be characterized. In this dissertation, our focus is on active tomography.

Many link-level metrics are characterized by a probability distribution (such as the queuing delay experienced by a packet, etc.). The link metrics over an end-to-end path can combine in different ways in the network. Some examples of ways in which they could combine are: (i) super-linear, e.g. bit error rates; (ii) multiplicative, e.g. reliability; (iii) additive, e.g. link delays; and (iv) concave, e.g. bottleneck bandwidth (Figure 1.3). An important objective of active network tomography is to derive the link-level probability distribution of the desired metric by observing the
behavior on a certain set of pre-established paths often referred to as the “statistical inverse problem” (6).

\[
\Delta_R = \frac{L}{\min_{i=0,\ldots,m} (C_i)}
\]

Figure 1.3: Estimation of a non-linear metric such as bottleneck bandwidth (2) capacity (min \( C_i \)) by measuring the dispersion in time for a packet pair observed at the receiver \( \Delta_R \).

Several works in network tomography attempt to solve such an inverse problem in various contexts such as: identifying additive link metrics (latency (7; 8; 9; 10) and distances (11; 12)), linearly combining optical characteristics (13), network topology (14; 15), and placement of monitors (16). For a detailed survey in the field, we refer the readers to (17; 18; 19; 20; 21).

\[
\text{Path delay} = \sum d_i
\]

Figure 1.4: An illustration of a metric that combines in an additive manner over a path.

In this dissertation, our focus is on inferring link metrics that (can be approximated to) combine in an additive manner over a path. This problem is complicated in real-life networks due to two main reasons: (1) link-level metrics are stochastic in nature; and (2) measurements are noisy. The inference of additive link metrics can be viewed as a set of linear equations as below:

\[
Y_t = AX_t + \epsilon_t
\]

(1.1)

where \( Y \) is a column vector denoting the set of observed path/cycle measurements, and \( X \) is a column vector that denotes the unknown link metrics. \( \epsilon \) is a column
vector that denotes the noise involved in the measurements. The sub-script $t$ denotes the stochastic nature or time-dependence of the measurements. The matrix $A$ is a boolean matrix, referred to as the routing matrix. Each row in $A$ denotes a measurement path or cycle established between measurement nodes indicating the links that are included in the path/cycle. The goal (inverse problem) is to compute $X_t$ given $Y_t$. The noise and statistical variance in the inferred metrics are often handled by employing least-squares approximation (11) and pseudo-likelihood estimates (19), respectively. There are several works in the literature that consider various flavors of this problem. For a detailed survey in the field, we refer the readers to (17; 18; 20), and (21). Factors such as noisy measurements, stochasticity, and an unknown network topology better model real-world scenarios in network tomography. While this may be true, even in the absence of these real-world constraints, we lack a good understanding of the basic inference problem in network tomography.

One of the fundamental problems in network tomography is to assume that link metrics are constants and additive. While this assumption simplifies real-world settings in many applications, identifying link metrics even under this assumption is not well-understood as there are no known theoretical guarantees. Consider the following model that assumes that the metrics are unknown constants and the measurements are noise-free, the set of linear equations reduce to:

$$Y = AX$$  \hspace{1cm} (1.2)

The underlying fundamental problem is to ensure that the routing matrix $A$ has full rank so that the unknown link metrics in $X$ may be computed. Even under this setting, the problem of identifying additive link metrics using end-to-end measurements is often severely under-constrained, the primary reason being the network topology. While it is known that the network topology (22) plays a role in the achievable rank of matrix $A$, its precise impact is largely unknown.

An important goal in this dissertation is to address exactly this issue. To this end, we develop the fundamental theory on the necessary and sufficient conditions on the network topology and polynomial time algorithms to compute the link metrics across
the taxonomy of graph topologies. The solutions developed in this dissertation are also applicable in scenarios where the distribution of the metric on a link is known and the parameters of the distribution need to be computed. In such cases, one may observe a path for a period of time to deduce the sum of the parameters of the individual link metric distribution along the path.

In order to be able to uniquely quantify the maximum achievable rank by the routing matrix, we need the maximum number of linearly independent linear equations that can help build the routing matrix. These equations are simply the measurements on the paths and cycles established between the measurement nodes in the network. The maximum achievable rank of the routing matrix then is the maximum number of linearly independent paths/cycles that can be constructed between the measurement nodes. We refer to this maximum achievable rank as the link rank of the network, denoted by $R_G$. The link rank is upper-bounded by the number of links in the network. The link rank is a useful metric that helps characterize and quantify the extent of identifiability or inference of the unknown link metrics on a graph topology.

Finally, it is worth noting that any deficiency in rank analyzed under the simplified model (in Equation 1.2) is largely a result of the underlying network topology and hence will be present in more sophisticated models that incorporate noise and time-variance of the inference problem as well. Thus, understanding where and why the deficiency in rank of a given network stems from will provide a fundamental insight into the structure of the problems in practical settings and thus be able to complement more sophisticated models and techniques that can exploit such information in arriving at better estimates.

Although we have talked about network tomography thus far, the problems outlined above are at their heart, graph problems from a linear algebraic perspective. In that light, the problems find applications in several other areas:

Application in nano-electronics and power systems management: The solutions developed in this dissertation are also applicable in the area of evaluating nano-electronic devices. Currently more processing units are being fabricated on a chip
that are being connected by an on-chip network, e.g. multi-core chips and FPGAs. The on-chip network is comprised of standard CMOS-based switches and metallic wires or carbon nanowires/nanotubes (in future). Fabrication at such small feature size leads to several process-variations, resulting in some links performing poorly (due to increased resistance, or break in connectivity) (23). The precise resistance values of the links may be measured by computing the pin-to-pin resistances over different paths that are linearly independent, obtained using different switching configurations inside the chip. Based on individual link characterization, the computing elements may be connected only using the “good” links (24).

Figure 1.5: Inference from end-to-end resistance measurements on fabricated chips can help estimate individual link resistances internal to the chip.

The solutions are also applicable in the area of power systems where power flow models are used to study the systems. These models are sets of equations that represent the energy flow on the transmission lines of a power grid (3). An example power grid is shown in Figure 1.6. AC power flow models while being more accurate often involve nonlinear equations and modeling can get computationally intensive for large systems. Hence, DC power flow models which are less accurate but linear, and hence much more tractable to model and analyze are used (25; 26).

The monitoring of power and voltages in such systems is critical for reliability. Thus, estimation of the state variables (bus voltages for example) which are system metrics is required and is done through the monitoring and measurement at various vantage points (using meters for example). Computing linearly independent paths/cycles in such networks and establishing meters in the right vantage points can greatly reduce the number of measurements required and can improve system
efficiency and reliability.

Thus far, we have introduced and motivated problems in network tomography - with the focus of trying to infer internal network characteristics and monitor their performance. However, failures do happen in networks, be it planned or unplanned and there is a growing need for techniques to handle such events effectively. This brings us to the other focus of this dissertation, namely designing efficient algorithms and techniques for providing fault tolerance, fast re-routing and better traffic engineering in networks.

There is a growing need for developing efficient end-to-end protocols for the Future Internet (27), specifically ones that can exploit multipath routing. One of the key technical challenges identified in (27) is:

“The outstanding technical issue with transport-based multipath is how to distinguish flows to ensure their routes diversify as soon as they enter the internetwork.”

Multipath routing (MPR) is an effective strategy to achieve robustness, load balancing, congestion reduction and low power consumption. Disjoint multipath routing provides increased security and bandwidth compared to non-disjoint multipath routing as link- or node-disjoint paths are employed. Despite the advances in multipath routing research, the use of multipath routing in IP networks is mostly
limited to equal-cost multipaths (ECMP). Recently, some sophisticated routers offer multipath routing \((28)\), however they are limited to two kinds: (i) source-based forwarding, which provides only single-path routing for a source; and (ii) forwarding port selection on a per-packet basis, which leads to high variance in the end-to-end delay, and may lead to significant throughput reduction for TCP traffic. Thus, we need an efficient mechanism to route traffic over multiple paths, ideally disjoint, in order to avoid contention for bandwidth.

On the other hand, the Internet is prone to link failures on an everyday basis \((29)\), be it due to planned maintenance or unplanned outages. As the data rates increase, the amount of data lost due to temporary service disruption increases. To ensure fast recovery from failures, the rerouting schemes must have the following characteristics: (i) proactive recovery – whereby the backup forwarding ports are calculated a priori; (ii) local recovery initiated by the node next to the failed link, rather than the source; and (iii) local recovery from a link failure without the knowledge of other failures, in case of multiple link failures.

Traditional routing in Internet Protocol (IP) networks involves computing a forwarding link for each destination, referred to as the primary (preferred) forwarding link. When a packet is received at a node, it is forwarded along the primary forwarding link corresponding to the destination address in the packet. To recover from the failure of the primary forwarding link, a node must re-route the packet over a different link, referred to as the backup forwarding link. The backup forwarding link at different nodes in the network must be chosen in a consistent manner to avoid looping. Although there have been several approaches developed for recovering from failures, many of them do not support both multipath routing and failure recovery in a combined manner.

In this dissertation, we develop a routing scheme that is capable of exploiting disjoint multipath routing while also guaranteeing recovery from arbitrary two link failures in IP networks with as little per packet overhead as possible. To this end, we develop a routing technique based on three link-independent trees. For every destination node \(d\) in the network, we construct three link-independent spanning
trees (30), referred to as red, blue, and green trees, rooted at \( d \). The property of these trees is that the path from any node \( n \) to \( d \) on the three trees are mutually link-disjoint. The three trees may be used simultaneously for disjoint multipath routing. In addition, the trees may be employed for recovering from arbitrary dual link failures by re-routing packets from one tree to another. When employed for multipath routing only, packet forwarding decision is based on the destination address and the input interface over which the packet was received. Thus, no per-packet overhead is necessary. When the trees are employed for recovering from arbitrary two link failures, at most two overhead bits are required, depending on the flexibility sought during routing. Figure 5.12 shows an example network and the trees constructed on it.

![Example network and three link independent spanning trees rooted at node B.](image)

The techniques to handle link failures in Ethernet networks are very different from those in IP networks. Also, Ethernet is becoming an attractive solution in metropolitan and wide area networks as it offers a cost-effective way to provision high data rate services (31). The simplicity and cost-effectiveness however comes with two major shortcomings of ethernet networks, poor support for traffic engineering and slow failure recovery times (32; 33). These two shortcomings are a direct consequence of employing an undirected spanning tree as the basis for forwarding in Ethernet
networks. The spanning tree plays a key role in: (i) reducing the unnecessary overhead created by broadcasting when a destination address is not available, and (ii) retaining the backward learning mechanism, which is crucial in supporting the scalability and mobility of the end-hosts. The spanning tree, however, provides only one path between any node pair and hence the failure of any link or node would disconnect the spanning tree.

To overcome the deficiencies of the spanning tree approach, there have been several revisions to the original spanning tree protocol, such as support for faster re-convergence (RSTP) (34), and support for multiple spanning trees (MSTP) (35) that can help create smaller regions for recovery. Protocols to reduce fault detection times, such as bi-directional forwarding detection (BFD) (36) were developed. Despite all these efforts, we still lack a fundamental understanding of the application of undirected spanning trees in achieving good resiliency in network design.

In this dissertation, we study the use of multiple spanning trees with interesting properties for achieving fast recovery in Ethernet networks. We develop methods to achieve fast recovery from link failures in virtual LANs (VLANs) using proactive approaches that rely only on local information with a constant overhead. Every spanning tree may be configured with a unique VLAN identifier. The VLANs are precomputed and preconfigured, thus enabling fast recovery from link failures. In addition, traffic may be split over multiple VLANs to provide increased cross-sectional bandwidth. The algorithms and protocols have provable performance guarantees.

1.1 Organization

The rest of this dissertation is organized as follows.

The second and third chapters are devoted to understand the theoretical foundations of computing linearly independent cycles in arbitrary networks and the associated problem/solution characteristics. In particular, Chapter 2 considers the problem of identifying additive link metrics on networks. It develops the necessary and sufficient conditions on network topologies in which a full rank can be achieved.
It then illustrates how to compute linearly independent paths and cycles to achieve the full rank on such networks. It also provides some insight on networks in which a full link rank cannot be achieved. Chapter 3 considers the problem of identifying link metrics and quantifying the link rank on networks that do not meet the sufficient conditions outlined in Chapter 2. Thus, in combination, the second and third chapters solve the problem of computing the link rank and identifying the unknown additive link metrics across the entire family of graph topologies.

The fourth and fifth chapters are devoted to an important tool we use throughout the dissertation, namely three edge independent spanning trees on graphs. We had briefly alluded to their applications while introducing fault tolerance in IP networks. However, it will become evident across several chapters that their properties are crucial in developing many of the results in this dissertation. Chapter 4 first introduces some background to the problem of computing three edge independent spanning trees in networks. In this chapter, we show that a prior result on independent trees turns out to be incorrect and we provide a counter-example to substantiate this claim. Besides opening up the problem of computing three edge independent spanning trees, the results in this chapter also opens up a conjecture thought to be closed almost two decades ago. Chapter 5 discusses in detail our algorithm to construct three edge independent spanning trees.

The last two chapters of the dissertation are focused on fault tolerance in networks. Chapter 6 considers the problem of guaranteeing recovery from dual link failures in IP networks using three edge independent spanning trees. We also discuss multipath routing capabilities that these trees provide. Chapter 7 considers the problem of providing fast recovery from single link failures in Ethernet networks. We develop three different routing schemes all of which guarantee fast recovery from single link failures with various parameters as trade-offs such as complexity, the graph topologies on which the solutions are applicable and the protocol support and overhead required to implement these solutions for fast recovery.

Finally, Chapter 8 concludes the dissertation.
CHAPTER 2

NETWORK TOMOGRAPHY - PART I

In this chapter, we study the problem of identifying constant additive link metrics using linearly independent monitoring cycles and paths. A monitoring cycle starts and ends at the same monitoring station while a monitoring path starts and ends at distinct monitoring stations. We show that three edge connectivity is a necessary and sufficient condition to identify link metrics using one monitoring station and employing monitoring cycles. We develop a polynomial time algorithm to compute the set of linearly independent cycles. For networks that are less than three-edge connected, we show how the minimum number of monitors required and their placement may be computed. For networks with symmetric directed links, we show the relationship between the number of monitors to be employed, the number of directed links for which metric is to be known a priori, and the identifiability for the remaining links.

2.1 Related work

The surveys in (18), (19), (20) summarize in great detail the general problem of unidentifiability of link metrics and why the measurement matrix is not invertible in most scenarios.

In (11), the authors estimate distances (time delays) of unknown paths by inferring measurements on known paths using tracer stations\(^1\). While they extract as much as possible from the measurement matrix, they do not attempt to characterize the conditions under which the matrix will have full rank. In (12), the authors exploit the fact that the shortest path has the lowest end-to-end weight and develop

\(^1\)Such techniques are useful in peer-to-peer and overlay networks, since a good choice of a server can be made when distances are known.
a consistent constraint system for inferring link weights. They then measure how well their solution approximates observed routing. In (17), the author acknowledges the problem of unidentifiable links and tries to identify the worst performing links in a sub-network. In (22), the authors try to estimate link level loss rates using multicast trees and show that there could exist unidentifiable links in the network. The works in (7), (8), (9), (10) consider an overlay system with \( n \) end hosts and develop a methodology to monitor \( k \) paths (between the end-hosts) that form a basis so that all other paths metrics may be identified using the basis. The authors show that their method achieves good approximation while bounding \( k \) as \( O(n \log n) \). However their approach cannot identify the individual link metrics in the network since they face the problem of unidentifiable links due to rank deficiency. They do not address the problem of when individual link metrics may be identified. Xia and Tse (37) consider arbitrary directed networks, where the link metrics on either direction could be different. They show that unless every link has a monitor attached on either end, none of the links can be identified. While their result holds, we show that knowing a few link metrics a priori in symmetric directed networks can greatly reduce the number of monitors required to identify all other link metrics.

While all the above works provide some insights into the problem of identifying link metrics using end-to-end paths, some fundamental questions on the identifiability remain unanswered:

(i) Under what topological conditions, are additive link metrics identifiable?

(ii) How many monitoring nodes (nodes that have the ability to make measurements) are required and where should they be placed in the network? and

(iii) Can we compute linearly independent cycles/paths in polynomial time?

This chapter provides the answers to these fundamental questions.

Related work in failure localization and compressive sensing: In the area of failure localization, several researchers have studied the problem of identifying link failures by observing failure of end-to-end paths (or cycles) (38; 39; 40; 41; 42; 43). These approaches assume that the link metrics are binary in nature and the path metric is simply a boolean OR function (assuming failure is represented by 1 and operational
links are represented as 0s). In this chapter, we seek a similar approach, except that link metrics are additive and not restricted to boolean.

The same problem of identifying additive link metrics is also being approached by using the tools provided by compressive sensing (44). The premise in the paper is that only a subset of links $k$, are likely to be congested in the network. Hence all other link metrics are approximated to zero while the congested links have some metric that they wish to identify. Our results show that we are able to achieve much better bounds even when we make no assumptions on sparsity.

### 2.2 Problem Illustration

Consider an example network as shown in Figure 2.1 with five nodes and eight links. The numbers on the links represent link IDs. Assume that node $m$ is the monitor that can start and terminate probes.

![Figure 2.1: Example network to illustrate the problem.](image)

We may compute eight cycles originating and terminating at $m$ as shown in Figure 3.14. We may represent these cycles (denoted by the links present in them) in a matrix form. Let $A = (a_{ij})$ be an $L \times L$ matrix, where $L$ denotes the number of links and an element $a_{ij}$ is a boolean entry that denotes whether link $j$ is present in cycle $i$ or not. Recall that $A$ is the routing matrix. If each cycle accumulates the link metrics on its path, then the value observed at $m$ is the sum of the link metrics. Let $Y$ denote the column matrix ($L \times 1$) of the accumulated metrics corresponding to the cycles. Let $X$ denote the column matrix ($L \times 1$) of the link variables that we are trying to identify. Our goal is to solve the system of linear equations represented by $Y = AX$. In order to uniquely determine $X$, $A$ has to be invertible. Such a
matrix is also called identifiable, since it has full rank. Cycles that make up such a matrix are referred to as *linearly independent cycles*. The eight cycles computed in Figure 3.14 are linearly independent, thus all link metrics may be identified.

\[
A = \begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}
\]

Figure 2.2: List of cycles (left) and the corresponding routing matrix (right).

The matrix \( A \) shown above has full rank, hence all link metrics can be uniquely identified by solving for \( X = A^{-1}Y \). This example illustrates that in certain topologies, all link metrics may be identified using one monitor employing only cycles. In this chapter, we study the topological constraints under which the link metrics are identifiable using one monitor employing monitoring cycles only. In addition, when the network does not satisfy the topological conditions, we identify the number of monitors required and their placement in order to identify link metrics using monitoring cycles and paths.

Note that in the above example, any arbitrary set of eight cycles may not give full rank, even though every link appears in at least one cycle\(^2\). To the best of our knowledge, there is no prior work on the computation of linearly independent cycles traversing a given node in the network. One approach to compute the set of linearly independent set of cycles is to consider one cycle at a time. Given a set of linearly independent cycles, where the cardinality of the set is less than \( L \), we need to compute another linearly independent cycle that traverses the monitor. This problem, however, is not studied in the literature. An alternate approach to construct the set of linearly independent cycles is to develop a systematic method to construct these cycles, which we employ in this chapter. We show that by constructing three

\(^2\)One such set: 124, 476, 1276, 1386, 1354, 4586, 12586, 67531 has rank six.
edge-independent trees rooted at the monitor, we may construct the desired linearly independent cycles.

2.3 Contributions

The contributions of this chapter are listed below:

1. We show that three edge connectivity is a necessary and sufficient condition for uniquely identifying additive metrics on all links using one monitoring station and employing monitoring cycles.

2. We develop a polynomial time algorithm to compute linearly independent cycles using one monitor and thus uniquely identify additive metrics on all the links.

3. Given the measurements on the linearly independent cycles and paths, we may obtain the individual link metrics in linear time, without having to compute matrix inverse.

4. For networks that do not satisfy the necessary conditions to identify metrics with one monitor, we identify the minimum number of monitors, their placement, and compute linearly independent cycles and paths for a given placement of monitors.

5. For networks with symmetric directed links, we prove unidentifiability with one monitor, show the relationship between the number of monitors employed, the number of directed links for which the metrics are known a priori, and identifiability of metrics on all other links.

2.4 Organization

The rest of this chapter is organized as follows. Section 2.5 discusses in detail the aspects of identifiability in undirected graphs by providing a detailed analysis of the necessary conditions, the underlying method required to identify all link metrics,
describes some properties of three link independent trees which are required to solve
the problem, and shows that the number of distinct cycles that are also linearly
independent equals the number of links. Section 2.6 discusses some implementation
issues and challenges. Section 2.7 studies the identifiability problem in symmetric
directed networks.

2.5 Identifying Metrics on Undirected Networks

We consider a network denoted by \( \mathcal{G}'(\mathcal{N}, \mathcal{L}') \), where \( \mathcal{N} \) denotes the set of nodes and \( \mathcal{L}' \) denotes the set of undirected links. Let \( w_\ell \) denote the unknown constant weight on link \( \ell \). We assume that the network will employ monitors at some nodes in the network. Monitoring paths that start and end at distinct monitoring stations may be established. Alternatively, monitoring cycles that start and end at the same monitor may be established. We allow the use of non-simple cycles and paths. In this dissertation, non-simple cycles (paths) are those where a node may appear more than once, however a link will not.

**Problem Statement:** Given a network \( \mathcal{G}'(\mathcal{N}, \mathcal{L}') \) with undirected links: (1) What are the necessary and sufficient conditions on the topology of the network such that the constant additive link metrics may be identified with only one monitor employing only monitoring cycles? (2) If the network topology satisfies the necessary and sufficiency conditions, develop an algorithm to construct the \(|\mathcal{L}'|\) linearly independent cycles. (3) If the network does not satisfy the necessary conditions for identifying the link metrics with only one monitor, what is the minimum number of monitors required and their placement such that all the link metrics may be identified using linearly independent cycles and paths?

2.5.1 Necessary and sufficient conditions

**Theorem 1** Three-edge connectivity is a necessary and sufficient condition for identifying all additive link metrics using one monitor employing monitoring cy-

---

3The term graph and network are used interchangeably unless otherwise specified
Proof We prove the necessary part of the theorem by contradiction. Firstly, in a one-edge-connected network, there exists a link whose removal would disconnect the graph. Thus, no cycles may be constructed through that link. Hence, the metric on that link is not identifiable. Now, assume that the given network is 2-edge connected. Then, there exists at least one cutset of size two, as shown in Figure 2.3. As the monitor is present in only one of the components, any cycle that traverses link $\ell_1$ will also traverse link $\ell_2$. Hence, the metrics on links $\ell_1$ and $\ell_2$ cannot be uniquely identified. Thus, three-edge connectivity is a necessary condition.

![Figure 2.3: A cutset in a two-edge-connected network involving two links $\ell_1$ and $\ell_2$.](image)

We prove the sufficiency part by demonstrating a method to identify the additive link metrics. Figure 2.4 shows the procedure to construct a set of linearly independent cycles in any three-edge-connected network that has a monitoring station. The procedure reduces the given network into a minimally three-edge-connected network $G$. In $G$, we construct three link-independent spanning trees rooted at the monitoring station. The existence of three independent trees, rooted at a node, are guaranteed in any three edge connected graph (30). The property of link-independent trees is that the paths from any node $n$ to the root on the trees are mutually link-disjoint. Let $T_r$, $T_b$, and $T_g$ denote the three link-independent trees in $G$ rooted at the monitor node $m$. Since the network is minimally three-edge connected, every link appears in at least one of the trees. Let $P_{ni}$ denote the path from node $n$ to the monitor on tree $T_i$, where $i = r, b, g$. For every node $n$, we compute three cycles by combining all combinations of two of the three paths from node $n$ to $m$ on the three trees. The set of all cycles computed is denoted by $C$. We first show that using the cycles in $C$, we may uniquely identify all the link metrics in $L$.

Consider the tree $T_r$. Arrange the nodes in the tree in the breadth-first manner,
Procedure: Construct Linearly Independent Cycles

Input: Graph $G'(N, L')$ and a monitoring node $m$.
Output: $C'$ – Set of linearly independent cycles traversing $m$.

1. Initialize: $C = \phi$, $C'' \leftarrow \phi$.

2. Remove edges from $G'$ to make the graph minimally three edge connected. Let $G(N, L)$ denote the minimally three edge-connected network.

3. With $m$ as root, compute three link-independent spanning trees in $G$: $T_r$, $T_b$, $T_g$. Let $P_{nr}$, $P_{nb}$, and $P_{ng}$ denote the paths from a node $n$ to $m$ on the three trees.

4. For every node $n \in N$, do:
   (a) $c_{n1} \leftarrow P_{nr} + P_{nb}$.
   (b) $c_{n2} \leftarrow P_{nb} + P_{ng}$.
   (c) $c_{n3} \leftarrow P_{nr} + P_{ng}$.
   (d) $C \leftarrow C \cup \{c_{n1}, c_{n2}, c_{n3}\}$.

5. For every link $\ell \in L' \setminus L$, do.
   (a) Consider graph $G''(N, L \cup \{\ell\})$.
   (b) Compute a cycle $c_\ell$ that traverses $\ell$ and $m$ in $G''$.
   (c) $C'' \leftarrow C'' \cup \{c_\ell\}$.

6. $C' \leftarrow C \cup C''$.

Figure 2.4: Procedure to construct linearly independent cycles in a three edge connected network given the position of a monitoring station.
based on its distance from the root—referred to as “level.” A node at level \( j \) is \( j \) hops away from the monitor. Consider a node at level \( j \), say \( n \). Let \( P_{ni} \) denote the path from node \( n \) to the monitor on tree \( T_i \), where \( i = r, b, g \). Let \( W_{ni} \) denote the sum of the link metrics on path \( P_{ni} \). If \( \alpha_{n1}, \alpha_{n2}, \) and \( \alpha_{n3} \) denote the sum of the link metrics in these cycles, respectively, we have three linear equations:

\[
\begin{align*}
W_{nr} + W_{nb} &= \alpha_{n1} \\
W_{nb} + W_{ng} &= \alpha_{n2} \\
W_{nr} + W_{ng} &= \alpha_{n3}
\end{align*}
\]

From the above equations, we may obtain the value of \( W_{nr}, \forall n \). For a node at layer 1, \( W_{nr} \) simply denotes the link metric as the node is directly connected to the monitor on \( T_r \). Thus, by considering the nodes in the breadth first manner, based on \( T_r \), the metrics on all the links in \( T_r \) may be identified successively. By repeating the same procedure for \( T_b \) and \( T_g \), all link metrics may be identified. Thus \( C \) contains sufficient number of linearly independent cycles to identify all \( |L| \) link metrics.

To compute the metrics on links that are pruned from the original graph, we consider the pruned links one at a time. For a pruned link \( \ell \), we consider the graph obtained by adding link \( \ell \) to the minimally three edge connected graph. In the resultant graph, we compute a cycle \( c_{\ell} \) traversing \( m \) and \( \ell \). The cycle \( c_{\ell} \) involves only one unknown metric, the metric on link \( \ell \), thus may be identified uniquely.

The set of linearly independent cycles is simply obtained as the set of all the cycles computed in the above steps. □

Note that we compute three cycles per node by combining two disjoint paths at a time in a minimally three edge connected network. Thus, we consider a total of \( 3(|N| − 1) \) cycles to be added to \( C \). Since we are able to compute all the link metrics from \( C \), there are at least \( |L| \) linearly independent cycles in \( C \). In fact, there are exactly \( |L| \) linearly independent cycles as there are only \( |L| \) variables in the system of equations and hence the rank cannot exceed \( |L| \). Interestingly, we can show that the number of distinct cycles in \( C \) is exactly \( L \), i.e. \( |C| = |L| \).
The counting of the number of distinct cycles obtained by merging independent paths is complicated due to the following two factors: (1) The number of occurrences of a cycle in the set of $3(|\mathcal{N}| - 1)$ cycles differs from one cycle to another; and (2) although every link is present in at least one of the trees and a link may appear in at most two trees, not all links appear in two trees. In order to show that $|\mathcal{C}| = |\mathcal{L}|$, an understanding of the structure involved in computing the three link independent trees is required.

2.5.2 Constructing Three Link-Independent Trees

In order to understand the computation of the number of distinct cycles in $\mathcal{C}$, it is necessary to understand the three tree construction procedure. In the interest of readability, we briefly discuss the algorithm to construct three independent trees here and refer the interested readers to Chapters 4 and 5 for detailed descriptions.

We consider a minimally three edge connected network, denoted by $\mathcal{G}(\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ and $\mathcal{L}$ denote the set of nodes and links, respectively. Every link $\ell \in \mathcal{L}$ is assumed to be bi-directional. In addition, we are given a monitoring station $m \in \mathcal{N}$.

Figure 2.5 shows the outline to compute the three link-independent trees rooted at $m$ in a minimally three-edge-connected graph. The first step is to divide the given network into three-edge and two-vertex connected components, referred to as 3E-2V components. Every link in the graph belongs to a unique component. Any two 3E-2V components may share at most one common node. If the decomposition results in more than one component, then we identify a “root” node, referred to as $r$, in every component that does not contain node $m$. The root node $r$ of a component is the node through which any path from a node in the component to $m$ must traverse. For the component that contains the monitor, the root node is the monitor. The second step is to consider each of the 3E-2V components and compute three link independent trees, referred to as red, blue, and green trees, that are rooted at the root node of that component. As the final step, the corresponding trees from different components are merged.

---

4Referred to as an articulation node.
1. Divide the network into two-vertex-connected components. Thus, every component is three-edge and two-vertex connected (3E-2V, for short). We identify a root node $r$ in each component. This root node is the node through which every path from a node in the component to $m$ must traverse.

2. Construct three link-independent trees in each 3E-2V component rooted at $r$.

3. Merge the trees constructed in each 3E-2V component to get the final three link-independent trees for destination node $m$.

Figure 2.5: Outline to construct three link-independent trees rooted at monitoring station $m$ in a minimally three-edge-connected graph.

![Figure 2.5](image)

Figure 2.6: An example of a two-vertex and minimally three-edge-connected network.

The computation of three trees in a 3E-2V component involves computation of a sequence of “segments” that have special properties. In the following subsections, we discuss the properties of segments which will help us in counting the number of distinct linearly independent cycles in $C$.

**Segments and their properties:** Given $G$ and a node $r$, an arbitrary neighbor of $r$, say $u$, is selected and the link $r–u$ is removed. The network is then decomposed into a sequence of $S$ segments numbered from 0 to $(S–1)$. The $i^{th}$ segment is denoted by $S_i$.

The segments have the following properties:

1. Segment $S_0$ is a cycle that starts and ends at $r$.

2. At every stage $i \geq 1$, the segment $S_i$ starts and ends at two (not necessarily distinct) nodes that are already part of earlier segments and traverses only new nodes (at least one).
3. For each $i \geq 0$, the following property holds: Consider the graph obtained by removing all the links in $\bigcup_{j=0}^{i} S_j$. The vertices and all their other links are retained. In this network, all nodes in $\mathcal{N} \setminus \bigcup_{j=0}^{\max(0,i-1)} S_j$ remain connected to node $u$.

4. $\bigcup_{j=0}^{j-S-1} S_j$ contains all vertices in $\mathcal{G}$.

Consider an example 3E-2V network shown in Figure 3.11. Figure 2.7 shows the sequence of segments computed on the example network. The monitor $r$ is at node $B$ and $u$ is node $A$ in this example. The solid lines denote the links that are part of the segments. The dotted lines indicate other links in $\mathcal{G}$ but not in the set of segments. In particular, the link $A - B$ is the link that was removed between $r - u$.

Figure 2.7: Segments on the example network.

**Computing the red and blue trees:** The red and blue trees are computed using the path augmentation approach (45; 46), where the sequence of segments are used for augmentation. In order to construct the red and blue trees, a global order (45) or partial order (46) among the links in the network is maintained to ensure the disjointedness of paths in two trees. We outline the construction of red and blue trees from the segment.

We initialize $\mathcal{T}_r$, $\mathcal{T}_b$ to be empty. Let the first cycle $S_0$ consist of the nodes $r, n_1, n_2, ..., n_k, r$. We let the red chain be $r \leftarrow n_1 \leftarrow n_2 \ldots \leftarrow n_k$ and the blue chain be $n_1 \rightarrow n_2 \ldots \rightarrow n_k \rightarrow r$. These chains are added to the respective trees. The partial order maintained is denoted by $[r, n_1] \prec [n_1, n_2] \prec \ldots \prec [n_{k-1}, n_k] \prec [n_k, r]$.

Let segment $S_i$ consist of nodes $x, n_1, n_2, ..., n_k, y$ where $x, y$ are nodes that were
added in earlier segments and nodes $n_1, n_2, \ldots, n_k$ are not present in any segment $S_j$; $j < i$. For any node $v$, let $v_R$ and $v_B$ denote the parent on the red and blue trees, respectively.  

Then, if $[x, x_R] \not\succ [y, y_R]$ in the partial order, the red chain is $x \leftarrow n_1 \leftarrow n_2 \ldots \leftarrow n_k$ and the blue chain is $n_1 \rightarrow n_2 \ldots \rightarrow n_k \rightarrow y$. The red and blue chains are added to the red and blue trees, respectively. The partial order for these edges are updated as $[x_R, x] \prec [x, n_1] \prec [n_1, n_2] \prec \ldots \prec [n_{k-1}, n_k] \prec [n_k, y] \prec [y, y_B]$.

If $[x, x_R] \succ [y, y_R]$, the chain is reversed and the partial order is $[y_R, y] \prec [y, n_k] \prec [n_{k-1}, n_{k-2}] \prec \ldots \prec [n_2, n_1] \prec [n_1, x] \prec [x, x_B]$. In this fashion, all segments are processed to compute the red and blue neighbors for all nodes.

Observe that for every segment $i$, the links $x - n_1$, $y - n_k$ are the only links that have a single color assigned on them (red or blue). All other links in $S_i$ have both the red and blue colors defined on them in opposite directions. Hence, the green edge for a node could only be on the following links:

1. A link that has only one color defined on it after $T_r$, $T_b$ are constructed; or

2. A link that is not part of the sequential ordering of segments and hence has no color defined.

The actual computation of the green neighbors is omitted here and can be found in Chapter 5 since it requires the knowledge of how the segments themselves are computed. For the purposes of counting the number of distinct cycles, it is sufficient to understand which links may be used on the green tree.

**Example:** We now illustrate how the segments are processed on the example network in Figure 5.11. When the first cycle is added, the red chain is $B \leftarrow C \leftarrow G \leftarrow F$. The partial order is $[B, C] \prec [C, G] \prec [G, F] \prec [F, B]$. For $S_1$, since $[C, G] \not\prec [F, G]$ in the order, $G$ becomes the red end of the chain. The partial order among the newly added edges is $[C, G] \prec [G, H] \prec [H, K] \prec [K, A] \prec [A, F] \prec [F, B]$. For $S_2$, since $[H, K] \not\prec [A, K]$, the red neighbor of $E$ is $K$. The partial

---

5. $x_R$, $x_B$, $y_R$, $y_B$ are already part of some $S_j$; $j < i$.
6. If $x$ and $y$ are the same node, this condition always holds.
order for the edges is $[H, K] ≺ [K, E] ≺ [E, A] ≺ [A, F]$. Finally for $S_3$, since $[G, H] ≺ [K, E]$, $H$ becomes the red end of the chain. The red, blue and green neighbors for all nodes are shown in Figure 5.11.

![Figure 2.8: Counting cycles on the example network](image)

![Figure 2.9: Example network and three link independent spanning trees rooted at node B.](image)

Observe that the sequential ordering of segments only guarantees the inclusion of all nodes but not necessarily all links in $G$. Hence, there are some links that have a color defined on only one direction while other links have colors on both edges. Further, since the network is minimally three-edge connected, all links have at least one color defined on them (part of at least one tree). The three link-independent trees obtained on the example network is re-drawn in Figure 5.12.

$\text{Because } \prec \text{ is transitive.}$
Having understood the structure behind the link independent trees, we can now enumerate the number of distinct cycles that could be obtained by merging all possible two of the three paths at all nodes, i.e. the number of distinct cycles in \( C \) in Step 4 of Fig. 2.4.

2.5.3 Counting Distinct Cycles

**Theorem 2** The number of distinct cycles obtained by combining all possible two of the three independent paths at every node in \( G \) is \(|L|\).

**Proof** For every node other than \( m \), we compute three cycles by combining two of the three independent (disjoint) paths. The total number of cycles thus obtained is \(3(|N| - 1)\). However, in such a counting, several cycles are counted more than once. In order to compute the number of distinct cycles, we analyze every segment that is augmented and count the number of distinct cycles it provides.

Every link in the network is present in at most two independent trees. Thus, every link has at most two colors assigned to it – one in each direction. Let \( G_1 \) denote the number of links in which one direction is colored green and the other edge is uncolored. Let \( G_2 \) denote the number of links in which one direction is colored green and the other direction is colored either red or blue. Since \( G_1 \) and \( G_2 \) together count all the links that are on the green tree, we have:

\[
G_1 + G_2 = |N| - 1 \tag{2.1}
\]

Consider segment \( i \). Let \( K_i \) denote the number of newly added nodes in this segment. Let the nodes in the segment be numbered \( n_0 \) through \( n_{K_i+1} \), where \( n_0 \) and \( n_{K_i+1} \) denote the two end nodes (previously added) to which this segment connects\(^8\).

Based on the above notation, we may derive a relationship between the number of links, number of nodes, and the number of segments added. Segment \( i \) with \( K_i \) new nodes has \( K_i + 1 \) links. Besides these links, there exist links that have only

\(^8\)Note that \( n_0 \) and \( n_{K_i+1} \) need not be distinct since the paths from any node to the root are only guaranteed to be link-disjoint.
a green directed edge on them in $G$. These links do not appear in the segments themselves. Thus, we have:

$$|L| = G_1 + \sum_{i=0}^{S-1} (K_i + 1)$$  \hspace{1cm} (2.2)$$

$$= G_1 + \sum_{i=0}^{S-1} K_i + \sum_{i=0}^{S-1} 1$$  \hspace{1cm} (2.3)$$

$$= G_1 + |N| - 1 + S$$  \hspace{1cm} (2.4)$$

For this segment, we count the number of distinct cycles in the set of $3K_i$ cycles, that we would obtain when the paths for the newly added nodes are merged.

We first consider the cycles obtained by merging the red and blue paths. The newly added segment will be employed for the red tree in one direction and the blue tree in the other direction for all the $K_i$ nodes. The cycles obtained by combining the red and blue paths from each of the nodes from $n_1$ through $n_{K_i}$ will be identical. Thus, the segment contributes only one distinct cycle when when the red and blue paths for any of the $K_i$ newly added nodes are combined.

We now consider the cycles formed by combining the red and green paths. Since $K_i$ new nodes are added in the segment, the cycles obtained by merging the red and green paths of nodes $n_1$ through $n_{K_i}$ will all be distinct from each other. However, the cycle obtained at node $n_1$ will be the same as that obtained for $n_0$ (counted as part of the segment in which $n_0$ was added) and thus may result in counting it twice. Let $g_i$ denote if the link $n_0-n_1$ has green label on $n_0\rightarrow n_1$: 1 if true, 0 otherwise. The number of distinct cycles obtained by merging red and green paths from the newly added nodes is $K_i - g_i$.

We now consider the cycles formed by combining the blue and green paths. By following the same line of argument as we did for the cycles obtained by combining the red and green paths, we consider the link $n_{K_i}-n_{K_i+1}$. Let $g'_i$ denote if the link $n_{K_i}-n_{K_i+1}$ has green label on $n_{K_i+1}\rightarrow n_{K_i}$ or not: 1 if true, 0 otherwise. The number of distinct cycles obtained by merging blue and green paths from the newly added nodes is $K_i - g'_i$. 
Thus, the total number of distinct cycles provided by the \(i\)th segment, denoted by \(U_i\), is computed as:

\[
U_i = 1 + K_i - g_i + K_i - g_i' \quad (2.5)
\]

\[
= 2K_i + 1 - g_i - g_i' \quad (2.6)
\]

The total number of distinct cycles obtained from all the segments, denoted by \(U\), is computed as:

\[
U = \sum_{i=0}^{S-1} (2K_i + 1 - g_i - g_i') \quad (2.7)
\]

\[
= \sum_{i=0}^{S-1} 2K_i + \sum_{i=0}^{S-1} 1 - \sum_{i=0}^{S-1} (g_i + g_i') \quad (2.8)
\]

\[
= 2(|N| - 1) + S - G_2 \quad (2.9)
\]

Subtracting (2.4) from (2.9), and using (2.1), we have:

\[
U = |L| \quad (2.10)
\]

Now, observe that these \(|L|\) distinct cycles are also linearly independent as we may employ the methodology outlined in Theorem 1 to identify the link metrics\(^9\).

\[\square\]

**Example:** We will illustrate the counting on the example network considered earlier. The three link-independent trees along with the segments augmented in \(G\) are shown in Figure 5.11. We have links C-A, D-A, B-A that appear only on the green tree, thus \(G_1 = 3\). Links G-H, F-A, K-E, H-D, E-D appear on two trees of which one of them is green, thus, \(G_2 = 5\).

The number of distinct cycles obtained from segment \(S_i\) is given by \(U_i = 2K_i + 1 - g_i - g_i'\) and computed as below for \(i = 0, 1, 2,\) and 3. The number of distinct cycles is 15, which is the total number of links in the network.

\(^9\)Note that linear independence of cycles implies distinct cycles, however the converse is not true.
\[ U_0 = 2(3) + 1 - 0 = 7 \]
\[ U_1 = 2(3) + 1 - 2 = 5 \]
\[ U_2 = 2(1) + 1 - 1 = 2 \]
\[ U_3 = 2(1) + 1 - 2 = 1 \]

**Theorem 3** The number of linearly independent cycles produced by the procedure described in Figure 2.4 is \(|\mathcal{L}'|\), the number of links in the given network.

**Proof** The number of cycles computed by the procedure in Figure 2.4 is \(|\mathcal{C}| + |\mathcal{C}''|\). By Theorem 2, \(|\mathcal{C}| = |\mathcal{L}|\). The set of cycles in \(\mathcal{C}''\) is constructed by considering one link at a time. Thus, \(|\mathcal{C}''| = |\mathcal{L}'| - |\mathcal{L}|\). Thus, the total number of cycles constructed by the procedure is \(|\mathcal{L}'|\), the number of links in the given network. \(\square\)

Note that the computation of the number of distinct cycles does not depend on the algorithm employed to construct the three independent trees. We may use the notion of segment to denote a chain of nodes \(x - v_1 - v_2... - v_k - y\) such that (a) \(x \leftarrow v_1\) is red; (b) \(v_k \rightarrow y\) is blue; (c) \(v[i - 1] \leftarrow v_i\) is red; (d) \(v[i - 1] \rightarrow v_i\) is blue. Given any three independent trees, we may compute these segments in a straightforward manner. We may use the same computation outlined above disregarding whether these segments form augmenting paths or not.

The computation above assumes that every link is present in at least one tree, which is readily satisfied if we consider a minimally three-edge connected network. However, given three independent trees rooted at a node, the network obtained by considering all the links in them need not be minimally three edge connected. Thus, given a network and three independent trees, we may simply assume that \(\mathcal{L}\) is the set of links that appear in at least one tree while \(\mathcal{L}'\) denotes the set of all links in the network.

2.5.4 Algorithm complexity and correctness

The correctness follows from the constructive proofs in Theorems 1 and 3. We now analyze the running time complexity of the algorithm in Figure 2.4. The input to the algorithm \(G'(\mathcal{N}', \mathcal{L}')\) is at least three-edge connected.
Step 2 of the procedure reduces $G'$ to a minimally three-edge connected network, which may be achieved in two steps. First, we compute a three-edge connected sparse spanning subgraph of $G'$ (47). The number of edges in the sparse graph is guaranteed to be at most $3N - 6$. Second, we reduce the sparse spanning subgraph to a minimally three edge connected graph. We consider one edge at a time and check if the edge may be removed without affecting the three edge connectivity of the spanning subgraph, which may be achieved in $O(|N|)$ time for every edge (48). As the number of edges in the sparse subgraph is $O(|N|)$, the minimally three edge connected graph is obtained in $O(|N|^2)$ time.

Step 3 of the procedure, which constructs three link-independent trees on the minimally three edge connected graph $G(N, L)$, takes $O(|N|^2)$ time (49; 30).

Step 4, which involves computing three equations (cycles) per node, may be computed in constant time per node from the three trees, thus takes $O(|N|)$ time.

Step 5 involves computing one cycle for every link in $L' \setminus L$. The cycle computation for each link takes a constant time, by making use of the partial order of the nodes that the link connects to. For example, consider link $\ell$ that connects nodes $x$ and $y$. If $y$ traverses $x$ on the red path or there does not exist an ordering between them ($[x, x_R] \not\succ [y, y_R]$), we may compute the cycle through $\ell$ by merging the red path from $x$ and blue path from $y$ with $\ell$. Otherwise ($[x, x_R] \succ [y, y_R]$), we may compute the desired cycle by merging the red path from $y$ and blue path from $x$ with $\ell$. Thus, the remaining cycles may be obtained in $O(|L'|)$ time.

Hence the overall complexity of the procedure in Figure 2.4 is $O(|N|^2)$.

Once the linearly independent cycles are obtained, we may construct a matrix of size $|L'|$ and compute its inverse to obtain individual link metrics. The complexity of inverting a matrix of size $|L'|$ is $O(|L'|^3)$. However, we may use the structure of the graph to reduce the complexity of solving for individual link metrics (as outlined in Theorem 1) to $O(|L'|)$. For every node, we solve a system of three equations in constant time, which provides the metrics along the red, blue, and green paths from the node to the monitoring station. From these path metrics, the individual link metrics are obtained by considering the links in the breadth-first order on each
tree. Therefore, computing the metrics on all the links in the minimally three-edge-connected graph takes $O(|N|)$ time. Computing the metrics on all other links (in $L' \setminus L$) takes $O(|L'|)$ time—constant time per link as the path metrics are already known. Thus, the time complexity to compute individual link metrics from cycle metrics is $O(|L'|)$.

2.5.5 Application to networks that are less than 3-edge connected

While three-edge-connectivity is a necessary and sufficient condition for identifying additive link metrics using one monitor employing only monitoring cycles, some networks may not be three-edge connected. In such scenarios, we need to employ multiple monitors. From the proof of Theorem 1, the scenarios in which we may not be able to identify linearly independent cycles become obvious.

**Lemma 1** In order to uniquely identify the metrics on all links, it is necessary and sufficient that every component obtained by removing any two links in the network must each have a monitoring station.

**Proof** See Lemma 1 of (43). Although the lemma in (43) is used in the context of distinguishing two link failures, the argument used in the lemma applies here as well.

As the placement of monitors to uniquely identify the link metrics follows the same conditions as that required for uniquely localizing single link failures, we may use the placement of monitoring stations in (43) for our problem. For sake of completeness, we outline the procedure briefly. Given a network, it is divided into three-edge-connected components. Every three-edge-connected component that has a degree\(^{10}\) of two or fewer requires a monitor inside the component. The network is then decomposed into two-edge-connected components. For every two-edge-connected component that has degree two or fewer, and that does not have a monitor already placed using the previous step, a monitor is required.

\(^{10}\)The degree of a component is the number of links that connect a node in the component to another node outside the component.
Once the monitors are placed, they are simply merged together. The resultant network is at least three-edge-connected with a single monitor. The algorithms developed in the previous sections of this chapter may be applied to obtain the cycles. Observe that some cycles in the graph where monitors are merged will become monitoring paths (from one distinct monitor to another) in the original network. Thus, given any network, we may identify the minimum number of monitors required, their placement, and the linearly independent cycles and paths to identify the additive link metrics on all undirected links. □

2.6 Discussion

In the previous sections of this chapter, we developed the necessary theoretical foundations for identifying link metrics using linearly independent cycles and paths. We now discuss how the cycles/paths may be realized in practice and some of the challenges involved for undirected networks.

2.6.1 Measurements in IP networks

Network monitoring and identifying link metrics can prove useful to a service provider. As a natural choice, we consider here how these algorithms may be implemented in an IP network. In order to enable a monitor to send packets along a cycle, we may employ IP-in-IP tunneling (RFC2003 (50)). For example, in order to measure the metric on a cycle obtained by combining the red and blue paths at node $n$, the monitoring station must send a packet on the red (blue) tree to node $n$ and node $n$ must forward that packet back to node $m$ on the blue (red) tree. To achieve this routing, we employ three undirected spanning trees—obtained from the three link independent trees rooted at $m$ and treating all the links on the tree as undirected. For each of these three spanning trees, every node maintains a routing table entry for every other node. Thus, the monitor $m$ may create a packet that is destined to itself to be routed over the blue (red) tree which is then encapsulated in another header that is destined to node $n$ to be routed on the red (blue) tree. Such
a packet may be used to measure the metric along the cycle obtained by combining the red and blue paths at node $n$.

2.6.2 Identification Using Only Paths

Thus far, we have assumed that we can use: (1) cycles only; or (2) cycles and paths. However, it is possible that we may not be able to employ cycles in some networks. For example, if an IP network does not implement IP-in-IP tunneling, then routing over a desired cycle may not be possible. In such scenarios, we may need to utilize only paths that start and end at distinct monitoring stations. Thus, we need to identify the minimum number of monitors such that all link metrics may be identified using only paths, assuming non-simple paths are allowed.

We show that if the network employs at least three monitoring stations and that the placement of monitoring stations satisfy the requirements in Lemma 1, then the link metrics may be identified with only monitoring paths. The procedure for constructing the paths is as follows: Add a virtual node $v$ and connect it to all the monitoring stations in the network. The resultant network is three edge connected because: (1) $v$ has at least three links; and (2) the placement of monitors in the original network is such that when the monitoring nodes are merged, the resultant network is three edge connected. Assuming $v$ as the only monitoring node, compute the linearly independent cycles. Every cycle in this transformed network is a path in the original network.

However, it is not clear if monitoring paths alone are sufficient if the network employs only two monitoring stations and their placement satisfies the requirements in Lemma 1.

2.7 Identifying Metrics on Directed Links

In this section, we study the problem of identifying additive link weights on symmetric directed networks. In particular, we consider the scenario when bi-directionality of links is achieved using two uni-directional links. Thus, if a link $i \rightarrow j$ exists, then
The metrics on these two directed links could be different. We consider a graph $\mathcal{G}(\mathcal{N}, \mathcal{L})$, where $\mathcal{L}$ now denotes the set of directed links. We assume that a cycle may traverse link $i \to j$ and $j \to i$.

We now show no amount of connectivity will help in identifying metrics on directed links with only one monitor.

![Figure 2.10: Cutset to show lack of identifiability in directed networks.](image)

**Theorem 4** Given a directed network, it is not possible to identify any link metric with one monitor employing cycles only.

**Proof** Consider some arbitrary cutset of the graph as shown in Figure 2.10 involving $2k$ links. Without loss of generality the monitor is in component $G_1$. Further assume that all link metrics on either side of the cutset are known. Even under this highly relaxed assumption, it is not possible to identify any link metric on the cutset. Consider all possible cycles involving (two) links in the cutset. We have a total of $k^2$ equations involving $2k$ variables. All other cycles passing through more than two links of the cutset can be written as a linear combination of these $k^2$ cycles and hence provide no additional information.

The matrix representation of these cycles (each row represents a cycle, columns $C_1$ through $C_{2k}$ represent the variables $x_1, ..., x_k, y_1, ..., y_k$ respectively) will be of the form
The first \( k \) rows represent the \( k \) cycles that pass through two links one of which is \( x_1 \). Each identity matrix is of size \( n \times n \). The \( \text{rank}(A) \leq 2k \) since the minimum of the row and column ranks dictate the maximum rank of any matrix. We show that the rank is strictly less than \( 2k \), by applying the linear transformation to columns \( C_1 \) and \( C_{k+1} \) in matrix \( A \).

\[
A = \begin{pmatrix}
1 & 0 & \cdots & 0 & I \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & 0 & \cdots & 0 & I \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & \cdots & 0 & 1 & I \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & \cdots & 0 & 1 & I \\
\end{pmatrix}
\]

Now columns, \( C_1, C_{k+1} \) are identical and hence \( \text{rank}(A) < 2k \). Observe that we have assumed that the components \( C_1 \) and \( C_2 \) provide rich connectivity. If the connectivity is sparse, then not all \( k^2 \) combinations of summations of \( x_i \) and \( y_j \) may be evaluated, hence the rank may be significantly lower than \( 2k \). □

In the proof above, the rank of the matrix is in fact exactly \( 2k - 1 \), if \( G_1 \) and \( G_2 \) components provide rich connectivity such that all \( k^2 \) summations may be computed. This is true because the metric on any one directed link say \( x_i \) helps uniquely identify all other variables. Now, observe that if a network has any cutset where all the link metrics are unknown, then none of them may be identified. Therefore, the natural follow-up is to to identify the minimum number of directed links for which the metric must be known so that the metric on all other directed links may be identified with only one monitor.

**Theorem 5** Given a symmetric directed network with \( |\mathcal{N}| \) nodes, the minimum number of directed links for which the metrics need to be known such that the metrics on other links may be identified using one monitor and monitoring cycles is \( |\mathcal{N}| - 1 \).
Proof We show the necessary part of the proof using contradiction. Assume that the metrics on $|\mathcal{N}| - 2$ directed links are known. Let $\mathcal{L}'$ denote the set of undirected links for which the metric on at least one direction is known. Then, $|\mathcal{L}| \leq |\mathcal{N}| - 2$. Now, view the network as an undirected graph with $|\mathcal{N}|$ nodes and $|\mathcal{L}'|$ links. Clearly, the set of $|\mathcal{N}|$ nodes cannot form a connected spanning network with $|\mathcal{N}| - 2$ or less links. Thus, the network must have at least two components one of which contains all the known metrics. Thus, any link traversing between these two components is part of a cutset, where all link metrics are unknown. From Theorem 4, none of the links metrics on the cutset are identifiable.

We prove the sufficiency part by construction. Consider the undirected version of the network. Compute a spanning tree, rooted at the monitor. This spanning tree contains exactly $|\mathcal{N}| - 1$ undirected edges, or equivalently $2(|\mathcal{N}| - 1)$ directed edges. For each undirected edge in the spanning tree, assume that the link metric on one direction is known, thus requiring the knowledge of exactly $|\mathcal{N}| - 1$ link metrics. We can now show that any other link metric may be identified with these known metrics. First, given that only one direction of the link metric is known, the metric on the link in the other direction may be identified by considering the links in the breadth first manner and establishing a path from the monitor traversing the link and back to the monitor. By successively computing these metrics over the spanning tree, all the $2(|\mathcal{N}| - 1)$ directed link metrics may be known. For any link $i \rightarrow j$ that was not part of the spanning tree, simply compute a cycle by merging the path from $m$ to $i$ on the spanning tree, link $i \rightarrow j$, the path from $j$ to $m$ on the spanning tree. As the metric on link $i \rightarrow j$ is the only unknown metric, it may be identified. □

While one approach to identify all link metrics is to assume that some link metrics are known a priori, another approach is to employ multiple monitors such that monitoring paths may be employed. We now generalize the above result when the number of monitors employed in the network is $K$.

Theorem 6 Given a symmetric directed network $\mathcal{G}(\mathcal{N}, \mathcal{L})$, where $\mathcal{L}$ denotes a set
of directed links, and $K$ monitors, it is necessary and sufficient to know the metrics on $|\mathcal{N}| - K - 1$ links in order to identify the metrics on all other links.

**Proof**  Merge all the nodes that are monitors. The resultant network contains $|\mathcal{N}| - K$ nodes with one monitor. Thus, by Theorem 5, we can show that it is necessary and sufficient to know the metrics on exactly $|\mathcal{N}| - K - 1$ links in order to identify the metrics on all other links. \(\square\)

**Notes on arbitrary directed networks:** Theorem 4 is applicable to arbitrary directed networks, where the existence of link $i \rightarrow j$ does not necessarily imply the existence of link $j \rightarrow i$. By assuming that there are $k_1$ links from $G_1$ to $G_2$ and $k_2$ links from $G_2$ to $G_1$, we may follow the same line of transformation as described in the proof to show that the rank of the matrix is strictly less than $k_1 + k_2$. However, the proof of Theorem 5 uses the symmetry in the directed network, thus is not applicable to arbitrary directed networks.
CHAPTER 3

NETWORK TOMOGRAPHY - PART 2

Central to network tomography is the problem of identifiability, the ability to identify internal network characteristics uniquely from end-to-end measurements. This problem is often underconstrained even when internal network characteristics such as link delays are modeled as additive constants. While it is known that the network topology can play a role in determining the extent of identifiability, there is a lack of a fundamental understanding of being able to quantify it for a given network.

In Chapter 2, we developed three key results for undirected networks: (i) In three-edge-connected networks, we showed that it is sufficient to have one measurement node to achieve a rank of $|\mathcal{L}|$. (ii) An $O(|\mathcal{V}|^2)$ time algorithm was developed to obtain the set of linearly independent cycles that achieves the rank of $|\mathcal{L}|$. (iii) For networks that are less than three-edge-connected, the minimum number of measurement nodes required to achieve full rank of the routing matrix was identified. However, we did not show how to compute the rank of the routing matrix of an arbitrary network. Since real-life networks may not often be three-edge connected, it is of interest to study the more general problem of finding the link rank in arbitrary networks. Another motivating factor is that employing more monitors to address the deficit in rank may prove both cost-prohibitive and result in lack of flexibility.

In this chapter, we consider the problem of identifying additive link metrics in an arbitrary undirected network using measurement nodes and establishing paths/cycles between them. For a given placement of measurement nodes, we define and derive the “link rank” of the network—the maximum number of linearly independent cycles/paths that may be established between the measurement nodes. We achieve this in linear time. The link rank, denoted by $R_g$ helps quantify the exact extent of identifiability in a network. We also develop a quadratic time algorithm
to compute a set of cycles/paths that achieves the maximum rank.

3.1 Related Work

Consider a network \( G(\mathcal{N}, \mathcal{M}, \mathcal{L}) \), where \( \mathcal{N} \) denotes the set of nodes, \( \mathcal{M} (\subseteq \mathcal{N}) \) denotes the set of measurement nodes, and \( \mathcal{L} \) denotes the set of links. In such a network, we have \( |\mathcal{L}| \) unknowns that we seek to identify by establishing paths/cycles between the measurement nodes.

There are two classes of networks that are of interest: directed and undirected. In arbitrary directed networks, there can be unidirectional links between nodes, i.e. \( i \rightarrow j \) exists and \( j \rightarrow i \) may not. Moreover, the metric on each directed link is independent. In such networks, it is known that the maximum achievable rank is \( |\mathcal{L}| - |\mathcal{N}| + |\mathcal{M}| \), (51; 37; 52). Observe that in order to achieve the link rank as \( |\mathcal{L}| \), all nodes must be measurement nodes. In symmetric directed networks, if link \( i \rightarrow j \) exists then link \( j \rightarrow i \) also exists, however the link metrics may be different. In such networks, we showed in (52) that it is necessary to know \( |\mathcal{N}| - |\mathcal{M}| \) link metrics and is also sufficient if their undirected representation forms a spanning graph of the network in which all the monitors are merged. Thus, the problem of computing rank deficiency is well-understood in directed networks. However, we still lack a deeper understanding of the problem in the context of undirected networks.

The undirected nature of the problem appears in several areas not just limited to network tomography. As described in the Introduction, these include power systems (3; 53) where state estimation is an important goal, sparse signal recovery in information theory and error-correction codes where compressive sensing techniques (54) are being applied, nano-electronics (24; 23) where inferring pin-to-pin resistances can be very helpful in detecting process variations in IC’s. In many of these applications, it is often impractical to set up measurements that traverse a link more than once on the modeled network. Our assumptions in this chapter also restrict a measurement path/cycle to traverse a link at most once.

Recently, there has been an interest in recovering link metrics on undirected
graphs by using rich tools and techniques provided by compressive sensing. Works in this area assume that measurements are \( k \)-sparse, meaning that there are at most \( k \) non-zero metrics that appear in any measurement path/cycle \((55; 56; 57; 58)\). In addition, most of these works also assume restricted classes of graphs, such as expander graphs \((56; 58)\) and trees \((57)\). Under these assumptions, researchers have proposed algorithms and heuristics to establish bounds on the number of measurement paths and the identifiability of the unknown link metrics. The solutions we develop in this chapter and in the previous one are far more general and can be viewed in the context of compressive sensing in the following manner. Our results show exact algorithms that explicitly outline the measurement paths/cycles required that would enable us to distinguish the links whose metrics can be identified from those whose metrics cannot be identified. Also, we are able to uniquely identify the metrics on the identifiable links on arbitrary topologies and for arbitrary sparsity.

The techniques for computing linearly independent paths (or cycles) often employ random path selections in arbitrary networks \((55; 8)\) or assume certain restricted topological structures, such as trees \((11)\). These assumptions are made even in undirected networks as evidenced in above-referenced works. For a more detailed related work section, we refer the interested reader to \((52)\), Chapter 2.

### 3.2 Problem Illustration

There are several key challenges in computing the link rank. We first observe that the number of possible cycles in a network can be far more than the number of links in the network, possibly even exponential in the number of nodes/links. This makes it computationally infeasible to extract a linearly independent set by brute-force examination. Moreover, even if one were to employ brute-force heuristic techniques, it is essential to at least know the maximum achievable rank for a given topology to evaluate their effectiveness.

Consider the example network shown in Figure 3.1 which highlights some of these issues. Node \( m \) is the monitoring node and we would like to construct monitoring
cycles that start and end at this node with the goal of identifying as many unknown link metrics as possible. We are allowed to use any cycle as long as it does not traverse the same link more than once.

Figure 3.1: An example network to illustrate some of the challenges in computing the link rank of a network.

Observe that the block of four nodes, ABCD, provides four different paths through it. These are AD, ABD, ABCD and ADBCD. Similarly, the EFG block provides two paths, EG and EFG, and the block of HIJK provides four distinct paths through it. Thus, it is easy to see that there are a total of 32 (obtained as $4 \times 2 \times 4$) different cycles that can be constructed starting from $m$. Notice that this is larger than the number of links in this network, 17. What is more interesting is that the maximum number of linearly independent cycles that can be constructed on this network is only 8. This essentially means that if we can find a set of 8 linearly independent cycles, they are sufficient to know the measurement on any other cycle that one could potentially construct through $m$ without having to actually set up the measurement cycle. It also implies that we need to know the metrics on 9 links in order to identify all 17 link metrics. However, it is not obvious which 9 link metrics would be required to solve for all the unknown link metrics. An arbitrary set would not work. Similarly, it is not necessary that we can actually compute any of the 8 unknown link metrics although the link rank is 8. So, another facet to the problem is to be able to pinpoint the links whose metrics can be inferred even though the system is rank deficient in the absence of additional information. The cardinality of this set that can be uniquely identified can range anywhere from zero to the link rank. In the example above, it is two, implying that two link metrics can be uniquely identified. In this case, they are the metrics on links BD and HJ.

Let us for the moment assume that we know how to compute the link rank. We
later show that, rather surprisingly, in spite of the obstacles outlined above, we can actually achieve the computation of the link rank in linear time. Now, we would also want to compute a set of as many cycles as the link rank that are also linearly independent. One such approach to build such a set of linearly independent cycles could be to try and incrementally add a cycle to a partial set (initially empty) of linearly independent cycles. If this additional cycle is linearly independent from the partial set thus far, then we add it to the set. Otherwise, we discard and proceed to the next cycle. In this manner, once a cycle enters the set, it stays and we can avoid expensive backtracking operations. Notice however that even if the set of cycles were manageable, checking if an additional cycle improves the rank of a partial system of linear equations is an expensive computation. To complicate this issue, the number of cycles can be large as we have seen through the example. Thus, it is computationally intensive and possibly intractable to proceed in this naive manner. We need better techniques to compute the set of cycles.

Consequently, we have the following fundamental questions: (i) What is the link rank of a given network? (ii) Can we identify a set of link metrics for which there exists a unique solution even when no link metrics are known a priori? and (iii) Can we develop an algorithm of polynomial time complexity to compute a set of cycles/paths that achieves the link rank? This chapter provides answers to these questions.

3.3 Contributions

Given an undirected network with a set of measurement nodes, the fundamental contributions of this chapter are as follows. (1) We develop an $O(|L|)$ algorithm to compute the link rank. (2) We develop an $O(|N|^2)$ algorithm to compute a set of cycles/paths that achieves the link rank. The cardinality of this set is exactly the link rank. (3) We divide the network into (type-1 and type-3) components. We divide the links in the network into two groups: (a) links that lie within a component; and (b) links that connect two components. We show that the metrics
on links that lie within a component have a unique solution even if all the links are unknown to begin with. We show that the links that connect two components cannot be identified if all the links are unknown to begin with. Thus, only links that connect two components are responsible for the rank deficiency. We show that this classification may be achieved in $O(|\mathcal{L}|)$ time. (4) The graph obtained by viewing every (type-1 and type-3) component as a node and only the links that connect two different components shows the dependency structure of the links. This transformed network allows us to evaluate what combination of link metrics need to be known such that all other or some subset of link metrics may be uniquely identified. We show that the structure of the dependencies may be computed in $O(|\mathcal{L}|)$ time. (5) Given the set of cycles from (2), the results (3) and (4), and some other link metrics to compensate the deficiency in rank, we solve for all other unknown link metrics in $O(|\mathcal{L}|)$ time.

Note that given a set of linear equations from (2), the results (3) and (4) may be obtained using matrix decomposition techniques, ignoring the graph structure. However, algorithms to solve a system of $E$ linear equations with $|\mathcal{L}|$ variables would take $O(E|\mathcal{L}|^2)$ time using matrix decomposition techniques.

3.4 Organization

The rest of the chapter is organized as follows. Section 3.5 describes the network model. Further, we map the general problem of computing the link rank when multiple measurement nodes are given to a problem of computing linearly independent cycles with one measurement node. Section 3.6 discusses the derivation of the link rank in a 2-edge connected network. Section 3.7 derives the link rank of a 1-edge connected network. Section 3.8 discusses some properties of the link rank. Section 3.9 develops the algorithm to compute linearly independent cycles. Section 3.10 is the Appendix with some properties of graph decompositions and the detailed proofs of the theorems in this chapter. It has been kept separate to improve readability.
3.5 Network Model

We consider an undirected network. We assume that the paths and cycles used may be non-simple, i.e., a path/cycle may traverse a node many times but may traverse a link only once. The assumption of undirected links implies that the measurement on a link, hence a path, is identical irrespective of the direction in which the measurement is made.

We view the network as a multi-graph, where (1) there could be multiple links between two nodes; and (2) a link could loop at a node. Even though a real-life network need not be modeled as a multi-graph, viewing it so significantly simplifies the understanding of the proofs and notations employed. As an example, observe that any node with degree two will contribute to rank deficiency. This is because the metrics of the two incident links will always appear together in any measurement that involves the node and can never be separated. For example, consider node $F$ in Figure 3.1. We can convert the two links $EF$ and $EG$ into a single link by removing $F$ while accounting for a rank deficiency of one. The resulting graph would be a multigraph. In general, we will refer to the set of links connecting two nodes as a link-set. In our case, we assume that a link-set may contain either one or two links.

1 Mapping the problem of multiple monitors: The problem of computing linearly independent cycles/paths with multiple monitors may be transformed into the problem of computing linearly independent cycles with a single measurement node by simply merging all the measurement nodes into a single node, say $m$. Observe that a measurement cycle on the transformed network is either a valid measurement path or a measurement cycle on the original network. Also, every possible measurement path (between distinct monitors) or cycle on the original network exists as a monitoring cycle in the new merged network and vice-versa. Further, since we have not added or removed links in this transformation, the link rank of the original network remains unchanged. Because of this simple yet powerful transformation, we now

\footnote{If a link-set contains three or more links, we show that the link metrics can be identified in a straight-forward manner based on our assumptions.}
focus on the problem of computing the link rank of a given network with a single
measurement node \( m \) that is capable of establishing measurement cycles for the rest
of the chapter.

![Figure 3.2: Simplifying the problem of multiple monitors by merging the monitor
nodes.](image)

As an example transformation, consider the same network as before with two
monitoring nodes as shown in Figure 3.2(a). Merging them results in the network
shown in Figure 3.2(b). One can observe the one-to-one mapping of paths and cycles
on the original network to cycles on the transformed network and vice-versa.

In the sections that follow, we first derive the link rank of a 2-edge connected
network (2E-network) with one measurement node. We identify a set of links in the
2E-network such that if the metrics on these links are known a priori, all other link
metrics can be computed at the measurement node using cycles. We then compute
the link rank of a 1-edge connected network (1E-network), where it is not possible
to employ cycles traversing some links and a measurement node. We now provide
an intuition of our solution for a 2E-network which is central to deriving the link
rank in 1E-networks as well.

### 3.5.1 The intuition behind the solution

With the results in Chapter 2 as the basis, the intuition behind the solution de-
veloped in this chapter is as follows: (1) One linear equation, by itself, provides
a rank of 1. If it involves \( k \) variables, it has a rank deficiency of \( k - 1 \). Equiva-
ently, if we consider a ring network with one measurement node or a line network
with two measurement nodes at the endpoints involving \( k \) links, we can make only
one measurement. Thus a path or a cycle involving \( k \) links, by itself, has a rank
deficiency of \( k - 1 \). (2) We showed that every link metric is identifiable in a three-
edge-connected graph. Therefore, if we decompose an arbitrary network into 3-edge-connected (3E) components, it is possible that this result is applicable to links inside the 3E-components\(^2\). (3) If all the links inside a component are uniquely identified, we may eliminate the link variables corresponding to this component from any equation, which is equivalent to contracting a component into a single node. (4) The graph obtained by decomposing a network into 3E-components and contracting each 3E-component into a node is not only minimally two-edge connected, but may also be decomposed into a unique set of cycles, where no two cycles share a link. (5) Given that every link can be classified as belonging to a 3E-component or to a cycle, and that we understand the characteristics of a 3E-component and a cycle, we can use these facts to compute the link rank of the network.

3.6 Link Rank of a 2-Edge Connected Network

**Step 1:** We first consider a cycle that traverses nodes \(m, n_0\) and \(n_z\), as shown in Figure 3.3. We show that we may replace the link-set with a link (of unknown metric) without affecting the rank.

![Figure 3.3: A link set is transformed into a single link (of unknown metric).](image)

We then consider an open-chain consisting of \(z\) link-sets, as shown in the top part of Figure 3.4. We analyze the rank deficiency of this open chain. We then transform the open chain of \(z\) link-sets into a single link with an unknown additive link metric, as shown in the bottom of Figure 3.4. This transformation results in reducing the size of the network.

\(^2\)As we show later in the chapter, the result holds in 3E-components with some additional constraints. Here, we are simply trying to provide an intuitive feel for why this line of solution would even be attempted by someone.
Figure 3.4: An open chain consisting of \( z - 1 \) nodes and \( z \) link-sets is transformed into a single link (of unknown metric).

**Step 2:** We decompose the given 2E-network into 3E-components\(^3\). We consider the super-graph (referred to as the 3-graph) \( G_3 \), where every 3E-component is viewed as a node. The 3-graph has a special structure, there exists exactly two link-disjoint paths between any two nodes\(^4\). The measurement node in the 3-graph is assumed to be the one corresponding to the 3E-component that contained the measurement node in the original network. With the measurement node as the root in the 3-graph, we compute the ear decomposition - where the graph is divided into a sequence of cycles (closed ears). We show that the structure of the last ear, as shown in Figure 3.5 is the same as the open chain considered earlier. We analyze the rank deficiency of the last ear and replace it with a link (of unknown metric), which will be consumed inside one of the remaining nodes in the 3-graph. We recursively analyze the ears (selecting one at a time from the tail of the ear sequence), analyzing the rank deficiency, accumulating the rank deficiency, and replacing the ear with a link. We perform this recursive operation until we are left with only one node in the 3-graph (which will be the measurement node in the 3-graph). If the accumulated deficiency is denoted by \( d \), then the link rank of the 2E-network \( G \) is given by \( R_G = |L| - d \).

\(^3\)A 3E-component is one in which any two nodes inside it will have three link-disjoint paths between them.

\(^4\)Note that this property is much stronger than minimally 2-edge connected property. In a minimally 2-edge connected network, we may have some node pairs that have more than two link-disjoint paths between them.
Figure 3.5: Structure of the last ear in the 3-graph. A ear starts and ends at the same node in the 3-graph.

3.6.1 Analysis of an open chain

Consider a cycle that traverses nodes $m$, $n_0$, and $n_z$ as shown in Figure 3.3. Let $\Psi$ denote the link-set connecting $n_0$ and $n_z$. Let $P_0$ denote the path from $m$ to $n_0$ and $P_z$ denote the path from $n_z$ to $m$. Assume that $P_0$ and $P_z$ are link-disjoint so that cycles can be established traversing the links in the link-set.

Assume that all the link metrics are unknown. We can establish at most two cycles traversing $m$, such that these cycles differ only in one link, where the differing link belongs to the link-set. Let $y_1$ and $y_2$ denote the measurements on these cycles, $x_1$ and $x_2$ denote the metrics on the links in $\Psi$, and $w_0$ denote the sum of the metrics on the paths $P_0$ and $P_z$. The two cycles result in two linear equations involving three variables and is of the form:

$$y_i = w_0 + x_i, \text{ where } i = 1, 2. \quad (3.1)$$

It is fairly straight-forward to see that the rank of this system of linear equations is 2. Thus, if we either know $w_0$ or $x_i$ for some $i$, we may compute the other metrics. If we assume that we do not have the knowledge of $w_0$, then we may simply replace the link-set connecting $n_0$ to $n_z$ with just one link (with an unknown metric). This link could represent any link in the link-set. If the resultant structure is such that this unknown link metric is identifiable by establishing cycles (assuming the network topology allows), then we may compute all the link metrics in the link-set. If the resultant structure does not permit identification of this link metric, then none of the link metrics in the link-set is known. Thus, from the link identifiability point of
view, connecting \( n_0 \) and \( n_z \) with a link-set of two links (of unknown metrics) is the same as connecting the two nodes with a single link (of unknown metric). Thus, we transform a link-set with unknown metrics into a single link with an unknown metric.

Note that if the link set has three or more links, then it is possible to identify all link metrics without any metrics on links outside the link-set. Consider the same Figure 3.3 and assume that we have three links. We use the same notations as before and all link metrics are assumed to be unknown. We can now establish at least three cycles traversing \( m \), such that these cycles differ only in one link, where the differing link belongs to the link-set. The three cycles involving four variables are of the form (as before):

\[
y_i = w_0 + x_i, \text{ where } i = 1, 2, 3. \quad (3.2)
\]

As we allow for non-simple cycles to be established, we may construct a fourth cycle as:

\[
y_4 = w_0 + x_1 + x_2 + x_3. \quad (3.3)
\]

The rank of the above system of linear equations is 4, Hence all variables can be solved for. Thus, the link-set is not rank deficient and \( n_0, n_z \) can be merged to a single node. The merging of nodes indicate that there exists a path from one node to another with known path path metric.

**Identifying metrics on a chain of link-sets:** Now, consider a cycle that traverses through an open chain consisting of \( z \) link-sets, as shown in Figure 3.4. The nodes in the open-chain (\( n_1 \) through \( n_{z-1} \)) do not have direct links connecting to any other nodes in the network.

Consider the link-set connecting nodes \( n_0 \) and \( n_1 \), say \( \Psi \). Assume the same notations employed earlier for the metrics on the links, paths, and cycles. Let \( w_{1z} \) denote the metric on the path segment from \( n_1 \) to \( n_z \). The two cycles result in two linear equations involving four variables and is of the form:

\[
y_i = w_0 + w_{1z} + x_i, \text{ where } i = 1, 2. \quad (3.4)
\]
Again, the rank of the above system is 2. If we assume that we have no knowledge of $w_1$ and $w_0$, then in order to identify the link metrics in $\Psi$, we need to know one link metric in the link-set. Thus, the link-set results in a deficiency of 1 in the rank. This deficiency in rank is overcome by the a priori knowledge of one of the link metrics in $\Psi$. That is, if a link metric in $\Psi$ is known, we may use the above equations to compute the other link metric in $\Psi$. Now, since the metric on any path from $n_0$ to $n_1$ (traversing either link in $\Psi$) is known, we may simply collapse nodes $n_0$ and $n_1$ into a single node. Thus, the resultant structure is a chain of $z - 1$ link-sets. We may apply this approach recursively until we are left with one link-set connecting $n_0$ and $n_z$. We may then use the earlier argument to transform a link-set into a link (of unknown metric) without adding to the rank deficiency. Irrespective of whether this link metric is identifiable or not with the rest of the network structure, the deficiency of $z - 1$ contributed by the chain does not change.

**Note:** The final link with an unknown metric that we employ to connect nodes $n_0$ and $n_z$ can be viewed as a logical link that has two different interpretations. First, the logical link represents some path connecting $n_0$ and $n_z$ through the open chain. Thus, if the network structure allows identification of the metric on the logical link (a path), a metric on one of the paths from $n_0$ to $n_z$ is known. The knowledge of this metric is what helps in having one link-set where none of the link metrics are known a priori (the deficiency of $z - 1$ and not $z$). Second, if the metric on the logical link is not identifiable using the remaining network structure, then in that scenario the logical link corresponds to some link in a link-set where none of the metrics were assumed to be known. Thus, if we need to have an a priori knowledge of the logical link metric, then it implies that one of the link metrics (on a link-set) in the chain corresponding to the logical link needs to be known.

### 3.6.2 Analysis of the last ear

We consider the last ear obtained in the ear-decomposition of the 3-graph, which has a structure as shown in Figure 3.5; a chain of 3E-components in the original network (that are nodes in the 3-graph), where adjacent components are connected.
by a single link. Since this is the last ear, the 3E-components in the chain do not have any direct links to any other 3E-components.

Consider a 3E-component in this chain, say $C$. Any two nodes inside this component have three link-disjoint paths between them. Since the 3E-component has only two outgoing links, at most one disjoint path can depend on the nodes and links outside this component. Thus, every 3E component in the last ear must be at least 2E-connected within itself (deleting the outgoing links from the component). It is possible that the 3E-component remains 3E-connected even after the removal of external edges (3E-connected internally). Thus, we now consider two cases:

- Case 1: A 3E-component remains 3E-connected after the removal of its outgoing links; *Type-1 component*.

- Case 2: A 3E-component is not 3E-connected, but 2E-connected after the removal of its outgoing links; *Type-2 component*.

We now show the transformation of the 3E-components under these two cases to a chain of nodes connected by link-sets.

**Case 1:** The cycles established from the measurement node through the 3E-component $C$ must enter the component through one link and exit through the other link. It is possible that these two links can be incident on the same node inside the component (as we consider only edge-connectivity). In that case, we may simply assume that the 3E-component as a 3E-network and the node that is connected to the outgoing links of the component as the measurement node, say $m'$. With this assumption, we may use the result in (52) to compute linearly independent cycles traversing $m'$. All these cycles will have to combine with an external cycle\(^5\) traversing $m$ and $m'$. Thus, all links inside the component are identifiable.

Now, consider the scenario where the two outgoing links of the component are incident at two distinct nodes, say $m_1, m_2 \in C$. In this case, any measurement path from $m$ traversing this component must enter through $m_1$ and exit through $m_2$.

\(^5\)Although we may not know the individual link metrics on the external cycle, the effect of the entire external cycle can easily be computed by avoiding all nodes in $C$. 
Thus, we need to identify linearly independent paths through the component $C$ that start at $m_1$ and end at $m_2$. The following results guarantee the presence of the required linearly independent cycles through type-1 components.

**Result 1:** Given a 3E-network with two distinct measurement nodes $m_1$ and $m_2$, we may identify all link metrics using linearly independent paths, where every path starts at $m_1$ and ends at $m_2$. Note that the paths may be non-simple. For a formal proof, refer to Theorem 7, Appendix-B.

Note that any cycle from the measurement node traversing this component will have an additional unknown metric. The path metric from $m$ to $m_1$ and the path metric from $m_2$ to $m$. The following result ensures that these external unknowns do not hamper the identification of the links inside a type-1 component.

**Result 2:** Consider a component $C$ that is connected to a measurement node through two path segments $p_1$ and $p_2$, as shown in Figure 3.6. Let us assume that the metrics on links $p_1$ and $p_2$ are unknown. In such a scenario, we may identify the metrics on all links inside the component $C$ using linearly independent cycles traversing $m$ if the component $C$ is 3E-connected after the removal of path segment $p_1-m-p_2$. See Theorem 8, Appendix-B.

![Figure 3.6](image)

Figure 3.6: (a) Component $C$ is connected to measurement node $m$ using two path segments $p_1$ and $p_2$, whose metrics are assumed to be unknown. Component $C$ is assumed to be three-edge-connected after the removal of segment $p_1-m-p_2$. (b) As the metrics on all the links inside the component are uniquely identified, the network can be transformed to a simple two-node structure.

As the metric of every link in the component is uniquely identified, we may simply collapse the component $C$ into a single node and assume that two path segments connect the two nodes. We refer to the transformed node arising from a type-1 component as a *type-1* node. Also of note is that these path segments may contain
just one link in each, hence viewing the network as a multi-graph helps.

**Case 2:** Consider the network structure shown in Figure 3.6. Let us assume that the component $C$ is 3E-connected in the presence of path segment $p_1 - m - p_2$ and is not 3E-connected when the path segment $p_1 - m - p_2$ is removed. As the path segment can be part of only one disjoint path, the component $C$ has to be 2E-connected after the removal of the path segment. In such a scenario, not every link inside $C$ can be uniquely identified. The number of links that can be uniquely identified can be computed by observing the structure of component $C$ after the removal of the path segment $p_1 - m - p_2$.

After removing $p_1 - m - p_2$, we further decompose $C$ into 3E-components. As the component with the path segment is 3E-connected and the path segment provides at most one disjoint path between any two node pairs in the component, the decomposition of $C$ results in a simple structure as shown in Figure 3.7. See Lemma 3 in Appendix-A. The decomposition results in a chain of sub-components $C_1$ through $C_k$, where any two successive sub-components will have exactly two links between them. In addition, the path segments $p_1$ and $p_2$ will be incident on the two end sub-components. Each such sub-component that has resulted from a decomposition of a type-2 component further into 3E-components is referred to as a **Type-3 component**.

Any link $\ell \in C$ may either be inside one of the sub-components or may connect two successive sub-components.

**Case 2a:** If $\ell$ is inside one of the sub-components, we can uniquely identify the link metric even if no other link metrics are known. Let us assume the link is inside sub-component $C_1$. In this case, we may collapse sub-components $C_2$ though $C_k$ into a single node $x$. The resultant structure is similar to that in Figure 3.8 and we use Result 3 (see below). The same argument holds if the link is inside sub-component $C_k$ as we may collapse sub-components $C_1$ through $C_{k-1}$ to $x$ as shown in Figure 3.8 and apply Result 3. Assume that the link falls within a sub-component $C_i$, where $i = 2, \ldots, k - 1$. We may collapse sub-components $C_1$ through $C_{i-1}$ as node $x$ and $C_{i+1}$ through $C_k$ as node $y$ and apply Result 4 (see below). Thus, the metrics of all
Figure 3.7: (a) Component $C$ is 3-edge connected when the segment $p_1-m-p_2$ is present and 2-edge connected when $p_1-m-p_2$ is removed. The component $C$ is decomposed into 3-edge connected sub-components ($C_1$ through $C_k$) after removing $p_1-m-p_2$. (b) Transformed network.

Result 3: Consider a component $C$ that is connected to measurement node $m$ as shown in Figure 3.8. Assume that metrics of path segments $p_1$ through $p_4$ are unknown. In such a scenario, we may identify the metrics on all links inside the component $C$ using linearly independent cycles traversing $m$ if the component $C$ is 3E-connected after the removal of path segment $p_1$ (or $p_2$). Note that, in this scenario, it is possible that one of the three link-disjoint paths between any two nodes in $C$ may traverse path segment $p_3-x-p_4$. As the metrics on all the links inside $C$ are identified, the network can be transformed as shown in Figure 3.8(b). See Theorem 9, Appendix-B.

Figure 3.8: (a) Nodes in component $C$ have 3-edge connectivity after removal of path segment $p_1-m-p_2$. One of the disjoint paths between two nodes inside $C$ may use $p_3-x-p_4$. (b) Transformed network.
**Result 4:** Consider a component $C$ that is connected to measurement node $m$ as shown in Figure 3.9. Assume that metrics of path segments $p_1$ through $p_6$ are unknown. In such a scenario, we may identify the metrics on all links inside component $C$ using linearly independent cycles traversing $m$ if component $C$ is $3E$-connected after the removal of path segment $p_1$ (or $p_2$). In this scenario, the three link-disjoint paths between any two nodes in $C$ may traverse path segment $p_3-x-p_4$ and/or $p_5-y-p_6$. See Theorem 10, Appendix-B.

![Figure 3.9](image)

Figure 3.9: (a) Nodes in component $C$ have 3-edge connectivity after removal of path segment $p_1-m-p_2$. The disjoint paths between any two nodes in the component may traverse nodes segments $p_3-x-p_4$ and/or $p_5-y-p_6$. (b) Transformed network.

**Case 2b:** We now consider the scenario that $\ell$ is a link that connects two sub-components. In this case, the identifiability of the link metric will depend on whether the other link connecting the same two sub-components is known or not.

The nodes that result from collapsing type-3 components are referred to as type-3 nodes. Observe the structural similarities of the transformed networks in Figures 3.6(b) and 3.7(b).

It is now fairly straight-forward to observe that if we consider a chain of $3E$-components of the original graph(chain of nodes in the 3-graph), all of the above results transform it into an open-chain of nodes connected by link-sets, where each link-set contains one or two links.

3.6.3 Putting it all together

Figure 3.10 shows the steps involved in computing the link rank of a 2E-network.
Procedure Compute Link Rank in 2E Network:
Input: $G(N, L)$ - A 2E-network $G$, where $N$ and $L$ denote the set of nodes and links, respectively.
Output: $R_G$ - Link rank.

1. Initialize $G' \leftarrow G$; $d \leftarrow 0$.
2. Obtain the 3-graph of $G'$ namely $G'_3$.
3. Compute the ear decomposition of $G'_3$ with the measurement node as the root.
4. If $G'_3$ contains only one node, go to step 4.
5. Transform the last ear into a chain of link-sets, say $z$.
6. $d \leftarrow d + z - 1$. // Increase deficiency in link rank by $z - 1$.
7. Replace the last ear with a link.
8. Go to Step 4.
9. $R_G \leftarrow |L| - d$. // Compute link rank.
10. Stop.

Figure 3.10: Steps to compute the link rank of a 2-edge connected network.

3.6.4 Complexity Analysis

The link rank of a 2E-network may be computed in linear time as detailed below. We analyze the complexity of the procedure in computing the link rank shown in Figure 3.10 one step at a time.

Step 2) $G_3$ can be computed from $G$ in $O(|L|)$ time (59).

Step 3) The ear decomposition of $G_3$ can be computed in $O(|L|)$ time (60).

Steps 4) through 8) consist of a recursion that goes as deep as the number of ears in $G_3$. At any stage, the number of logical links present in the ear under consideration is at most one. Transforming any ear(+ 1 logical link) to an open chain involves the computation of 3E-components of type-1 and/or type-2 nodes in the ear. This takes time proportional to the number of links in the ear including the ones inside the super-nodes of the ear under consideration. Hence, the total time taken over all
Figure 3.11: Example network and its transformation into 3E-components, 3-graph, and last ear (open-chain transformations).

The ears\(^6\) in \(G_3\) is \(O(|\mathcal{L}| + |N_3|)\) which is \(O(|\mathcal{L}|)\).

Therefore, the link rank can be computed in time linear in the number of links in the given 2E-network.

3.6.5 An Example

Figure 3.11 illustrates the concepts discussed above using an example network. Figure 3.11(a) shows the example two edge connected network and its decomposition into 3E-components (shaded squares). Figure 3.11(b) shows the corresponding 3-graph. The node names in the 3-graph are simply the combination of the names of the nodes inside that component. The 3-graph is decomposed into two ears: \(m\text{–}AB\text{–}m\) and \(AB\text{–}CDEF\text{–}GHIJ\text{–}AB\). Note that the component CDEF remains 3E-connected even when links AC and EG are removed. On the other hand, component GHIJ remains only 2E-connected when links GE and IB are removed. Thus, the transformation of the last ear to an open chain yields \(AB\text{–}CDEF\text{–}G\text{–}HJ\text{–}I\text{–}AB\), consisting of 5 link-sets. This results in a deficiency of 4 in the link rank. This is

\(^{6}\)The maximum number of ears is \(|N_3| - 1\); assuming exactly one new node is added per ear. \(|N_3|\) denotes the number of nodes in the 3-graph.
shown in Figure 3.11(c). Assume that the links for which the metrics are known
priori are A–C, E–G, G–H, and I–B. We have link-set HJ–I of size two in which we
do not know either link metric. We now replace the entire last open-chain using a
single link, which falls inside the component AB. The last ear in the next iteration
is $m–AB–m$, which results in an open-chain $m–A–B–m$, as shown in Figure 3.11(d).
This chain now has three link-sets, thus adds a deficiency of 2 to the link rank. The
total link rank deficiency is 6. The total number of links in the network is 17, hence
the link rank of the network is 11.

Observe that the logical link representing the open-chain in the first iteration
appears as a link in the second iteration. This has two interpretations. First, if we
assume that the metrics on links $m–A$ and $m–B$ are given, then we do not need any
other link metrics in the link-set A–B. We may use the $m–A$ and $m–B$ metrics to
compute the metric of the logical link (shown in bold) in A–B. This implies that
whatever path-segment we use from A to B, traversing the chain AB–CDEF–G–
HJ–I–AB, we know the metric of the path segment. Thus, we can compute the link
metrics H–I and J–I. The second interpretation is obtained if we assume that only
link $m–A$ is known. Then, we need to know the metric on either the link A–B or
of the logical link between A and B in order to compute the metric on link $m–B$.
Assume that the logical link metric needs to be known. This implies that we need
to know one of the two link metrics: H–I or J–I. Thus, the logical link points to a
link of unknown metric in a link-set.

3.7 Link Rank of a 1-Edge Connected Network

Figure 3.12 outlines the steps to compute the link rank of a 1-edge connected net-
work. The link rank of a 1-edge connected network is the same as the link rank
of the 2E-component that contains the measurement node. Note that for all links
outside of the component, we cannot establish a measurement cycle, thus contribute
to rank deficiency.
Procedure Compute Link Rank in 1E Network:

Input: $\mathcal{G}(\mathcal{N}, \mathcal{L})$ - A 1E-network $\mathcal{G}$, where $\mathcal{N}$ and $\mathcal{L}$ denote the set of nodes and links, respectively.
Output: $R_{\mathcal{G}}$ - Link rank.

1. Initialize $\mathcal{G}' \leftarrow \mathcal{G}; d \leftarrow 0$.
2. Obtain the 2-graph of $\mathcal{G}'$ namely $\mathcal{G}'_2$.
3. Let the component containing $m$ be denoted by $C$. Use the procedure in Figure 3.10 to compute the rank in $C$, say $R_C$.
4. $R_{\mathcal{G}} \leftarrow R_C$. // Compute link rank.
5. Stop.

Figure 3.12: Steps to compute the link rank of a 1-edge connected network.

**Complexity Analysis:** For 1E-networks, we first need to compute the 2E-components which can be achieved using depth first search numbering, which requires $O(|\mathcal{L}|)$ time. Hence, the link rank can still be obtained in $O(|\mathcal{L}|)$ time.

3.8 Discussion

We discuss below some interesting properties of the link rank of a 2E-network.

**A set of links whose metrics need to be given:** Although we did not state this explicitly, our algorithm makes it possible to compute various sets (of links) each of whose cardinality is the deficiency in link rank of the network. Given the metrics of all the links in any such set, all the remaining links in the network can be identified. One such example is as follows. We can pick one link from the first link-set in the ear under consideration at every iteration to be the one that gets folded in as a logical link. Hence, we add one link from each of the other $z-1$ link-sets in the ear to the set we are building. Once our algorithm terminates, we have a set of links whose metrics need to be given as desired. Given the set of links, we may solve for the remaining unknowns with equations (3)–(6) in $O(|\mathcal{L}|)$ time. This is because, these equations already show that in any link-set of size two, given one link metric, the other can always be identified. Further, the identifiability of the first link-set
that is assumed to be unknown is determined once the ear folds as a logical link. If it is found to be identifiable, it amounts to a path in the ear (that was folded as) being identifiable. Since, all but one link-set is unknown in this path, the unknown link-set is also identifiable. It it is found that the logical link is not identifiable, then we will require a link metric on the link-set. Thus, it is always possible to compute a set of link metrics that need to be given in a network so that all other links can be identified.

**Given some set of known link metrics:** If we assume that we are given some\(^7\) set of known link metrics and we are asked to find additional links for which the metrics need to be known, then we may employ a simple strategy. If two nodes in the network are connected by a link with a known metric, we merge the two nodes and remove the known link. This is because, we can traverse the path from one node to the other without incurring an unknown metric on the way. We analyze the resultant network for its link rank. Thus, any a priori knowledge of link metrics can directly be factored into our algorithm to compute the remaining rank deficiency.

### 3.9 Computing the linearly independent cycles

We now discuss how to compute the set of linearly independent cycles in the last ear. The same procedure may be applied repeatedly to the sequence of ears as the last ear gets folded as a logical link.

#### 3.9.1 Linearly independent cycles in the last ear

Let the number of link-sets in the last ear be denoted by \( z \). Let \( z_1 \) be the number of link-sets of size one and \( z_2 \) be the number of link-sets of size two \((z = z_1 + z_2)\). The last ear is made up of a chain of 3E-components that are of either type-1 or type-2. As each type-2 component may be decomposed into a chain of type-3 components, the last ear may be viewed as as a combination of type-1 components,

\(^7\)It is important to remember that any set of cardinality that addresses the deficit in a full link rank need not be sufficient to identify all other link metrics.
type-3 components and $z$ link-sets as shown in Figure 3.13.

Figure 3.13: The last ear with $z$ link-sets

Let the total number of links inside all the type-1 and type-3 components in the last ear (i.e. excluding all link-sets) add up to $L_{int}$. Then the total number of links in the last ear, $L_{total}$ is given by:

$$L_{total} = L_{int} + z_1 + 2z_2$$

We know from Section 3.6 that the deficiency in rank of the last ear with $z$ link-sets is given by $z - 1$. Thus, our goal is to compute a set of linearly independent cycles whose cardinality is $L_{total} - (z - 1)$, or equivalently $L_{int} + z_2 + 1$.

The outline of computing the set of linearly independent cycles in the last ear is shown in Figure 3.14.

We now detail the steps involved in computing these cycles. Chapter 2 showed that there are as many distinct linearly independent cycles in a 3E-network as the number of links in the network, all of which start and end at a given node, say $m$. These set of cycles are obtained by first constructing three link-independent trees rooted at $m$. These trees, referred to as red, blue and green are derived from a sequence of segments (paths/cycles) that attach to previously computed segments. The first segment is a cycle that starts and ends at $m$. Once these trees are computed, for any node $n$, the three paths along each of the three trees are mutually link-disjoint. Thus, we may combine two of the tree paths at a time to form cycles.

In the following sub-sections, we will look at how we can construct trees that enable us to compute various linearly independent cycles in type-1 and type-2 components. Once we understand the cycles through each of these components, we can
**Compute linearly independent cycles in last ear:**

**Input:** \( G(\mathcal{N}, \mathcal{L}) \) - A 2E-network \( G \), where \( \mathcal{N} \) and \( \mathcal{L} \) denote the set of nodes and links, respectively.

**Output:** \( C_G \) - A set of \( R_G \) linearly independent cycles.

1. Transform the last ear with \( L_{total} \) links into a chain of link-sets, say \( z \).
2. For every type-1 component in the last ear, compute a basic path. Call this path as \( P_1 \).
3. For every type-2 component in the last ear, compute two basic (disjoint) paths. Call these paths as \( P_1 \) and \( P_2 \).
4. Compute the basic path for the last ear \( B \) by combining all the basic paths \( P_1 \) from the type-1 and type-2 components. When the last ear gets folded as a logical link, any further paths/cycles in which his logical link appears, the logical link will be replaced by this basic path.
5. Compute \( L_{total} - (z - 1) \) linearly independent cycles.
6. Attribute all but the one cycle that traverses through the basic path \( B \) to the last ear. [Note: As we replace the last ear with the logical link, we will consider the basic path in the next iteration, thus we will not count this basic path as part of the last ear.]
7. Stop.

---

**Figure 3.14:** Steps to compute the linearly independent cycles in the last ear.

Fix the basic paths and count the cycles in the last ear. The naming of the external path segments shown in dotted lines in the following sub-sections are re-used to keep the notation simple. They do not necessarily indicate the same path. The exact external path for each component will be defined towards the end of the section once the basic paths get fixed in each component.

### 3.9.2 Linearly independent cycles through a type-1 component

Consider Figure 3.15 for reference. Since the type-1 component, by definition, remains 3E-connected even when the external paths are removed, we remove \( p_1 \) and \( p_2 \) and construct three link-independent trees such that two trees are rooted at \( m_1 \) and one tree is rooted at \( m_2 \). This is achieved by introducing a virtual node \( v \) and
connecting three virtual links to it, as shown in Figure 3.15(b).

![Diagram](image)

Figure 3.15: (a) $C$ is 3E-connected after removal of path segment $p_1-m-p_2$. (b) Three trees constructed at a virtual node $v$.

Let the number of links strictly internal to the type-1 component be $l_1$. Then, we have $l_1 + 3$ linearly independent cycles in the network shown in Figure 3.15(b). When $v$ and the virtual links are removed, we lose two cycles. One of them is the red and blue cycle of node $m_1$. The other is because the red and green, the blue and green cycles of $m_1$ are now the same, they are simply the green path from $m_1$ to $m_2$. Hence, we have a total of $l_1 + 1$ unique paths from $m_1$ to $m_2$.

Of these, the paths resulting from the combination of the red and green tree paths at a node, the blue and green tree paths at a node would still be valid since they become paths that go from $m_1$ to $m_2$ (when $v$ is removed). However, the cycles resulting from the red and blue paths from a node are not useful since they would start and terminate at $m_1$ (when $v$ is removed).

To make up for the loss of these cycles, we compute a modified version of the red-blue cycle that branches off somewhere along the red or blue path to $m_1$ and is forced to reach $m_2$. The exact path is described in Theorem 7 in Appendix-B. We create one such path to replace every red-blue cycle. The fact that these cycles are linearly independent can be seen in the proof of Theorem 7 where the individual link metrics inside the type-1 component are identified.

Thus, to summarize, the $l_1 + 1$ unique linearly independent cycles are:

1. The combination of the red and green paths from a node inside $C$ along with the external path $p_1 - m - p_2$.

2. The combination of the blue and green paths from a node inside $C$ along with...
the external path \( p_1 - m - p_2 \).

3. The paths that are modified versions of the combination of the red and blue paths along with the external path \( p_1 - m - p_2 \).

**The basic path \( \mathcal{P}_1 \):** We choose the basic path \( \mathcal{P}_1 \) for a type-1 component as the red path from \( m_2 \) to \( m_1 \). This gets stitched as part of the basic path \( \mathcal{B} \) for the ear. Hence, each type-1 component provides \( l_1 \) cycles excluding the basic path.

3.9.3 Linearly independent cycles in a type-2 component

Consider a type-2 component \( C \) that is made up of \( k \) type-3 components as was shown in Figure 3.7. Let \( m_1 \) and \( m_2 \) denote the two nodes at either ends of \( C \), in \( C_1 \) and \( C_k \) respectively through which we enter and exit the type-2 component. For each of the type-3 components in \( C \), we will now show how to compute three link-independent trees rooted at \( m_1 \).

**Cycles through an end component**

We consider component \( C_k \) for illustration. The case for \( C_1 \) is analogous. Consider Figure 3.16 (a) for reference. We treat the external path from \( m \) to \( p_1 \) together with the path from \( m_2 \) back to \( m \) as a single link of unknown metric and construct three link-independent trees rooted at \( m_1 \)\(^8\) as shown in Figure 3.16(b). \( p_3 \) and \( p_4 \) are also assumed to be shrunk as single links of unknown metrics.

\[ \text{Figure 3.16: (a) End component } C_k, \text{ (b) Trees being constructed in } C_k \]

\(^8\)\( C_k \) by definition is 3E-connected even in the absence of the external path. We are using the external path here to ensure that \( m_1 \) along with \( C_k \) is 3E-connected so that we can construct the trees.
Let the number of links internal to the type-3 component $C_k$ be $l_{3k}$. Then, we have $l_{3k} + 3$ linearly independent cycles that start and end at $m_1$. As in a type-1 component, we convert all of them to paths starting from $m_1$ and ending at $m_2$. One difference to note is that paths $p_3$ and $p_4$ remain unlike the case of the virtual links for a type-1 component. In the process of converting the linearly independent cycles to paths from $m_1$ to $m_2$, we lose one path since we cannot compute a replacement for a red-blue cycle that traverses both $p_3$ and $p_4$ and reaches $m_2$ since there is a link-set of size two incident on $C_k$. Hence, we get a set of $l_{3k} + 2$ cycles in total summarized below.

1. The combination of the red and green paths from a node along with the external path $p1 - m - p2$.

2. The blue and green paths from a node combined with the external path $p1 - m - p2$.

3. The modified red and blue paths that start at $m_1$ and end at $m_2$ along with the external path $p1 - m - p2$. We lose out one cycle in this set because there is a link-set of size two incident on the component.

The missing cycle and the fact that these cycles are also linearly independent can be observed in the proof of Theorem 9 in Appendix-B.

**Cycles through other type-3 components**

![Diagram](a) (b)

Figure 3.17: (a) End component $C_i$. (b) Trees being constructed in $C_i$

We now compute cycles through the remaining type-3 components. Assume that we are working with $C_i$ having $l_{3i}$ links in it as shown in Figure 3.17(a). Then, we
have $l_{3i} + 5$ cycles of which we lose out on not being able to convert two of the cycles into paths. These are the red-blue cycle involving the first segment containing $p_3, p_4$ and the red-blue cycle containing the last segment $p_5, p_6$. Thus, we have a total of $l_{3i} + 3$ cycles in each of the type-3 components, $C_2, C_3, .., C_{k-1}$.

![Figure 3.18: Two common cycles across type-3 components.](image)

The basic paths $\mathcal{P}_1, \mathcal{P}_2$: We are now ready to compute the basic paths and count the cycles through the type-2 component as a whole. We will show here that there exist two link-disjoint paths from $m_1$ to $m_2$ in $C$ that lie strictly on the trees constructed in each of the type-3 components. This will help us in fixing the external path segments for each type-3 component. Observe that for each component $C_i$, in which four links are incident, there has to be a pairing of two links, one from each side of the component that belong to the same tree. As an example, reconsider Figure 3.17(b). Either $p_5$ or $p_6$ will have to be on the red tree. In either case, we can take the red and blue tree paths inside $C_i$ to reach the corresponding links on the other side, namely $p_3$ and $p_4$. We continue to take these link-disjoint tree paths to reach $m_1$ and $m_2$ on either side. Hence, there exist two cycles that are common to each type-3 component. These two cycles can also be viewed as the red-green and blue-green cycles of $m_2$. We fix the red and blue paths of $m_2$ as the basic paths $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively.

Hence, by subtracting these two common cycles in $k - 1$ of the $k$ type-3 components, we have the following as the number of cycles through a type-2 component

\[
\sum_{i=1,k} (l_{3i} + 2) + \sum_{i=2}^{k-1} (l_{3i} + 3) - 2(k - 1) \\
= (\sum_{i=1}^{k} l_{3i}) + 4 + 3(k - 2) - 2(k - 1) \\
= (\sum_{i=1}^{k} l_{3i}) + k
\]
Of these, we choose \( P_1 \) to be stitched as part of the basic path \( B \) for the last ear. Thus, we have \((\sum_{i=1}^{k} l_{3i}) + k - 1\) cycles through a type-2 component.

3.9.4 Counting cycles across all components in the last ear:

The basic path \( B \) that is common to all type-1 and type-2 components in the last ear is stitched as defined in Figure 3.14. From the above subsections, we can see that the total number of linearly independent cycles in the last ear is given by \( L_{int} + z_2 \) which is the same as \( L_{total} - z \). This count is excluding the cycle containing the basic path \( B \) which will be accounted for once the ear folds as a logical link in the successive iteration.

**Complexity:** The complexity of computing the cycles is dominated by having to construct three link-independent trees in each of the 3E-components. This can be achieved in a total time of \( O(|N|^2) \) since the largest 3E-component can have \(|N|\) nodes at most.

3.9.5 Cycles on the example network

Consider Figure 3.19 which is the same example network considered earlier. The first 3E-component along the last ear is of type-1 and has six links in it. The three link-independent trees constructed at \( C \) are shown in Figure 3.20(a). The seven cycles are listed below.

1. \( m-A-C-F-E-G-J-I-B-m \) (contains \( B \))
2. \( m-A-C-D-E-G-J-I-B-m \)
3. \( m-A-C-D-F-E-G-J-I-B-m \)
4. \( m-A-C-F-D-E-G-J-I-B-m \)
5. \( m-A-C-F-D-C-E-G-J-I-B-m \)
6. \( m-A-C-F-E-D-C-E-G-J-I-B-m \)
Figure 3.19: Cycles on the example network

Figure 3.20: (a) Trees in type-1 component, (b) Decomposition of the type-2 3E-component (b) Trees being constructed in $C_2$

7. $m$–A–C–E–G–J–I–B–m

For the second 3E-component, which is of type-2 containing three type-3 components $C_1, C_2, C_3$ (or two link-sets) shown in Figure 3.20(b), we obtain the following cycles. $C_1$ is a type-3 end-component having zero internal links. Thus, there are a total of two cycles,

1. $m$–A–C–F–E–G–H–I–B–m
2. $m$–A–C–F–E–G–J–I–B–m (contains B)

In $C_2$, the trees constructed are shown in Figure 3.20 (c). $C_2$ is a type-3 component has one internal link and hence a total of four cycles,

1. $m$–A–C–F–E–G–H–I–B–m (repeated)
2. $m$–A–C–F–E–G–H–J–I–B–m
\( C_3 \) is type-3 end-component with zero internal links. Hence, it has two cycles,

1. \( m-A-C-E-G-H-I-B-m \) (repeated)

2. \( m-A-C-E-G-J-I-B-m \) (contains \( B \))

Note that \( C-F-E \) is \( \mathcal{P}_1 \) in the type-1 component while \( G-J-I \) and \( G-H-I \) constitute \( \mathcal{P}_1, \mathcal{P}_2 \) respectively in the type-2 component. Thus, the basic path \( \mathcal{B} \) for the last ear is \( A-C-F-E-G-J-I-B \). We have a total of three new cycles in the type-2 component excluding the cycle containing \( \mathcal{B} \). The count of three is also the number of internal links (one) plus the number of link-sets of size two in the type-2 component (two). Thus, there are a total of nine linearly independent cycles in the last ear excluding the cycle containing \( \mathcal{B} \). When the last ear folds, the tail of the ear sequence will have two linearly independent cycles. This can be seen on the multi-graph in Figure 3.11(d). Thus, we have a total of eleven linearly independent cycles that achieve the link rank of the network. Of the two cycles in Figure 3.11(d), one of them traverses the logical link (the cycle containing \( \mathcal{B} \)) that was not counted in the previous iteration.

3.10 Appendix

A. Properties of 3-graphs and their ear-decomposition

The decomposition of a given graph into its three edge connected components (3E-components) is well studied and there are several linear time algorithms for this decomposition (59; 61). Two nodes are said to belong to the 3E-component (super-node) if there exist three link-disjoint paths between them. For the purposes of our problem setting, \( \mathcal{G} \) is assumed to be 2E-connected. Below, we discuss some properties of the 3-graph \( \mathcal{G}_3(\mathcal{N}_3, \mathcal{L}_3) \) of a 2E-connected graph \( \mathcal{G}(\mathcal{N}, \mathcal{L}) \). Here, \( \mathcal{N}_3 \) denotes the 3E-connected components of \( \mathcal{G} \) and \( \mathcal{L}_3 \) is the set of links in the 3-graph and is a subset of the links in \( \mathcal{G} \). Notice that this decomposition can result in multiple edges between the super nodes and hence \( \mathcal{G}_3 \) may be a multigraph. \( \mathcal{G}_3 \) has some interesting properties:
1. $G_3$ consists of simple cycles such that any two of them share at most one super-node (62).

2. $G_3$ is minimally two edge connected.

We refer the interested reader to Lemma 3.1 in (62) for a proof of property (1). Property(2) can be deduced from (1) as follows. Property (1) already shows that $G_3$ is two edge connected. Assume $G_3$ is not minimally two-edge connected. Then there exists at least one link $\ell$ whose removal keeps $G_3$ two edge connected. Let the two super-nodes on either end of $\ell$ be $n_1, n_2$. There exist two link-disjoint paths, say $P_1, P_2$ between the super-nodes $n_1, n_2$ in $G_3 \setminus \ell$. This implies that there are two simple cycles in $G_3$ that meet at more than one super-node thus violating property (1). Hence $G_3$ is minimally two edge connected.

Other corollaries that arise from the above properties are: Every link $\ell \in \mathcal{L}_3$ is part of at least one two edge cutset in $G_3$ and that any two neighboring nodes in $G_3$ can have at most two edges between them.

**Ear decomposition of $G_3$:** An ear decomposition (49) of a two-edge connected graph $G_3 = P_0 \cup P_1 \cup \ldots \cup P_j$ where $P_0$ is a simple cycle and each $P_i$ is a simple cycle or simple path with only one or two vertices in common with $P_0 \cup \ldots \cup P_{i-1}$ respectively. Each $P_i$ is called an ear. In our setting, we construct an ear decomposition with the first ear starting and ending at the measurement node $m$.

**Lemma 2** Every ear in the ear decomposition of $G_3$ is a simple cycle.

**Proof** Assume there is an ear $P_i$ that attaches to two distinct super-nodes $n_1, n_2$. By the definition of the ear decomposition, the subgraph that contains $P_0 \cup P_i \ldots \cup P_{i-1}$ is two edge-connected. The addition of $P_i$ results in three link-disjoint paths between the two super-nodes. Thus, $n_1$ and $n_2$ cannot be distinct nodes in $G_3$, a contradiction. □

**Lemma 3** The ear decomposition of $G_3$ is unique.
Proof We know that (i) the entire graph is composed of a union of simple cycles where any two cycles can have at most one super-node in common (Property 1) and (ii) Each ear is a simple cycle (Lemma 2). Thus, it can be seen that the ears in the decomposition of \( G_3 \) are unique. Although their sequence may be different in different ear decompositions, the actual ears are unique for a given \( G_3 \) and are algorithm independent. This is important as the ear decomposition maintains a unique representation of the given 2E-connected graph \( G \). □

Properties of super-nodes in \( G_3 \):

1. Every super-node has an even degree \( \geq 2 \).

2. Any cycle originating from the measurement node that passes through a super-node \( n \) has to traverse two unique exit links of \( n \).

Let the first ear in which a super-node \( n \) is added be \( P_i \). Let the two links of \( n \) in \( P_i \) be \( \ell_1, \ell_2 \). The super-node \( n \) could be part of several other ears as long as they start and end at \( n \) since all ears are simple cycles. This proves the first property.

Since every other ear that \( n \) is part of will be added after \( P_i \) in the decomposition, none of the super-nodes in those ears can reach \( m \) without traversing \( n \). Hence, the only way to reach \( m \) from \( n \) is to take one of the two links \( \ell_1, \ell_2 \). Alternatively, any cycle containing \( m \) and \( n \) has to traverse links \( \ell_1, \ell_2 \). This proves the second property.

Decomposition of type-2 components: We now consider the decomposition of the type-2 components in the 3-graph, namely 3E-components that become 2E-connected when the outgoing links are removed.

Lemma 4 Given a 3E-connected component \( C \) that is connected to a measurement node through paths \( p_1, p_2 \) and is only 2E-connected upon the removal of \( p_1 - m - p_2 \), its internal structure is a chain of sub-components as shown in Figure 3.7.

Proof We prove the lemma by contradiction. Consider \( C \) after the external path has been removed. We now decompose this sub-graph into 3E-components. From
the properties of a 3-graph of a 2E-connected graph, this decomposition has to result in a graph composed of a union of simple cycles. Let us assume however that this does not result in a chain of simple cycles as shown in Figure 3.7. Then there exist some other cycles that branch off this chain. Any such branch has to end in a cycle containing at least one sub-component \(C_i\) of degree two\(^9\). Note that \(C_i\) is neither \(C_1\) nor \(C_k\). This in turn implies \(C_i\) does not have three link-disjoint paths to any other sub-component. This contradicts the fact that all sub-components belong to \(C\) which is a 3E-component to begin with. Thus, the chain of sub-components is the only possible decomposition of a type-2 component. □

B. Construction and properties of linearly independent cycles and paths

In this section, we state and prove several theorems that establish the identifiability of any link inside a type-1 3E-component and similarly of any link inside type-3 components. We assume that the paths can be non-simple (i.e) they can traverse the same node more than once, however any link appears at most once. We let \(w_\ell\) denote the unknown constant weight on link \(\ell\).

\textbf{Theorem 7} Given a 3E-connected network \(\mathcal{G}(N, \mathcal{L})\) that has two measurement nodes \(m_1, m_2\) where measurement paths start at \(m_1\) and terminate at \(m_2\), we can identify all the constant additive link metrics in the network.

\textbf{Proof} The proof of the theorem relies on having three link-independent trees. The construction of these trees is not required to understand the proof. An interested reader can find the construction procedure in Chapter 5.

\textbf{Notation and Preliminaries:} We consider a minimally 3E-connected graph throughout the proof for simplicity and finally show how all links on any given 3E-connected graph can be identified. The three link independent trees are denoted by \(T_r, T_b\) and \(T_g\). For any node \(n\), its red, blue and green paths to the respective measurement nodes are denoted by \(P_r(n), P_b(n), P_g(n)\) respectively. The corresponding

\(^9\) Otherwise, it is easy to see that there will be simple cycles sharing more than one super-node violating Property (1) of 3-graphs.
path metrics are denoted by $W_r(n), W_b(n), W_g(n)$. A path between two nodes $n_1, n_2$ on a specific tree, say red is denoted by $P_r(n_1, n_2)$. The path metric is denoted by $W_r(n_1, n_2)$.

The tree construction is based on an ordered sequence of segments\textsuperscript{10} where each segment is a path or cycle that attaches to the previous segments. Let the nodes in a segment be numbered $n_0$ through $n_{k+1}$, where $n_0$ and $n_{k+1}$ denote the two end nodes (previously added) to which a segment connects\textsuperscript{11}. After each segment computation, $T_r, T_b$ are updated as follows. $n_0 ← n_1 ← n_2... ← n_k$ is added to $T_r$ and $n_1→n_2...→n_k→n_{k+1}$ is added to $T_b$. Thus, in any segment all but two links belong to both the red and blue trees. The other two links that attach to previous segments are called attachment links. These attachment links may or may not be part of the green tree.

The outline of the proof is as follows.

1. We construct three link-independent spanning trees\textsuperscript{12}, namely red, blue and green. The red and blue trees are rooted at $m_1$. The green tree is rooted at $m_2$.

2. We classify the links in the network into three different groups; (i) links that belong to the red and blue trees, (ii) links that belong only to the green tree, and (iii) all other links.

3. For each group, we identify the link metrics.

**Step 1:** We add a virtual node $v$ and add two links between $v$ and $m_1$. We add another link between $v$ and $m_2$. This new network remains three-edge connected. We now construct three link independent trees rooted at $v$ such that $m_2→v$ is the

\textsuperscript{10}These segments have properties that they are non-separating induced paths/cycles

\textsuperscript{11}Note that $n_0$ and $n_{k+1}$ need not be distinct since the paths from any node to the root are only guaranteed to be link-disjoint.

\textsuperscript{12}The properties of the trees are that for any node $n$, the paths to the root on the three trees are mutually link-disjoint
final green edge that all nodes traverse to reach $v$. We remove the virtual node $v$ and links attached to it.

**Step 2:** Since we consider a minimally 3E-connected network, every link in the network is on at least one tree and at most two trees. Based on this observation, links in $G$ can be classified/partitioned into one of six mutually exclusive sets namely $L_{rb}, L_{rg}, L_{bg}, L_r, L_b, L_g$ where each set is representative of the trees to which a link in that set belongs to. We now make groups as follows. The first group is $L_{rb}$. The second group is $L_g$. The third group contains links in all other sets. Note that the third group consists of all attachment links.

**Step 3:** We consider one group at a time and show identifiability of links in each group. However, considering them in the order of Group 1 followed by Group 3 and finally Group 2 simplifies the proof considerably.

**Step 3a:** Identifying links in the Group 1 (links in red and blue trees):

Consider a link $l = (n_1, n_2) \in L_{rb}$ as shown in Figure 3.21. Without loss of generality, assume that the red neighbor of $n_2$ is $n_1$. Then, $P_r(n_1)$ is link-disjoint from $P_b(n_1)$ by definition. Hence, $P_b(n_2)$ is also link-disjoint from $P_r(n_1)$ as shown in Figure 3.21. Let $LCA_g(n_1, n_2)$ denote the lowest common ancestor on the green tree for $n_1$ and $n_2$. We make the following measurements that start at $m_1$ and terminate at $m_2$.

\[
W_r(n_1) + W_g(n_1) = \alpha_1 \tag{3.5}
\]
\[
W_b(n_2) + W_g(n_2) = \alpha_2 \tag{3.6}
\]
\[
W_r(n_1) + w_l + W_g(n_2) = \alpha_3 \tag{3.7}
\]
\[
W_b(n_2) + w_l + W_g(n_1) = \alpha_4 \tag{3.8}
\]
We obtain \( w_l \) as \( \frac{1}{2} [(3.7) + (3.8) - (3.5) - (3.6)] \).

**Step 3b:** Identifying links in Group 3 (All attachment links): Let the two attachment links of a segment be \( l_1, l_2 \). We consider their identifiability one segment at a time in the same order used to construct the trees.

Figure 3.22 shows a segment and its two attachment links. The sub-graph \( G_i \) contains all previous segments. The figure also shows a link \( X - Y \) which has to be present since the network is three-edge connected\(^{13}\). Node \( C \) is the first node encountered along \( P_r(X) \) that also belongs to either \( P_r(n_0) \) or \( P_b(n_{k+1}) \). In the figure, \( C \) is shown to be on \( P_b(n_{k+1}) \).\(^{14}\)

We compute \((w_{l_1} - w_{l_2})\) and \((w_{l_1} + w_{l_2})\), thus leads to identifying \( w_{l_1} \) and \( w_{l_2} \).

We compute \((w_{l_1} - w_{l_2})\) by making the following measurements. First, we combine the red and green paths at \( n_1 \) and then the blue and green paths at \( n_k \).

\[
\begin{align*}
W_r(n_0) + w_{l_1} + W_g(n_1) &= \alpha_5 \quad (3.9) \\
W_b(n_{k+1}) + w_{l_2} + W_b(n_1,n_k) + W_g(n_1) &= \alpha_6 \quad (3.10)
\end{align*}
\]

In the two equations above, the only unknowns are \( W_g(n_1), w_{l_1} \) and \( w_{l_2} \) as all other link metrics are either in Group 1 or are attachment links in earlier segments. In either case, their metrics have been identified. We obtain \((w_{l_1} - w_{l_2})\) as \([(3.9) - (3.10)]\).

We now make the following two measurements to compute \((w_{l_1} + w_{l_2})\). We first establish a path from \( m_1 \) that traverses both attachment links and then eventually reaches \( m_2 \). The second measurement is to remove the effect of the unknown metrics in the first measurement.

\[
\begin{align*}
W_r(n_0) + w_{l_1} + W_b(n_1,n_k) + w_{l_2} + \\
W_b(n_{k+1},C) + W_r(X,C) + w_{xy} + W_g(Y) &= \alpha_7 \quad (3.11)
\end{align*}
\]

\(^{13}\)X need not be distinct from \( n_0 \) or \( n_{k+1} \) since the network is not necessarily three-vertex connected.

\(^{14}\)The case when node \( C \) is on \( P_r(n_0) \) is similar and the measurement paths can be established accordingly.
\[ W_b(X) + w_{xy} + W_g(Y) = \alpha_8 \quad (3.12) \]

We obtain \((w_{l_1} + w_{l_2})\) as \([(3.11) - (3.12)]\). Again, this is because all other link metrics are either in Group 1 or are attachment links of previous segments.

\[ W_r(n_1) + w_l + W_g(n_2) = \alpha_9 \quad (3.13) \]
\[ W_r(n_2) + W_g(n_2) = \alpha_{10} \quad (3.14) \]

Since \(P_r(n_1), P_r(n_2)\) cannot contain any link in \(L_g\), the individual link metrics in these paths are known from the results in the previous steps. Hence, we obtain \(w_l\) as \([(3.13) - (3.14)]\).
Finally, if the network was not minimally 3E-connected to start with, we can prune the network and identify links on a minimal network as above. Then, we put back one link at a time and identify its metric by establishing a path from \( m_1 \) to \( m_2 \) that traverses the link. The only unknown on any such path would be that of the added link. □

**Theorem 8** Given a component \( C \) that is connected to a measurement node through paths \( p_1, p_2 \), it is possible to uniquely identify all link metrics inside \( C \) using measurement cycles even when the path metrics are unknown provided \( C \) remains three-edge connected upon the removal of \( p_1 - m - p_2 \).

**Proof** Let the two paths \( p_1, p_2 \) be incident on nodes \( m_1, m_2 \) respectively in \( C \). We now construct the measurement paths exactly as in Theorem 7 between \( m_1, m_2 \). We then append these measurement paths with \( p_1, p_2 \) to get measurement cycles in the given network. It is fairly easy to see that every measurement path in the proof of Theorem 7 would now have a \( W(p_1) + W(p_2) \) added to it. However, the same proof also shows that two measurements are always subtracted to identify any link in the network. Thus, the term \( W(p_1) + W(p_2) \) does not affect identifying metrics of any link inside \( C \). □

**Theorem 9** Given a component \( C \) as shown in Figure 3.8, it is possible to uniquely identify all link metrics inside \( C \) using measurement cycles even when the path metrics are unknown provided \( C \) remains three-edge connected upon the removal of \( p_1 - m - p_2 \).

**Proof** Let the path \( p_2 \) be incident on node \( y \) in \( C \). We remove \( p_1 - m - p_2 \) and replace it with a virtual link between nodes \( x, y \). We shrink the path segments \( p_3, p_4 \) to links. This resulting graph is clearly 3E-connected. We now construct trees as in Theorem 7 with two of them rooted at \( x \) and one rooted at \( y \). We then do away with the virtual link that was added between \( x, y \).

We first prove that after the virtual link \( x - y \) is removed, we can still identify all link metrics in \( C \) using paths from \( x \) to \( y \). We later add the external paths \( p_1, p_2 \)
and show that it does not affect the result. Consider Figure 3.24 for reference. We use the same notation of $m_1$, $m_2$ for the measurement nodes as in the previous theorems. These now represent nodes $x$, $y$ of Figure 3.8. The two unknown path metrics incident at $x$, namely that of $p_3$ and $p_4$ are shown here as links with unknown metrics $\beta_1$ and $\beta_2$, respectively. These two link metrics cannot be identified (as yet) since they represent a link-set inside a type-2 node. We now make the same measurements as in the previous theorems and observe if we can still identify all other links when the links incident at $m_1$ cannot be identified.

![Figure 3.24: Illustration for identifying link metrics for scenario in Figure 3.8, where $m_1 = x$ and $m_2 = y$.](image)

The measurements for the links in $\mathcal{L}_{rb}$, $\mathcal{L}_{g}$ (Groups 1,2) are not affected by the lack of knowledge of $\beta_1$, $\beta_2$. This can be verified in the respective measurements involved in Theorem 7. One subtle point is that for links in Group 2, it was important that all attachment links had been identified. We now show that those links can still be identified.

For the attachment links, we assumed that we are able to identify them in sequence. Now, since the first segment contains $\beta_1$, $\beta_2$ the argument in Theorem 7 does not hold anymore. However, it is still possible to identify the link metrics as follows:

We consider the attachment links in the sequence of the segments as before. The
measurements to compute \((w_{l_1} - w_{l_2})\) previously are now rewritten as\(^{15}\):

\[
\beta_1 + W_r(n_0, A) + w_{l_1} + W_g(n_1) \quad (3.15)
\]

\[
\beta_2 + W_b(n_{k+1}, B) + w_{l_2} + W_b(n_1, n_k) + W_g(n_1) \quad (3.16)
\]

If we start with the first segment, it is possible to compute the value of \((\beta_1 - \beta_2)\). \(\frac{(3.15) - (3.16)}{}\) gives \([w_{l_1} - w_{l_2}] + (\beta_1 - \beta_2)]\).

Computing \((w_{l_1} + w_{l_2})\): In Theorem 7, this was done using the two measurements (Equations 3.11, 3.12). The same measurements now become:

\[
\beta_1 + W_r(n_0, A) + w_{l_1} + W_b(n_1, n_k) + w_{l_2} +
W_b(n_{k+1}, C) + W_r(X, C) + W_g(Y) \quad (3.17)
\]

\[
\beta_2 + W_b(B, X) + W_{xy} + W_g(Y) \quad (3.18)
\]

\(\frac{(3.17) - (3.18)}{}\) gives \([w_{l_1} + w_{l_2}] + (\beta_1 - \beta_2)]\). This is because all other link metrics are part of prior attachment links or are in Group 1 both of which have been identified. Thus, we can still solve for \(w_{l_1}\) and \(w_{l_2}\).

Adding the external path: It is easy to check that links in Groups 1,2 are not affected by having \(W(p_1) + W(p_2)\) in each of their measurements. Similarly, in the four equations involving \(\beta_1, \beta_2\), we see that adding \(W(p_1) + W(p_2)\) to each of them does not affect the measurability of links in \(C\). \(\square\)

**Theorem 10** Given a component \(C\) that is connected to a measurement node through paths \(p_1, p_2\) as shown in Figure 3.9, it is possible to uniquely identify all link metrics inside \(C\) using measurement cycles even when the path metrics are unknown provided \(C\) remains three-edge connected upon the removal of \(p_1 \rightarrow m \rightarrow p_2\).

**Proof** This proof is an extension of the proof of Theorem 9. We remove \(p_1 \rightarrow m \rightarrow p_2\) and replace it with a virtual link between nodes \(x, y\). We shrink the path segments \(p_3, p_4, p_5, p_6\) as links. This resulting graph is clearly three-edge connected. We now

\(^{15}\)The measurement paths are exactly the same as before, except that metrics on all previous attachment links may not be assumed to be known anymore.
construct trees as in Theorem 7 with two of them rooted at $x$ and one rooted at $y$. We then do away with the virtual link that was added between $x, y$.

Let the two unknown link metrics incident at $x$ be $\beta_1, \beta_2$ as before and the two link metrics incident at $y$ be $\beta_3, \beta_4$. Note that the two links incident at $y$ represent a link-set as well and hence cannot be identified.

We first prove that in this graph (after the virtual link $x - y$ is removed), we can still identify all link metrics in $C$ using paths from $x$ to $y$. As in Theorem 9, links in Groups 1,2 are not affected even in this setting. For the attachment links, the two link metrics $\beta_3, \beta_4$ belong to the very last segment. Hence, not knowing their metric does not affect the measurability of any of the other link metrics in the group. Hence, using the same measurements in Theorem 9 is sufficient to identify all link metrics in $C$.

Adding the external path: An argument along the lines used in Theorem 9 shows that adding the external path will not affect the identifiability of the links. □
CHAPTER 4

THREE EDGE INDEPENDENT SPANNING TREES - PRELUDE

A set of $k$ spanning trees rooted at vertex $r$ are said to be vertex(edge)-independent if the paths from any vertex $v(\neq r)$ to $r$ on the $k$ trees are mutually vertex(edge)-disjoint. Khuller and Schieber (1992) in (63) developed a constructive algorithm to prove that the existence of $k$-vertex independent trees in a $k$-vertex connected graph implies the existence of $k$-edge independent trees in a $k$-edge connected graph. In this chapter, we show a counterexample where their algorithm fails.

4.1 Background

In (64), Zehavi and Itai posed two conjectures.

**Vertex Conjecture:** Any $k$-vertex connected graph has $k$-vertex independent spanning trees rooted at an arbitrary vertex $r$.

**Edge Conjecture:** Any $k$-edge connected graph has $k$-edge independent spanning trees rooted at an arbitrary vertex $r$.

The authors also posed the following question, which we refer to as the Implication Conjecture.

It would be interesting to show that either the vertex conjecture implies the edge conjecture, or vice versa.

In (63), Khuller and Schieber developed a constructive algorithm to prove the implication conjecture. However, we show in this chapter that their algorithm is not correct.
4.2 Counterexample

Consider a \( k \)-edge connected graph \( G(V, E) \). The technique developed in (63) constructs \( k \)-edge independent spanning trees in three steps.

1. Transform the given \( k \)-edge connected graph \( G \) into a \( k \)-vertex connected graph, denoted by \( G'(V', E') \).

2. Assume that \( k \)-vertex independent trees are given in \( G' \).

3. Compute the edge independent trees in \( G \) from the vertex independent trees in \( G' \).

Consider the example network shown in Fig. 4.1. The graph is 3-edge connected. Let \( R \) be the root node.

**Step 1.** For each vertex \( v \) in \( G \), there are \( k \) vertices \( v^1 \) through \( v^k \) in \( G' \). They are referred to as node vertices and denoted by \( \text{group}(v) \). For each edge \( e \) in \( G \), there is a vertex \( \ell(e) \) in \( G' \). They are referred to as edge vertices. The edges in \( G' \) are defined as follows. For each edge \( e \) connected to a vertex \( v \) in \( G \), there are edges from every \( v^j \in \text{group}(v) \) to \( \ell(e) \) in \( G' \). If \( G \) is \( k \)-edge connected, then \( G' \) is \( k \)-vertex connected. Fig. 4.2 shows the transformed graph for our example. Note that node vertices of \( R \) are shown twice for drawing convenience.

**Step 2.** Assume a set of \( k \)-vertex independent spanning trees on the transformed graph. For our example, Figs. 4.3, 4.4, and 4.5 show the three vertex independent trees, \( T'_1 \), \( T'_2 \), and \( T'_3 \), respectively. The trees are rooted at one of the expanded vertices of \( R \), namely \( R_1 \).

**Step 3.** Let \( P'_j[v^1, r^1] \) denote the path from vertex \( v^1 \) to root \( r^1 \) on tree \( T'_j \), where \( j = 1, 2, \ldots, k \). Let \( T_1 \) through \( T_k \) denote the \( k \)-edge independent trees to be constructed in \( G \). According to (63), the parent of vertex \( v \) in tree \( T_j \) is defined as follows:
Let $v^{f(j)}$ be the last vertex on the path $P'_j[v^1, r^1]$ that belongs to group$(v)$. (Clearly such a vertex exists since $v^1$ is in group$(v)$ and $r^1$ is not.) Let the outgoing edge from $v^{f(j)}$ on $P'_j$ be $(v^{f(j)}, \ell(e_m))$, for $e_m = (v, u)$ in $G$. Then, the parent of $v$ in $T_j$ is defined to be $u$. 

Following the above procedure for our example, we obtain three trees $T_1$ through $T_3$ as shown in Figs. 4.6, 4.7, and 4.8, respectively. Observe that the path from $W$ to $R$ on $T_1$ uses edge $U-V$ in the direction $V\rightarrow U$, while the path on $T_2$ uses the same edge in the direction $U\rightarrow V$. The same holds true for the tree paths of node $X$ to the root as well. Thus, as the tree paths from a node to the root of the spanning trees are not edge disjoint, the trees $T_1$ and $T_2$ are not edge independent.

4.3 Analysis

Let $P_1$ through $P_k$ denote the paths from vertex $v$ to root $r$ on trees $T_1$ through $T_k$. Lemma 2.3 in (63) claims that the paths $P_1$ through $P_k$ are mutually edge-disjoint. The proof states:

Assume that there are two paths $P_1[v, r]$ and $P_2[v, r]$ that use the same edge $e$. This implies that both paths $P'_1[v^1, r^1]$ and $P'_2[v^1, r^1]$ use the same vertex $\ell(e)$, contradicting the assumption that the paths $P'_1[v^1, r^1]$ and $P'_2[v^1, r^1]$ are internally vertex disjoint.

The highlighted statement shows the flaw in the proof. In our example, $P_1[W, R]$ and $P_2[W, R]$ share the edge $U-V$. However, $P'_1[W^1, R^1]$ and $P'_2[W^1, R^1]$ are internally vertex disjoint.
Figure 4.1: Example 3-edge connected graph.

Figure 4.2: Transformed graph.
Figure 4.3: Vertex independent spanning tree $T'_1$. 
Figure 4.4: Vertex independent spanning tree $T_2'$. 
Figure 4.5: Vertex independent spanning tree $T'_3$. 

Figure 4.6: Spanning tree $T_1$. 
Figure 4.7: Spanning tree $T_2$.

Figure 4.8: Spanning tree $T_3$. 
CHAPTER 5

CONSTRUCTING THREE EDGE INDEPENDENT SPANNING TREES

In this chapter, we provide an algorithm to construct three edge independent spanning trees on three-edge connected graphs. The complexity of our algorithm is quadratic in the number of vertices in the graph.

We consider an undirected graph $G(V, E)$, where $V$ and $E$ denote the set of vertices and edges, respectively. We let $V$ and $E$ denote the number of vertices and edges, respectively. A set of $k$ spanning trees rooted at vertex $d$ is said to be edge(vertex)-independent if the paths from any vertex $v(\neq d)$ to $d$ on the $k$ trees are mutually edge(vertex)-disjoint. Edge disjoint paths imply that if the path from a vertex $x$ to $d$ on one tree contains the directed edge, $i \rightarrow j$ (also referred to as an arc), then the path from $x$ to $d$ on any other tree cannot contain either $i \rightarrow j$ or $j \rightarrow i$.

5.1 Prior Work

Itai and Rodeh (60) introduced the concept of independent trees in undirected graphs. In a later paper in 1989, Zehavi and Itai (64) proved existence of three vertex independent trees in three vertex-connected graphs. In addition, they listed three conjectures in their paper.

1. Vertex conjecture: Any $k$ vertex connected graph has $k$ vertex independent spanning trees rooted at an arbitrary vertex $d$.

2. Edge conjecture: Any $k$ edge connected graph has $k$ edge independent spanning trees rooted at an arbitrary vertex $d$.

3. Implication conjecture: Would the vertex conjecture imply edge conjecture or vice-versa?
The developments on these conjectures over the last two decades are summarized below.

1. Vertex independent trees
   
   (a) For $k = 2$, Itai and Rodeh (60) proved the existence and developed an $O(E)$ algorithm.
   
   (b) For $k = 3$, Zehavi and Itai (64) and Cheriyan and Maheswari (49) proved the existence. In addition, Cheriyan and Maheswari (49) developed an $O(V^2)$ algorithm to construct the trees.
   
   (c) For $k = 4$, Curran et.al (65) proved existence and developed an $O(V^3)$ algorithm to construct the trees.
   
   (d) For the special case of planar graphs, Huck (66) proved existence for all $k \geq 1$.

2. Edge independent trees
   
   (a) For $k = 2$, Itai and Rodeh (60) proved the existence and developed an $O(E)$ algorithm to construct the trees.
   
   (b) For all $k \geq 1$, Khuller and Schieber (63) proved the existence and developed an algorithm to construct $k$ edge independent spanning trees in a $k$ edge connected graph, provided $k$ vertex independent spanning trees exist for a $k$ vertex connected graph. In Chapter 4/ (67), we showed that the approach developed in (63) fails to provide edge-independent trees.

5.2 Contributions

In this chapter, we develop an $O(V^2)$ algorithm to compute three edge independent spanning trees rooted at any given destination node in a three edge connected graph. We achieve this by transforming a given three edge connected graph into a three vertex connected cubic graph. Our expansion is different from previous
approaches (63; 48) in that we obtain a three vertex connected graph without expanding the edges to vertices. This makes it easy to maintain the partial order across the original edges (a common technique to build edge independent trees). Further, our expansion results in a cubic graph which greatly simplifies the computation of non-separating induced paths compared to the algorithm in (49). These properties are explained in more detail towards the end of Section 5.4.5.

5.3 Organization

Section 5.4 describes our approach to construct three edge independent spanning trees in three-edge connected graphs. Following which, we have an Appendix section that contains proofs of theorems required for the construction as well an example in which we walk through all the steps of the construction procedure involved to construct the trees.

5.4 Constructing Three Edge Independent Spanning Trees

Given a graph that is three-edge-connected and a destination vertex $d$, the outline of our approach to compute the three edge independent spanning trees rooted at a destination is as follows:

1. (a) Given a graph, prune edges to consider a minimally three-edge-connected graph.
   (b) Divide the resultant graph into two-vertex-connected components. Thus, every component is three-edge and two-vertex connected (3E-2V, for short). For a given destination, identify a root vertex $r$ in each component. This root vertex is the vertex through which every path from a vertex in the component to the destination must traverse. For the rest of the chapter, we assume that $G(V, E)$ is a two-vertex and minimally three-edge-connected graph.

2. Construct three edge independent spanning trees in each 3E-2V component rooted at $r$. 
3. Merge the trees constructed in each 3E-2V component to get the final three edge independent spanning trees for destination vertex \( d \).

In the following sections, we describe the rationale for employing the above procedure and the necessary algorithmic details for each step.

5.4.1 Graph pruning and decomposition

If the given graph is not minimally three-edge-connected, then we remove certain edges to make it minimally three-edge-connected. A minimally three-edge connected graph may be one-vertex connected, i.e. there exists a vertex (an articulation vertex) whose removal will disconnect the graph. In such scenarios, we may divide the graph into two-vertex connected (2V) components. If the graph has more than one 2V component, then some articulation vertices will be present in multiple components. In addition, a component may have many articulation vertices in it. However, given any component, there exists a unique articulation vertex such that any path from a vertex in that component to the destination must traverse this articulation vertex. We refer to such an articulation vertex as the virtual destination of that component. This is true since the 3E-2V graph cannot contain cycles and hence is a tree. The problem of computing three edge independent trees rooted at the destination may then be decomposed into computing three edge independent trees in three-edge and two-vertex connected components, rooted at the virtual destinations, and merging these trees. It is also worth noting that the division of a three-edge-connected graph to 3E-2V components is fixed and does not depend on the destination vertex. The virtual destination in each component may be different for different destination vertices.

5.4.2 Overview of the construction procedure

Given a graph \( \mathcal{G} \) that is three-edge and two-vertex-connected and a destination vertex \( r \), we construct three edge independent trees, named red, blue and green all of which are rooted at \( r \). The outline of our construction is as follows:
1. Expand $G$ into a three-edge connected cubic graph (where every vertex has exactly degree three). Call this cubic graph $Q(V^*, E^*)$.

2. Construct a sequence of paths\(^1\) in $Q$ rooted at the destination vertex (as the destination vertex may be expanded in the previous step, we may pick any vertex from the expansion of the destination).

3. Compute a set of segments from each path; each of which corresponds to a path or cycle in $G$. Compute the red and blue neighbors.

4. Compute green neighbors for all vertices in $G$.

5.4.3 Expansion to a cubic graph

This step is to transform the given graph into a three vertex connected graph. The expansion to a cubic graph, where every vertex in the graph has degree three, makes it easier to understand the construction procedure and facilitates the computation of the edge independent trees which will be evident in Section 5.4.5. Any graph where vertices have degree greater than three (such as a three-edge and two-vertex connected graph) may be expanded to a cubic graph in a trivial manner. However, the construction procedure must also guarantee that the cubic graph retains the three edge connectivity, hence three vertex connectivity, property from the original graph. This is required because $Q$ has to be three vertex connected in order to be able to construct three vertex independent trees in Step 2.

Figure 5.1 shows the procedure to expand a given 3E-2V graph into a three-edge-connected cubic graph. The algorithm works by expanding a vertex $n$ with degree $d$ ($d > 3$) into $d - 2$ sub-vertices, denoted by $n_1$ through $n_{d-2}$. The sub-vertices are internally connected as a path using additional edges. The edges connected to vertex $n$ are spread across the $d - 2$ vertices such that sub-vertices $n_1$ and $n_{d-2}$ have two edges connected to them, while all other sub-vertices have exactly one edge connected to them. The choice of edges to be connected to vertices $n_1$ and $n_{d-2}$ are based on the connectivity of the graph in the absence of vertex $n$.

\(^1\)The paths have special properties that are described later.
**Procedure Cubic Expansion**

1. For every vertex $n$ whose degree $d$ is more than three, do:
   
   (a) Remove vertex $n$ and all the edges connected to it. Let the resultant graph be $\mathcal{G}_{int} \setminus \{n\}$

   (b) Sub-divide the vertex into $d - 2$ sub-vertices, denoted by $n_1$ through $n_{d-2}$. Connect vertex $n_i$ to $n_{i+1}$, where $i = 1, 2, \ldots, d - 3$.

   (c) Divide $\mathcal{G}_{int} \setminus \{n\}$ into 2E-components.

   (d) If $\mathcal{G}_{int} \setminus \{n\}$ has only one 2E-component, then assign the edges connecting this component to $n$ arbitrarily such that; vertices $n_1$ and $n_{d-2}$ have two edges connected, while all other sub-vertices have one edge connected.

   (e) If $\mathcal{G}_{int} \setminus \{n\}$ has more than one 2E-component,

   i. Select two arbitrary leaf components, say $C_1$ and $C_2$. Select two edges from each of these leaf components. Connect these edges to sub-vertices $n_1$ and $n_{d-2}$ such that one edge from each of these two components is connected to each $n_1$ and similarly to $n_{d-2}$.

   ii. Connect any remaining edges from any of the components to $n$ in any order such that the degree of every sub-vertex is exactly three.

---

Figure 5.1: Algorithm to expand a given three-edge and two-vertex connected graph into a three-edge connected cubic graph.

Consider an intermediate stage of the graph where some vertices may have already been expanded. Call this graph $\mathcal{G}_{int}$. Assume that $\mathcal{G}_{int}$ is three-edge-connected and there exists a vertex $n$ with degree more than three. We now expand vertex $n$.

The algorithm works by removing the vertex $n$ and all the edges connected to it. The resultant graph is then divided into two-edge-connected (2E) components. Observe that the graph formed by the 2E-components is 1-edge-connected. If the resultant graph has only one 2E component, then it must have at least three edges connecting the component to the vertex $n$. These edges may be distributed to the sub-vertices in any manner as long as each sub-vertex has degree exactly three. If the resultant graph has more than one 2E-component, then every leaf component\(^2\) must have at least two edges connecting to vertex $n$. In addition, every component

\(^2\) A leaf component is a 2E-component that has an edge connecting to exactly one other 2E-component.
that is connected to exactly two other components must have at least one edge
connected to vertex \( n \). We select two leaf components, say \( C_1 \) and \( C_2 \). From each
of these two leaf components, we select two edges. We attach the first edge from \( C_1 \)
to \( n_1 \) and the second edge from \( C_1 \) to \( n_{d-2} \). We repeat this for \( C_2 \). Thus, vertices \( n_1 \)
and \( n_{d-2} \) have exactly degree three and the two selected leaf components, \( C_1 \) and
\( C_2 \), have one edge attached to each \( n_1 \) and \( n_{d-2} \). Any remaining edges connecting to
\( n \) may be assigned to the sub-vertices in an arbitrary order provided each sub-vertex
has degree exactly three. Figure 5.2 shows an example of the expansion.

![Expansion of node \( n \)](image)

Figure 5.2: Structure of an example graph when vertex \( n \) is expanded.

The expansion procedure retains the three-edge-connectivity property of the
graph and eventually results in a three-vertex connected graph. Please see The-
orem 11 and Lemma 5 in the Appendix for detailed proofs. We also refer the in-
terested reader to Section 5.5.1 in the Appendix for an illustration of the expansion
procedure on an example graph.

Given a three-edge and two-vertex connected graph graph \( G(V, E) \), we denote
the cubic expansion of that graph by \( Q(V^*, E^*) \). \( V^* \) and \( E^* \) denote the number
of vertices and edges, respectively. For every vertex \( n \) in \( V \) that is expanded to
multiple vertices in \( V^* \), we denote \( n_v \) as the original vertex \( n \in V \) corresponding to
the expanded vertex \( v \in V^* \). In addition, as the expansion retains the original edges
and adds new edges, we have \( E \subset E^* \). Let \( V^*_n \) denote the set of expanded vertices in
\( V^* \) corresponding to vertex \( n \in V \).
5.4.4 Constructing augmenting cycles/paths

Given a three-connected cubic graph and a root vertex $r$, we decompose the network into a sequence of paths. The first path is a cycle that starts and ends at vertex $r$. Every other path starts and ends at distinct vertices. The cycle and the paths satisfy the following properties.

1. The removal of vertices in the path keeps every other vertex not added in this path or earlier paths connected with each other.

2. Every vertex in the path is connected to at least one vertex that is not added this path or earlier paths.

The first condition implies the *non-separating* nature of the path. The second condition may be met in several ways. The algorithm in (49) simply decomposes the network into a sequence of non-separating cycle and paths that also satisfy the second property above in a specific manner\(^3\). For the sake of completeness and ease of understanding of the rest of the chapter, we briefly discuss the path augmentation technique employed by (49) to construct three vertex independent trees rooted at $r$, but tailored to the cubic graph expansion. It is important to note that we are not interested in computing three vertex independent trees. We obtain the cycle and paths from (49) and modify these paths to obtain cycle/paths in the original graph.

Figure 5.3 shows the steps involved in the path augmentation procedure. We pick an arbitrary neighbor of $r$, say $u$, and remove the edge $r-u$. We then decompose this graph into a sequential ordering of paths. The first path is a cycle as it will start and end at $r$. At every stage $i$, the path $P_i$ starts and ends at two distinct vertices that are already part of some earlier paths and traverses at least one new vertex. The last path augmented will have $u$ as the only new vertex added.

\(^3\)The algorithm in (49) merges all the nodes that have been added in the paths thus far into one node, say $v$. A non-separating induced cycle is then obtained through $v$. Since an induced cycle does not have any chord, any new node that is added is guaranteed to be connected to at least one node that is not added yet. However, computing an induced cycle is not a necessary condition for satisfying the second property.
1. Select an arbitrary neighbor of $r$, say $u$. Remove the edge $r-u$.

2. Compute a cycle starting and ending at $r$ and avoiding $u$ such that: (1) the removal of the cycle keeps every other vertex not in the cycle connected to $u$; and (2) every vertex in the cycle has one neighbor that is not added to the cycle.

3. $i = 0$.

4. If $\bigcup_{j=0}^{i} P_j$ does not include all vertices in $V^*$, do:
   
   (a) $i \leftarrow i + 1$.
   
   (b) Compute a path $P_i$ avoiding $u$, i.e.: (1) every vertex in the path $P_i$ has one neighbor in $Q \setminus \bigcup_{j=0}^{i} P_j$; and (2) every vertex in $Q \setminus \bigcup_{j=0}^{i} P_j$ remains connected to $u$.

   (c) Repeat.

5. Stop.

Figure 5.3: Algorithm to construct a sequence of paths in a three-connected cubic graph.

**Definition - Path index of a link:** The path index of a link is the index of the augmenting path/cycle in which the link is added in $Q$.

In order to guarantee disjoint paths, it is a standard practice to maintain a global order or partial order between vertices during the path augmentation phase (49; 45; 46). This ordering is used to decide the neighbors on each tree for every vertex.

Let the first cycle $P_0$ consist of the vertices $r, v_1, v_2, \ldots, v_k, r$. Let the red chain be $r \leftarrow v_1 \leftarrow v_2 \ldots \leftarrow v_k$. The partial order maintained on the red tree is denoted by $r \prec v_1 \prec v_2 \ldots \prec v_k$. Let path $P_i$ consist of vertices $x, v_1, v_2, \ldots, v_k, y$ where $x, y$ are vertices that were added earlier to the trees and vertices $v_1, v_2, \ldots, v_k$ are not present in any path $P_j; j < i$. Then, if $x \not\prec y$ in the partial order, the red chain is $x \leftarrow v_1 \leftarrow v_2 \ldots \leftarrow v_k$.

We refer an interested reader to Section 5.5.1 for an illustration of the construction of the paths on the cubic graph on an example graph.
5.4.5 Computing segments

The sequence of cycle/paths in $Q$ ensures that $u$ is not added until the last path and that all vertices that have not been added are connected to $u$. This is done so that all green paths eventually reach $u$ and then $r$ through the removed edge. It is easier to visualize this when the green tree is imagined to be grown downward rooted at $u$ and then, adding the missing edge $r - u$ makes it rooted at $r$ as required.

However, in $G$, it is not guaranteed that $n_u$ is added in the last path since the paths may only be edge-disjoint. Hence it is possible for $n_u$ to be added much earlier. However at any stage, all vertices would still have connectivity to $n_u$ through edges since the green path has to go through $n_u$ to reach $r$. Therefore, we assume that $n_u$ to be not added to a path, just for sake of maintaining the non-separating notion. We define the portion of a path in $Q$ that corresponds to a path/cycle on $G$ as a segment. A path in $Q$ may result in multiple paths/cycles in $G$.

Recall that $n_v$ denotes vertex $n$ in $G$ to which $v$ (in $Q$) belongs. A few other notations required to understand the segment computation are as follows. $\text{next}(v)$ is the vertex adjacent to $v$ along the direction of traversal in the path. $l(n_v)$ denotes the last vertex present in the same path as $v$ and also belongs in $V_n^*$ along the direction of traversal.

Figure 5.4 shows the procedure to compute a segment from path. We begin with the first cycle on $Q$. All vertices in the cycle except the root($r$) are unmarked to begin with. Then, $v$ in the procedure is simply $r$ itself since $\text{next}(r)$ is guaranteed to be unmarked. We traverse the cycle as shown in the procedure until we find a segment that would end at $r$. Once a segment is computed, for each $n_v$ in the segment, we mark all vertices $v \in V_n^*$ in $Q$ before we continue to extract any more segments. The procedure might leave out several portions of the path in $Q$. We repeat the procedure iteratively to find segments on the residual sub-paths. Every time we get a segment, we add it in $G$. Once we process all such segments, we repeat the entire process for the rest of the paths one at a time until all of them in $Q$ are exhausted. The above computation results in a sequential list of segments in
\(G\). We now illustrate this process with an example before describing how the trees are constructed from the segments.

**Procedure Compute Segment**

**Input:** A path in \(Q\): \((x, v_1, v_2, \ldots, v_k, y)\), where \(x\) and \(y\) are marked.

For the first cycle: \(x = y = r\).

**Output:** A segment \(S\) – a path or cycle in \(G\).

1. \(S = \emptyset\)

2. Traverse the path starting from \(x\) and identify the first vertex \(v\) such that \(v\) is marked and \(\text{next}(v)\) is unmarked.

3. If such a vertex is found, add \(n_v\) to \(S\). Otherwise, go to Step 8.

4. \(v' := \text{next}(v)\).

5. Add \(n_{v'}\) to \(S\).

6. If \(v'\) is marked, go to Step 8.

7. \(v = l(n_{v'})\) go to Step 4.

8. For each \(n_v\) in \(S\), mark all the vertices \(v \in V_n^*\) in \(Q\).

9. Stop.

---

**Figure 5.4:** Algorithm to compute a segment in a given portion of the path.

**Example:** Figure 5.5 shows a sample path to illustrate how the segments are computed. \(x\) and \(y\) are marked to begin with. \(z\) is some vertex in the path whose \(n_z\) has been added in \(G\) previously and hence \(z\) is marked. We start from \(s_1\) and skip to \(l(s)\) which is \(s_3\) and continue along the path to reach \(b\) and then reach \(t_1\). Now, we skip to \(l(t)\) which is \(t_3\). Finally we stop at \(y\) since it is marked. This would comprise of a segment \((x - s - b - t - y)\). We add this segment in \(G\) and mark \(s_1\) through \(s_{d-2}\) and \(t_1\) through \(t_{d'-2}\) in \(Q\). This leaves us with two portions of the path at this stage. We come back and compute \(s_1\), \(a\) and \(z\) and halt since \(z\) was marked at some earlier segment. Hence, we would get \((s - a - z)\) as the next segment. In this fashion the rest of path is processed to compute segments.

**Segments and their properties:** Let the segments obtained from the paths \(d, d'\) be the degrees of \(s, t\) respectively in \(G\).
in $\mathcal{Q}$ be numbered from 0 to $(S - 1)$. The $i^{th}$ segment is denoted by $S_i$.

The segments have the following properties:

1. Segment $S_0$ is a cycle that starts and ends at $r$.

2. At every stage $i \geq 1$, the segment $S_i$ starts and ends at two (not necessarily distinct) vertices that are already part of earlier segments and traverses only new vertices (at least one).

3. For each $i \geq 0$, the following property holds: Consider the graph obtained by removing all the edges in $\bigcup_{j=0}^{i} S_j$. The vertices and all their other edges are retained. In this graph, all vertices in $\mathcal{V} \setminus \bigcup_{j=0}^{\max(0,i-1)} S_j$ remain connected to vertex $u$.

4. $\bigcup_{j=0}^{S-1} S_j$ contains all vertices in $\mathcal{G}$.

We refer an interested reader to Section 5.5.1 for an illustration of the construction of the segments from the paths on the cubic graph on an example graph.

The segment computation described in this section is made possible because of our cubic graph expansion technique. Other approaches (63), (48) achieve three edge to three vertex graph transformations by converting edges to vertices. The presence of edges as vertices makes it very difficult to handle the partial order across various sub-vertices while trying to construct the paths on $\mathcal{G}$. This is specifically avoided in our cubic graph expansion where all the edges in the original graph are retained as edges in the expanded graph. Further, retaining a cubic three-vertex connected graph also greatly simplifies the construction of the paths and cycles described in Section 5.4.4 as opposed to the more involved scenarios that one may encounter in arbitrary three-vertex connected graphs. An interested reader can find examples of such scenarios in Section 3.3 in (49).
Procedure Red and Blue trees from segments

1. Initialize $T_r$ and $T_b$ to be empty.

2. Let the first cycle $S_0$ consist of the vertices $r, n_1, n_2, \ldots, n_k, r$. We let the the red chain be $r \leftarrow n_1 \leftarrow n_2 \ldots \leftarrow n_k$ and the blue chain be $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_k \rightarrow r$. These chains are added to $T_r, T_b$ respectively.

The partial order maintained is denoted by $[r, n_1] \prec [n_1, n_2] \prec \cdots \prec [n_k-1, n_k] \prec [n_k, r]$.

3. For each $1 \leq i \leq S - 1$ in increasing order:
   
   (a) Let segment $S_i$ consist of vertices $x, n_1, n_2, \ldots, n_k, y$ where $x, y$ are vertices that were added in earlier segments and vertices $n_1, n_2, \ldots, n_k$ are not present in any segment $S_j$; $j < i$. For any vertex $v$, let $v_R$ and $v_B$ denote the parent on the red and blue trees, respectively. Note that $x_R, x_B, y_R, y_B$ are already computed and part of some $S_j$; $j < i$.

   (b) Then, if $[x, x_R] \not\succ [y, y_R]$ in the partial order, the red chain is $x \leftarrow n_1 \leftarrow n_2 \ldots \leftarrow n_k$ and the blue chain is $n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_k \rightarrow y$. The red and blue chains are added to $T_r, T_b$ respectively. The partial order for these edges are updated as $[x_R, x] \prec [n_1, n_2] \prec [n_2, n_3] \prec \cdots \prec [n_{k-1}, n_k] \prec [n_k, y] \prec [y, y_B]$. Note that $x$ and $y$ are the same vertex, this condition always holds.

   (c) If $[x, x_R] \succ [y, y_R]$, the chain is reversed and the partial order is $[y_R, y] \prec [y, n_k] \prec [n_{k-1}, n_{k-2}] \prec \cdots \prec [n_2, n_1] \prec [n_1, x] \prec [x, x_B]$.

---

Figure 5.6: Procedure to construct the red and blue trees rooted at destination $d$ in $G$

5.4.6 Computing red, blue, and green trees

From the sequential ordering of segments computed, we can construct the three edge independent trees. The red and blue trees are computed using the path augmentation approach (45; 46), where the sequence of segments are used for augmentation. In order to construct the red and blue trees, a global order (45) or partial order (46) among the edges in the graph is maintained to ensure the disjointedness of paths in two trees. We outline the construction of red and blue trees from the segments in Figure 5.6. It is important to maintain an order on the edges since we could have segments starting and ending at the same vertex (cycles) in which case maintaining a partial order on the vertices is not sufficient.
Computing green neighbors  Every node chooses the link with the highest path index as its green neighbor. For vertex $n_u$, we choose $r$ as the green neighbor.

We refer an interested reader to Section 5.5.1 for an illustration of the construction of the three trees from the segments on an example graph.

5.4.7 Correctness and complexity

Please see Theorem 12 in the Appendix for the proof of correctness. The computational complexity of constructing the three edge independent spanning trees is $O(V^2)$. A detailed complexity analysis is available in the Appendix.

5.5 Appendix

**Theorem 11** The expansion procedure retains the three-edge-connectivity property of the graph and eventually results in a three-vertex connected graph.

**Proof** Note that as the original graph is two-vertex connected, removal of vertex $n$, keeps $G_{\text{int}} \setminus \{n\}$ connected. After an expansion at any stage, the graph still retains the $2V$ property. This can be checked by seeing that the removal of any vertex keeps the graph connected. Thus, it is sufficient to show that the algorithm retains three-edge connectivity at every vertex expansion stage. We show that after every vertex expansion, the removal of any edge will still leave the graph two-edge-connected.

We define a property called overlap on the edges connected to the expanded vertices of $n$. Consider only the leaf components in $G_{\text{int}} \setminus \{n\}$. Divide these components into two arbitrary groups, say $D_1$ and $D_2$. The smallest index of the expanded vertex $n$ that $D_i$ is connected to is referred as $min_i$ and the largest index of the expanded vertex $n$ that $D_i$ is connected to is referred as $max_i$. We define the interval of connection of the two groups as $(min_1, max_1)$ and $(min_2, max_2)$. We say that $D_1$ and $D_2$ overlap if these two intervals overlap (or one is completely contained in the other). If the two intervals are disjoint, then we say that the two groups do not overlap.
Let \( I_n \) denote the edges that connect two sub-vertices of \( n \). Consider an edge \( \ell \).

We have two possible scenarios: (1) \( \ell \not\in I_n \) or (2) \( \ell \in I_n \). In both scenarios, we show that the graph after edge removal is two-edge-connected.

Case 1: Since edge \( \ell \not\in I_n \), we consider three sub-cases. First, let \( \ell \) be within one of the two-edge-connected components \( C_i \). Before \( n \) is expanded, we have a 3E-2V connected graph. Any vertex \( v \) in \( C_i \) has three link disjoint paths to \( n \). Now removal of \( \ell \) in \( C_i \) can affect at most one of these three disjoint paths. Hence \( v \) still has at least two edge-disjoint paths to \( n \). Therefore, \( v \) has two edge-disjoint paths to any \( n_i \) after expansion since \( n_1 \) to \( n_{d-2} \) is a chain of newly added edges. This argument may be used for any vertex in \( G_{int} \setminus \{n\} \setminus \{\ell\} \). Since the existence of edge-disjoint paths is transitive, there are two edge-disjoint paths between any two vertices and hence the graph remains two-edge connected.

Second, consider the scenario that \( \ell \) connects some leaf component to an expanded vertex of \( n \). As every leaf component has at least two edges connecting to the expanded vertices in \( n \), there is still another edge that connects the component to \( n \) and hence the graph remains two-edge connected. If the edge connects a non-leaf component, then this component is connected to at least two other components. By following two of its neighbors, we will arrive at two leaf components, both of which have connectivity to \( n \). Hence, two-edge connectivity is retained.

And finally, let edge \( \ell \) be an edge that connects two components and its removal disconnects \( G_{int} \setminus \{n\} \setminus \{\ell\} \). Let the two disconnected components be denoted by \( D_1 \) and \( D_2 \). Observe that components \( C_1 \) and \( C_2 \) are both present in the same \( D_i \) or each is a part of a distinct \( D_i \). In either case, the edges connecting from \( D_1 \) to the expanded vertices in \( n \) and from \( D_2 \) to the expanded vertices in \( n \) will overlap. Hence, the resultant graph is two-edge-connected.

Case 2: If \( \ell \in I_n \), the removal of \( \ell \) will result in two distinct path segments. While one path segment will contain \( n_1 \), the other would contain \( n_{d-2} \). Figure 5.7 shows the structure of the expanded graph when edge \( \ell \) is removed. Consider the path from component \( C_1 \) to \( C_2 \) (the components that each had connections to \( n_1 \) and \( n_{d-2} \)) through the other components. Observe that this path through the components
forms two cycles, one with the path segment containing $n_1$ and the other with the path segment containing $n_{d-2}$. Thus, this path segment along with the expanded vertices of $n$ is two-edge-connected. In addition, as any other leaf component other than $C_1$ and $C_2$ has at least two edges connecting to $n$, every other leaf component has connectivity to at least one path segment, if not both. Thus, the entire expanded structure after the removal of edge $\ell$ still remains two-edge-connected.

Thus, the graph remains two-edge-connected after the removal of any edge. Therefore, the expanded graph is three-edge-connected. □

**Lemma 5** Any three-edge-connected cubic graph is also three-vertex connected.

**Proof** We prove the lemma by contradiction. Consider two vertices $n$ and $n'$. There exists three edge-disjoint paths between the two vertices. Assume that two of these have a vertex in common. Thus, the common vertex must have degree four, hence contradicts the assumption of a cubic graph. □

**Theorem 12** The trees obtained in $\mathcal{G}$ are edge independent.

**Proof** It is fairly straight-forward to see that from any vertex: (1) the red and green paths are edge-disjoint; and (2) the blue and green paths are edge disjoint. Let $p_\ell$ denote the path index of link $\ell$. Observe that while the red and blue paths will traverse edges with non-increasing values of $p_\ell$, the green path will follow a strictly increasing (at every hop) value of $p_\ell$. Thus, the red and blue paths are mutually disjoint with the green path.
It remains to be shown that the red and blue paths themselves are edge independent. This fact is readily seen by observing that every path that is augmented in $Q$ is the equivalent of one or more successive augmentations in $G$. The path augmentation is the same as that employed for computing two edge independent trees (46). Thus, the proof employed to show the edge-disjointedness of the red and blue paths (see Theorem 9 in (46)) using a partial order on the edges may be applied in a straightforward manner here. □

**Complexity Analysis:** Computing a minimally three-edge connected graph may be achieved in two steps. First, we compute a three-edge connected sparse spanning subgraph of $G$ (47). The number of edges in the sparse graph is guaranteed to be at most $3V-6$. Second, we reduce the sparse spanning subgraph to a minimally three edge connected graph. We consider one edge at a time and check if the edge may be removed without affecting the three edge connectivity of the spanning subgraph, which may be achieved in $O(V)$ time for every edge (48). As the number of edges in the sparse subgraph is $O(V)$, the minimally three edge connected graph is obtained in $O(V^2)$ time.

The complexity of the cubic expansion procedure is $O(VE)$. For every vertex, we compute the depth-first-search (DFS) numbering after removing the vertex to identify the two-edge-connected components. DFS numbering has a complexity $O(E)$, which is performed for every vertex. Note that the number of edges in a minimally three-edge-connected graph is linear in the number of vertices, hence the complexity is only $O(V^2)$.

The complexity of constructing the paths in $Q$ is $O(V^*E^*)$ (49). The number of edges in $Q$ is linear ($3/2$) in the number of vertices, hence the complexity is $O(V^{*2})$. The number of vertices in the cubic graph $V^*$ is bounded by $(V+(3V-6))$ by virtue of the cubic expansion procedure on the minimally three-edge connected graph. Hence, the complexity of computing the paths in $Q$ is $O(V^2)$.

Our procedure to segment the paths only consists of traversing each edge in the paths in $Q$ one at a time. Hence it takes only $O(E^*)$. Using the discussion above, this takes $O(V)$. 

Therefore the overall time complexity to compute three edge independent spanning trees is $O(V^2)$.

5.5.1 A complete example for algorithm illustration

We will use an example graph to illustrate the working of the various stages of the entire algorithm to construct three-edge independent spanning trees.

Example network and its cubic expansion

Figure 5.8 shows the example three-edge and two-vertex connected graph, $G$ and its expansion to a three-edge-connected cubic graph, $Q$ according to the cubic expansion algorithm described in Figure 5.1.

Figure 5.8: An example two-vertex and minimally three-edge-connected graph and its expansion to a three-edge connected cubic graph.

Figure 5.9: Cycles/Paths on the expanded graph.

\[ P_0 : B - C - G - F - B \]
\[ P_1 : G - H - K - A_2 - A_3 - F \]
\[ P_2 : C - A_4 - A_3 \]
\[ P_3 : A_4 - E - K \]
\[ P_4 : E - D - H \]
\[ P_5 : D - A_1 - A_2 \]
Non-separating paths computed on the cubic graph

Figure 5.9 shows the paths for the expanded example graph shown in Figure 5.8(b). We assume that vertex $B$ is the root and vertex $A_1$ is the vertex to be avoided. $P_0$ represents the cycle, while $P_1$ through $P_5$ are paths. In the example, when the first cycle is added, the partial order would be $B < C < G < F$. When $P_1$ is added, since $G < F$ in the order, $G$ becomes the red end of the chain. In this fashion, all paths are added to the partial order.

If our goal is to construct three vertex independent trees, we may follow the above method of maintaining a partial order. Since we are interested in computing three edge independent trees, computing the vertex independent trees and then selecting the forwarding edges from one (or more) of the expanded vertices will not work. The reason is that several vertices in $Q$ correspond to the same vertex in $G$. While the paths are augmented in $Q$ and the partial order is updated, each vertex is treated independently. This causes multiple definitions of precedence in the partial order when we go back to $G$. We illustrate the problem using one such approach.

Consider the vertices $A_1$, $A_2$, $A_3$, $A_4$. On the cubic graph, the red neighbors of $A_1$, $A_2$, $A_3$, $A_4$ are $D$, $K$, $A_2$, $C$, respectively. We have to decide on a red neighbor for vertex $A$. Since $A_2$ was added first, it is natural to pick the red neighbor as $K$. In that case, the red and blue paths of $E$ would be $E \rightarrow A \rightarrow K \rightarrow H \rightarrow G \rightarrow C \rightarrow B$ and $E \rightarrow K \rightarrow A \rightarrow F \rightarrow B$ respectively. Thus the paths are not edge-disjoint as $A - K$ is traversed by both the paths. This is the same problem that occurs in the construction mechanism described in (63).

Computing Segments

We now show the segments being computed from the paths constructed on the cubic expansion of the example graph in Figure 5.9. The root vertex $r$ is at vertex $B$ and $u$ is vertex $A$ in this example. Since $P_0$ (cycle) does not have any sub-vertices, the segment to be added is the same as the path. In $P_1$, the segment is $G \rightarrow H \rightarrow K \rightarrow A \rightarrow F$.

---

5This order defines the red neighbors. The red neighbor of $F$ is $G$, that of $G$ is $C$, and so on.
This is obtained by computing $G - H - K - A2 \rightarrow A3 - F$; $l(A) = A3$. The remaining portion in the path is $A2 - A3$ and there are no unmarked vertices to add. There is no segment in $P_2$ since $A4$ is already marked. In $P_3$, the segment is $A - E - K$. Finally, the segment $E - D - H$ is added.

The set of segments added in $G$ are shown in Figure 5.10. The solid lines denote the edges that are part of the segments. The dotted lines indicate other edges in $G$ but not in the set of segments. In particular, the edge $A - B$ is the edge that was removed between $r - u$.

![Figure 5.10: The list of segments on the example graph.](image)

**Computing the independent trees**

We now illustrate how the segments are processed on the example graph in Figure 5.11. When the first cycle is added, the red chain is $B \leftarrow C \leftarrow G \leftarrow F$. The partial order is $[B, C] \prec [C, G] \prec [G, F] \prec [F, B]$. For $S_1$, since $[C, G] \prec [F, G]$ in the order, $G$ becomes the red end of the chain. The partial order among the newly added edges is $[C, G] \prec [G, H] \prec [H, K] \prec [K, A] \prec [A, F] \prec [F, B]$. For $S_2$, since $[H, K] \prec [A, K]$, the red neighbor of $E$ is $K$. The partial order for the edges is $[H, K] \prec [K, E] \prec [E, A] \prec [A, F]$. Finally for $S_3$, since $[G, H] \prec [K, E]$, $H$ becomes the red end of the chain. This defines the red and blue neighbors for all vertices. For the green neighbors, vertices that have degree 3 in $G$ have only one neighboring vertex which is guaranteed to have been added in a higher path index and that becomes the green neighbor. This includes $C, G, F, H, K, E$. For vertex $D$,

---

*Because $\prec$ is transitive.*
since $A_1$ is a neighbor in $Q$ and was added last, that becomes the green neighbor. For vertex $A$ which is $n_u$, the green neighbor is $r$ which is $B$. The red, blue and green neighbors for all vertices are shown in Figure 5.11.


Figure 5.11: The neighbors defined on the example graph

The three edge independent spanning trees obtained for our example graph are redrawn in Figure 5.12 with the original graph.

![Diagram showing example graph and three edge independent trees rooted at vertex B: (a) Example graph, (b) Red tree, (c) Blue tree, (d) Green tree.]

Figure 5.12: Example graph and three edge independent trees rooted at vertex B.
CHAPTER 6

MULTIPATH ROUTING AND FAST RECOVERY IN IP NETWORKS

In this chapter, we develop an approach for disjoint multipath routing and fast recovery in IP networks that guarantees recovery from arbitrary two link failures. We employ three link-independent trees, referred to as red, blue, and green trees, rooted at every destination. The path from a source to the destination on the trees are mutually link-disjoint. The routing of packets is based on the destination address and the input interface over which the packet was received. We discuss different ways of employing the three link-independent trees for multipath routing and/or failure recovery. If the trees are employed exclusively for multipath routing, then no packet overhead is required. If the trees are employed for failure recovery, then the overhead bits will range from 0 to 2 bits depending on the flexibility sought in routing. We evaluate the performance of the trees in fast recovery by comparing the path lengths provided under single and dual link failures with an earlier approach based on tunneling.

6.1 Organization

The rest of the chapter is organized as follows. Section 6.2 describes the prior work. Section 6.3 discusses the network model. Section 6.4 describes ways in which the three trees may be employed for multipath routing and failure recovery. Section 6.5 describes the experimental setup and performance evaluation.
6.2 Prior Work

Although there have been several works in multipath routing, the only popular multipath routing employed in the Internet today is equal-cost multiple paths (ECMP) [RFC 2991, RFC 2992]. ECMP might not offer an advantage in terms of improving bandwidth for multipath since the paths are not guaranteed to be disjoint. The use of two independent trees, in which paths from nodes to the root(destination) are guaranteed to be disjoint has been suggested (60) for reliability, in reliable multicasting (68), for disjoint multipath routing to a single destination (69) or a multi-homed network (70; 71).

Among the solutions that are employed for fast recovery are: (1) Equal cost multi-path (ECMP); (2) Using MPLS tunnels or multi-hop repair paths for routing around failed links (72; 73); (3) Fast re-routing framework for IP networks (74); (4) multiple routing configurations (75; 76); (5) failure insensitive routing (FIR) (77; 78); and (6) tunneling using not-via addresses (79; 80). For a detailed description of the above approaches, their comparison, and additional references on similar approaches, the reader is referred to (81). All the above works have one or both of the following two limitations: (i) the number of protection/alias addresses (or equivalently the number of auxiliary graphs) is not constant; and/or (ii) they do not guarantee recovery except in the case of single link failures. In (4), the authors develop a mechanism for identifying backup ports for any given primary tree to tolerate single link and node failures. While the link failure may be tolerated with small increase in path lengths, recovery from node failures incur significant overhead as the recovery is performed assuming a sequence of link failures.

To the best of our knowledge, the only prior approach that guarantees recovery from two link failures is by Kini et al. (81). The authors develop a fast recovery method employing IP-in-IP tunneling. Every node is assigned one primary address and up to three protection addresses. Every node is also associated with up to three auxiliary graphs, corresponding to the three protection addresses. The auxiliary

---

1A good number of them do not provide guarantees even for single link failure recovery.
graphs for node $x$ are constructed such that: (i) every link attached to node $x$ is not present in at least one of the auxiliary graphs; and (ii) each auxiliary graph is two-edge-connected. Within each auxiliary graph, they construct two colored trees rooted at node $x$ (addressed using its protection address). Upon the first failure, the packet is tunneled to the other end of the link using one of the two colored trees. If it encounters a second failure en route, they switch the tree and reach $x$. Their approach requires up to seven routing table entries per node (1 default + 6 for protection). Also, it is not possible to split the traffic as recovery is designed to be on a link level.

In this chapter, we show how to employ three independent (colored) trees for handling arbitrary two link failures and/or for multipath routing.

6.3 Network Model

We consider a network denoted by $\mathcal{G}(\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ and $\mathcal{L}$ denote the set of nodes and links, respectively. Every link $\ell \in \mathcal{L}$ is assumed to be bi-directional and when the link fails, it is assumed that both directions are affected. As three-edge-connectivity is required to tolerate any two arbitrary link failures, we assume that the network will be three-edge-connected. We assume that the network employs link-state protocol, hence all nodes in the network are aware of the network topology.

6.4 Routing with Three Link-Independent Trees

6.4.1 Multipath Routing

The three link-independent trees may be employed simultaneously to achieve disjoint multipath routing in networks. The ingress routers may map the packets from a flow (such as a TCP flow) to any of the three paths. If a flow is destined to port number $p$ of a destination, then the flow may be routed on the red, blue, or green path if $p \mod 3$ is 0, 1, or 2, respectively. The transport/application layers may take advantage of the characteristics of the routing offered by IP by establishing multiple sub-flows to the same destination over consecutive port numbers to improve end-
to-end throughput. Observe that as the packets still follow the same path once mapped to a specific tree, packets transmitted over a specific tree will still arrive in order at the destination. A sub-layer (either above transport layer or in application layer) may implement splitting of traffic across different flows established to different destination port numbers and merging them at the destination.

6.4.2 Fast Re-Routing

The three link-independent trees may be employed in several different ways in practice depending on the default routing option used in the network. We highlight three specific approaches in this chapter and discuss the overhead involved. The approaches are classified based on how the network performs routing when there are no failures. In all these approaches, it is assumed that the forwarding decision at a node would be made using the destination address in the packet header, the input interface over which the packet was received, and any other overhead bits that may be required by a specific approach.

**Red Tree First Approach**

This approach restricts the tree over which the packets are routed by default. We assume that all packets will be routed on the red tree by default. Every packet carries a one-bit overhead bit (SF) which indicates if the packet has seen a second failure or not. Note that the forwarding decision at every node is based on the destination address and the incoming interface (directed edge). For a given destination, an incoming interface is colored uniquely. Thus, the incoming interface indicates whether the packet is being forwarded on the red/blue/green tree. Thus, if a packet is received on an incoming edge that is present in the red tree, it is assumed that the packet has seen no failures. We ignore the SF bit and route the packet on the red tree.

*One link failure:* Consider a link $x-y$, where $x\rightarrow y$ is on the red tree. If link $x-y$ fails, node $x$ will reroute the packet on either the blue or the green tree. The tree
on which the packet is rerouted is determined by the color of the edge $y \rightarrow x$. Thus, if $y \rightarrow x$ is blue, the packet will be re-routed on the blue tree. If $y \rightarrow x$ is green, the packet will be rerouted on the green tree. In both these scenarios, when the packet is rerouted from the red to blue/green tree, the SF bit is set to 0.

**Two link failures:** A node that receives a packet over the blue (green) incoming interface would forward the packet over the blue (green) tree. If the blue (green) forwarding link is not available and the SF bit in the packet is set to 0, this implies that the packet has seen one failure. Hence, the packet is rerouted to the green (blue) tree with the SF bit set to 1 (indicating that the packet has encountered the second failure). If the blue (green) forwarding link is not available and the SF bit in the packet is set to 1, the packet is dropped.

![Figure 6.1: Example network illustrating the Red Tree First approach](image)

Consider the example shown in Figure 6.1. Assume that links 2–3 and 3–4 have failed. When a packet that has not seen a failure arrives at node 4, it is to be forwarded on link 4–3 on the red tree. Since the link has failed, the packet must be rerouted on the green tree (the reverse color). If we use the blue tree instead, the packet will have to be dropped at node 3 as both the blue and green forwarding edges have failed. By using the reverse color, we ensure that the first failed link is avoided when the packet is rerouted after the second failure.
Any Tree First Approach

This approach allows the source to choose the default tree over which a packet is transmitted. This choice of the initial tree may be used by different sources in different ways. For example, it is possible that to reach a specific destination, the red tree may provide the shortest path for node $n$ while the blue tree may provide the shortest path for node $n'$. Thus, $n$ and $n'$ may transmit their packets on two different trees. Alternately, a node may choose to spread its traffic over the three trees by randomly choosing a tree for a packet. Such an approach improves security against eavesdropping on a link or at most two links. In this approach, every packet carries a two-bit overhead which encodes whether the packet has seen a failure or not; and if it has seen a failure, some information about the failed link. Upon the first failure, we still have to switch to the tree indicated by the reverse color. The four possible values of the overhead bits indicate the following: 00 - packet has not seen a failure; 01/10/11 - packet has seen failure of a link which does not have the color red/blue/green, respectively. The non-zero value of the two-bit overhead indicates the color of the tree to take when the second failure occurs. Thus, if a packet is received over a link whose color is not the same as that indicated in the two-bit field, then the packet has seen one failure thus far. If the packet is received over a link whose color matches the value in the two-bit field, it indicates that the packet has seen two failures. Thus, when another link failure is encountered, the packet is dropped.

If a reverse color does not exist on the first failed link encountered by the packet, we may assume any of the other two colors to be the reverse color.

Flexible Routing Approach

The default routing technique employed when a packet has not encountered any failures is left to the choice of the network. For instance, the network may choose shortest path routing. When failures occur, the routing is handled in a way similar to the any tree first approach. Thus, only two overhead bits are required.
6.5 Performance Evaluation

We evaluate the performance of the developed routing schemes through simulations. We use a home-grown C program to implement the routing algorithms. We also compare the performance of our scheme against the routing scheme employed in (81). We consider five networks, as shown in Figure 7.10: (a) ARPANET; (b) NSFNET\textsuperscript{2}; (c) Node-16; (d) Node-28; and (e) Mesh-4x4. The Node-16 and Node-28 networks are hypothetical \textit{minimally} 3-connected networks such that all nodes have exactly three links connected to them.

We implement three routing approaches. The Red Tree First (RTF), Shortest Tree First (STF) and Shortest Path First (SPF). The second approach falls under the Any Tree First while the last one is a Flexible Routing approach. We compare our SPF with Tunneling. In that way, both approaches will use the shortest path until a packet sees the first failure. After a packet sees a failure the two schemes handle it very differently.

In each of the four routing schemes, we are interested in computing the average path length under no failure scenario, when one link in the network fails, and when two links fail in the network. We describe the computation of the metric in detail in the following subsection.\textsuperscript{3}. For all three of our routing approaches, we pick a destination node $d$ and construct trees rooted at $d$.

6.5.1 No Failures

We compute the average path lengths in the network when there are no failures. This default path length from node $n$ to the destination $d$ under the specific routing approach is computed as:

\[ P_{nd0} = L_{RTF,n,d} \]

Then, the average path length under no failures in the network is given by

\textsuperscript{2}The NSFNET network considered here has been modified from the original network by adding link NE–GA to keep the network three-edge-connected.

\textsuperscript{3}Note that all four schemes may support more than two failures but only recovery from two link failures is guaranteed.
Figure 6.2: Networks considered for performance evaluation.

\[ P_0 = \frac{1}{|\mathcal{N}|} \frac{1}{(|\mathcal{N}| - 1)} \sum_{d \in \mathcal{N}} \sum_{n \in \mathcal{N} \setminus d} P_{nd0} \]

This metric is computed for all four routing approaches.

6.5.2 One Link Failures

We are only interested in computing the average path length when a link failure affects the default path. All other single link failure scenarios do not affect the routing schemes.

\[ P_{nd1} = \frac{1}{L_{RTF,n,d}} \sum_{\forall f} (L_{RTF,n,f} + L_{*,f,d}) \]

is the path length from node \( n \) to \( d \) averaged over all link failures that affect the default path. Note that the denominator in the expression for \( P_{nd1} \) signifies the number of failure scenarios that affect the default path. \( f \) denotes the node at which the default forwarding link has failed and \( L_{*,f,d} \) denotes the backup path length depending on the routing scheme used. The other end of the failed link is denoted as \( g \). Then,
1. Using Three trees: \( L_{*,f,d} = \) backup path length on the tree specified by the reverse link \((g \rightarrow f)\) color

2. Tunneling: \( L_{*,f,d} = L_{Tunnel,f,g} + L_{SPF,g,d} \)

Then, the average path length under one failure in the network is given by,

\[
P_1 = \frac{1}{|N|} \frac{1}{(|N| - 1)} \sum_{d \in N} \sum_{n \in N \setminus d} P_{nd1}
\]

We compute this metric for all four routing approaches.

6.5.3 Two Link Failures

The tunneling approach and our approach handle failures differently, the former is a link-level recovery while the trees directly reroute towards the destination.

Hence, it is not possible to compare a particular two link failure scenario across the schemes as a two link failure encountered when tunneling may not seen when the trees are employed and vice-versa. To this end, we consider all possible two link failure scenarios \((l_1, l_2)\) in the network such that at least one of these two links affect the default path from node \(n\) to \(d\). By doing this, we ensure that both approaches are affected by at least one failure. The other failed link may or may not appear in the respective backup paths. The total number of such failure scenarios is denoted as \(f.s_{count}\).

Without loss of generality, assume that \(l_1\) is present in the default path. \(f\) denotes the node at which the default forwarding link has failed. Let the node on the other end of the link be \(g\). \(L_{*,f,d}\) denotes the backup length depending on the routing scheme used. Let \(f2\) denote the node at which the default forwarding link \((l_2)\) has failed. Let the node on the other end of the link be \(g2\). Also, let the reverse colors on the two links \(l_1, l_2\) be denoted by \(C_1, C_2\) respectively. Then,

\[
P_{nd2} = \frac{1}{f.s_{count}} \sum_{v_f} (L_{SPF,n,f} + L_{*,f,d})
\]

1. Using Three trees: \( L_{*,f,d} = L_{C_1,f,f2} + L_{C_2,f2,d} \)
2. Tunneling:

(a) If \((l_2 \in \text{Path}_{\text{Tunnel},f,g})\)

\[
L_{*,f,d} = L_{\text{Tunnel},f,f_2} + L_{R/B,f_2,g} + L_{\text{SPF},g,d}
\]

(b) else if \((l_2 \in \text{Path}_{\text{SPF},g,d})\)

\[
L_{*,f,d} = L_{\text{Tunnel},f,g} + L_{\text{SPF},g,f_2} + L_{\text{Tunnel},f_2,g_2} + L_{\text{SPF},g_2,d}
\]

Note that the second failed link may not be encountered in either scheme. In that case, \(L_{*,f,d}\) is computed as in the single link failure section. The above metric \(P_{nd2}\) is the backup path length from node \(n\) to \(d\) under a two link failure. The average path length over all two link failures is computed as

\[
P_2 = \frac{1}{|\mathcal{N}|} \frac{1}{(|\mathcal{N}| - 1)} \sum_{d \in \mathcal{N}} \sum_{n \in \mathcal{N} \setminus d} P_{nd2}
\]

Table 6.1 shows the above metrics being computed for all four routing schemes in five different networks. Observe that the RTF approach has the worst performance as it may not provide the best path for all nodes. This is because the default routing for all nodes is being forced on a tree that may be sub-optimal. The STF approach performs better than the RTF. However, since the default paths are restricted to one of the three trees, they may not be the shortest path from the source to destination. This is reflected in the average path lengths under no failure scenarios for all networks.

Figures 6.3 - 6.6 show the average path lengths for each destination node averaged over all source nodes in the respective networks. These are denoted by \(P_{d1}, P_{d2}\) for the one and two link failures respectively. The last two bars for each destination shows the performance of SPF versus Tunneling. Although the tunneling performs slightly better, the difference is marginal for all destination nodes.

Among the SPF and the tunneling approach, the former employs the three link-independent trees for failure recovery, while the latter employs tunneling (using two link-independent trees). The latter approach performs better as the indepen-
Table 6.1: Average back up path lengths.

<table>
<thead>
<tr>
<th></th>
<th>ARPANET</th>
<th>NSFNET</th>
<th>Node-16</th>
<th>Node-28</th>
<th>Mesh-4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_0$</td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_0$</td>
<td>$P_1$</td>
</tr>
<tr>
<td><strong>RTF</strong></td>
<td>4.89</td>
<td>6.78</td>
<td>7.10</td>
<td>3.45</td>
<td>5.09</td>
</tr>
<tr>
<td><strong>STF</strong></td>
<td>3.06</td>
<td>6.62</td>
<td>6.99</td>
<td>2.12</td>
<td>4.87</td>
</tr>
<tr>
<td><strong>SPF</strong></td>
<td>2.79</td>
<td>6.53</td>
<td>6.88</td>
<td>2.07</td>
<td>4.9</td>
</tr>
<tr>
<td>Tunneling</td>
<td>5.48</td>
<td>6.13</td>
<td>4.55</td>
<td>5.03</td>
<td>5.83</td>
</tr>
</tbody>
</table>
dent trees for re-routing are computed assuming only one link failure, thus offering significantly shorter paths.

We are able to achieve similar path lengths in spite of having to use only four routing entries per router in the worst case. In (81), they would require up to seven routing entries per node. Further, in our scheme it is possible to use all three trees at the same time which is not possible in the tunneling approach.

![Figure 6.3: The Average Path Lengths to each destination node in ARPANET under the four routing approaches](image)

![Figure 6.4: The Average Path Lengths to each destination node in NSFNET under the four routing approaches](image)
Figure 6.5: The Average Path Lengths to each destination node in Node16 under the four routing approaches

Figure 6.6: The Average Path Lengths to each destination node in Mesh4x4 under the four routing approaches
CHAPTER 7

FAST RECOVERY IN ETHERNET NETWORKS

Fast-recovery is a fairly well-studied topic in IP networks. Employing fast recovery in Ethernet networks however is complicated as the forwarding is based on destination MAC addresses, which do not have the hierarchical nature similar to those exhibited in Layer 3 in the form of IP-prefixes. Moreover, switches employ backward learning to populate the forwarding table entries. Thus, any fast recovery mechanism in Ethernet networks must be based on undirected spanning trees if backward learning is to be retained. In this chapter, we develop three alternatives for achieving fast recovery from single link failures in Ethernet networks. All three approaches provide guaranteed recovery from single link failures. The approaches differ in the technologies required for achieving fast recovery, namely VLAN rewrite and/or MAC-in-MAC encapsulation. We study the performance of the approaches developed on five different networks.

7.1 Fast Recovery in IP vs. Ethernet

Fast recovery from link and node failures have been studied extensively in the context of IP networks (82; 46; 83). Traditionally, IP routing table entries are computed by constructing destination rooted trees for IP prefixes. The trees for different IP prefixes are computed independent of each other. The destination-rooted trees are directed in nature, directed towards the destination. Thus, the routes between two IP endpoints may not necessarily be symmetrical. Fast recovery in IP networks is achieved by providing one or more backup ports, in addition to the primary forwarding port used under no failures. The forwarding of IP packets is then based on the destination IP address, and some additional information, which is either carried in the packet or derived from the incoming port on the router.
The forwarding mechanism of Ethernet is somewhat similar to that of IP networks. Ethernet forwarding is based on destination MAC address (analogous to destination IP address), and VLAN (equivalent of additional information used in IP address). However, the failure recovery techniques developed for IP networks cannot be directly applied to Ethernet networks due to the fact that Ethernet employs backward learning to compute forwarding tables. The switches in Ethernet networks learn about the endpoints from the packets that arrive at the switch. If a packet with VLAN tag $v$ and source MAC address $m$ arrives on port $p$, then the switch infers that the host, identified by the VLAN-MAC tuple $(v,m)$ may be reached through port $p$. This backward learning forces the routes to be symmetrical and as only one entry per host is maintained at every switch, the approach works only on undirected spanning trees. In contrast to IP networks, Ethernet networks extract an undirected tree from the underlying topology, through spanning tree protocol variants, and then learn where the hosts are attached to that tree in order to populate the forwarding table. Depending on the variant of spanning tree protocol employed, the network may employ one or more trees. While each tree may be assigned one or more VLAN tags, each VLAN has a specific tree assigned to it. Thus, mechanisms developed for forwarding and/or fast recovery must rely on undirected spanning trees unless they provide an alternative mechanism for learning MAC addresses, thus the necessary forwarding logic, at the switches.

Techniques such as Provider Backbone Bridge Traffic Engineering (PBB-TE) (84) and Shortest Path Bridging (SPB) (85) avoid the reliance on a spanning tree for regular network operation. PBB-TE provides a centralized architecture and provides a connection-oriented approach to computing forwarding entries and protection paths. SPB provides a link state control plane that allows switches to independently compute forwarding tables from a common view of the network topology. In such scenarios, it is possible to employ techniques for fast re-routing that have been developed in IP/MPLS networks. Examples of such techniques include Colored trees (83), Not-via (86), Maximally redundant trees (87) and ESCAP (4). Since PBB-TE, SPB do not employ spanning trees, they cannot employ backward learn-
ing. Thus, these approaches must be augmented with techniques to disseminate the MAC addresses of the devices connected to the switches across the network. Then, every switch will have the knowledge of the destination switch to which a packet needs to be forwarded to for a given destination MAC address. However, our focus in this chapter is to study techniques for fast rerouting that would retain backward learning.

7.2 Related Work

An undirected spanning tree gets disconnected by a link or node failure. The spanning tree protocol and its variants recover from the failures by computing another tree after the link failure. The convergence time of the original spanning tree algorithm was in the order of 30-50 seconds. RSTP (34), which was developed later in 802.1d (88) can reduce the convergence time anywhere from tens or hundreds of milli-seconds to a few seconds (89; 90) depending on considerations such as network topology, port manipulation times, time for failure detection etc. As a spanning tree involving \( N \) switches uses only \( N - 1 \) links, several links in the network remain unused. In order to improve link utilization in the network, support for multiple spanning trees was developed in 802.1q and 802.1s, where one or more VLANs can be assigned to a tree. Multiple spanning trees, where each tree is identified using a specific VLAN, are employed to distribute traffic in the network (91; 92). When a failure occurs, the failure is notified to a central system (93) by the switch connected to the failed link or detected by receivers which then notify the senders (94; 95). The traffic is then redistributed by the source over the remaining spanning trees.

Our goal in this chapter is to employ multiple spanning trees, each identified with a unique VLAN, to achieve fast recovery from link failures. By fast recovery, we refer to forwarding of packets along an alternate port (path) from the switch connected to the failed link. The switching of traffic to an alternate path implies changing the VLAN tag in the packet, thereby employing a different spanning tree for routing. In (96; 97), the authors employ multiple spanning trees over which traffic
is distributed. Upon a failure, an intermediate switch would choose a spanning tree whose VLAN identifier is higher than the one that failed. The papers, however, do not quantify the number of spanning trees needed to guarantee a single link failure recovery, or even mention how the trees are computed. Link-disjoint spanning trees have been proposed as a mechanism for achieving load balancing and failure recovery (98; 99). In these approaches, two spanning trees are constructed such that no undirected link is common to both trees. The use of link-disjoint spanning trees in conjunction with switching traffic from one spanning tree to another at the point of failure guarantees recovery from single link failures. However, a network has to be four edge connected in order to obtain two link-disjoint spanning trees (100; 101), which is a stringent requirement. Note that, in contrast, we can compute two destination-rooted link-independent spanning trees\(^1\) in two edge connected networks and achieve fast rerouting in IP networks (82).

The use of multiple spanning trees, each identified using a unique VLAN tag, is a popular approach for distributing traffic across different paths. There have been prior works that have looked at the issues of improving the fault tolerance and bandwidth guarantees. Works (102; 103; 104) have studied/proposed how one could re-structure the original spanning tree upon link failures so that the changes during re-convergence are mitigated to some extent. Such schemes often rely on messaging (102; 103), or might require the use of several backup trees (104), which cannot scale well. Although these schemes can re-connect the broken tree quickly (termed fast recovery/re-connection), it is not obvious how the packets at the switch can be re-forwarded until such a new tree is set up. Thus, there will be (i) packet losses and (ii) control messages being exchanged with other switches until a new tree is setup.

In this chapter, our definition of guaranteed fast recovery upon a link failure is the ability of the switches to re-forward packets without having to inform other routers of the failure (purely local recovery), thus limited only by the failure detection time.

\(^1\)Independent spanning trees are rooted spanning trees, such that the path from any node to the root on the spanning trees are mutually link-disjoint.
and the time to modify the packet header.

7.3 Contributions

In this chapter, we focus on providing guaranteed recovery from single link failures in Ethernet networks using multiple undirected spanning trees, each identified with a unique VLAN. To the best of our knowledge, this is the first work that provides mechanisms that guarantee purely local fast recovery from single link failures in ethernet networks that are at least two-edge connected. This connectivity requirement is the minimum to ensure single link fault tolerance. Moreover, we also develop several schemes that can achieve this goal, and study the trade-offs between these schemes that vary based on the network topology, the techniques to be employed for forwarding upon encountering a failure, and the number of VLAN tags required.

The three approaches we propose denoted by 3Trees, 2Trees, and 1Tree each provide different tradeoffs. All approaches are based on multiple spanning trees and shifting the packet from one spanning tree to another upon failure, thus requiring VLAN re-write capability at the switches. In addition, some of the approaches may require MAC-in-MAC encapsulation (802.1ah (105)), with either 1 or 2 levels of encapsulation. Figure 7.1 shows the summary of the requirements.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Connectivity</th>
<th># VLANs</th>
<th>Encap (#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3Trees</td>
<td>3E</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>2Trees</td>
<td>2E</td>
<td>2</td>
<td>Yes (1)</td>
</tr>
<tr>
<td>1Tree</td>
<td>2E</td>
<td>2</td>
<td>Yes (2)</td>
</tr>
</tbody>
</table>

Figure 7.1: A comparison of network connectivity, number of VLANs required, and 802.1ah Mac-in-Mac encapsulation requirements for the different approaches developed.

All three solutions guarantee recovery from single link failures and the forwarding mechanism upon a failure is based purely on local information. Moreover, all of the solutions are proactive approaches. Thus, the switches may be programmed with the actions to be taken for forwarding a packet under normal circumstances and
upon failures.

Figure 7.2: Construction of three spanning trees in an example three edge connected network. (a) Example network. (b)–(d) Three independent trees rooted at node A. (e)–(g) Three undirected spanning trees derived from the independent trees. The three undirected spanning trees have the property that for any link in the network, there exists a spanning tree that does not contain the link.

7.4 Organization

In Section 7.5, we describe issues that are common to any fast recovery approach that employs multiple spanning trees. In Sections 7.6, 7.7, and 7.8 we describe the 3Trees, 2Trees, and 1Tree approaches, respectively. We illustrate the working of each approach with examples. In Section 7.9, we evaluate the performance of these approaches. Section 7.10 is an Appendix which has some theorems and their proofs required to ensure the correctness of the 1Tree approach.

7.5 Fast Recovery with Multiple Spanning Trees – General Considerations

Recall that any solution for fast recovery in Ethernet networks needs to use an undirected spanning tree due to backward learning. One approach to guarantee single link failure recovery is to decompose the network into multiple spanning trees such that for any given link failure, there exists a spanning tree that doesn’t contain that link. When a failure occurs, the switch connected to the failure may simply
transfer the packet to the spanning tree that does not contain the failed link by rewriting the VLAN tag in the packet. Thus, every switch must have the knowledge of which VLAN to rewrite the packet with, upon a link failure.

One approach to use multiple spanning trees is to designate one tree as primary and the others as backup. Thus, when a link in the primary tree fails, the switch connected to the failed link would rewrite the VLAN tag in the packet to correspond to the secondary tree and forward it along the secondary tree. However, there is a drawback to this approach. As the switches populate the forwarding table entries using backward learning, the switches would not have learnt about the end hosts on the backup spanning tree. This limitation would force the intermediate switch to broadcast the packet on the backup spanning tree. However, if the traffic is spread over all the spanning trees, then the broadcasting upon a failure may be avoided. If the traffic is sent over all the spanning trees, then the switches would need a way to verify if the incoming packet has already encountered a failure or not. We may encode this information with one bit. We may use one bit from the 3-bit class-of-service (CoS) field in the VLAN header. Note that the above considerations are applicable to any fast recovery mechanism using multiple spanning trees, thus they naturally apply to the approaches developed here.

While employing link-disjoint trees has the simplicity of using two trees, it requires the network to be four edge connected. However, if we increase the number of spanning trees employed, it may relax the requirement on the network to less than four edge connected. Consequently, we are interested in the following problem: What is the minimum number of undirected spanning trees an arbitrary network can be decomposed into such that for any given link, there exists a spanning tree that does not contain the link?. For this problem, we have the following results: (i) It is known that if the network is four edge connected, we may decompose the network into two spanning trees (101; 106; 107); (ii) For three edge-connected networks, we show how we can decompose the network into three spanning trees; (iii) For two edge connected networks, the number of spanning trees required is $O(N)$, where $N$ is the number of nodes (switches) in the network.
The last result can be shown from the following two facts: (i) Any two edge connected network can be made a minimally two edge connected network. The number of links in a minimally two edge connected can be restricted to $O(N)$ (47); and (ii) We can remove one link at a time and reduce the residual graph into a spanning tree each time thus giving a possibly loose upper bound on the number of spanning tree trees required. In the worst-case (ring networks for instance), we would require as many spanning trees as the number of links showing that the upper bound is tight.

7.6 3Trees Approach

Our first approach is for three edge connected networks. We employ three spanning trees, referred to as the red, blue, and green, identified using three VLAN tags. Every link will be present in at most two of the trees by construction. Thus, when a link fails, the switch connected to the failed link can re-write the VLAN corresponding to the tree on which the link is not present.

We can construct the desired spanning trees in two ways. In the first approach, we select an arbitrary node as a root node and construct three link independent trees (30). The three independent trees are directed trees, however we simply consider the undirected version of the three trees. The second approach is to select an arbitrary node as a root, construct three rooted arc-disjoint trees (108)$^2$, and consider the undirected version of the trees. Note that the independent trees are also arc-disjoint trees with the additional constraint that the paths from any node to the root on the trees are mutually link-disjoint$^3$. Both approaches will ensure that every link is part of at most two trees, thus there always exists a spanning tree that is unaffected by any single link failure. The first approach has a computation complexity of $O(N^2)$ (30), while the second approach has a complexity of $O(E + N \log^2 N)$ (108).

$^2$An arc is a directed link (or edge).

$^3$If a path on one tree contains directed link $i \to j$, the paths on the other two trees will contain neither $i \to j$ nor $j \to i$. 
Procedure for forwarding in 3Trees approach

Given: Destination \( d \), VLAN tag of packet \( T \in \{ R, B, G \} \).

\( n_d^T \) denotes the next-hop on VLAN \( T \) to reach destination switch \( d \).

1. Forwarding:
   (a) If link to \( n_d^T \) is available, forward packet. Go to Step 4

2. Check if packet has already seen a failure:
   (a) If \( T \neq R \), the packet has seen a failure.
   (b) Drop packet, Go to Step 4

3. Recovery:
   (a) If link is present on blue VLAN
      i. Re-write VLAN tag in packet to G.
      ii. Go to Step 1.
   (b) In all other cases,
      i. Re-write VLAN tag in packet to B.
      ii. Go to step 1

4. Stop.

Figure 7.3: Forwarding procedure in 3Trees approach.

Figure 7.3 shows the procedure and the sequence of steps used for forwarding packets in the 3Trees approach. Here, we assume that default packet forwarding in the absence of failures occurs on the red VLAN. Also, if the switch is connected to the destination host, the packet is forwarded to the host. Thus, Step 1 in the procedure simply forwards the packet along the VLAN in which it was received. This is the red VLAN when the packet has seen no failure and is either blue or green when the packet has already seen a failure. If however, the forwarding link has failed, the steps for recovering upon the link failure is discussed next.

Step 2 is entered when there is a failure of the forwarding link. This step checks to see if the packet has seen a failure already. This is easily known by checking the VLAN tag of the incoming packet, and if it is not red, we drop the packet.
We proceed to the third step only when this is the first link failure that the packet has seen. Since all VLANs are spanning, and we are guaranteed (by construction) that there exists one spanning VLAN untouched by the link failure, we can use the unaffected VLAN for recovery. However, we can do slightly better as detailed in the third step explained next.

**Step 3(a)** checks to see if the blue VLAN is also present on the failed link. If so, the green VLAN which is unaffected is used to recover. If we reach **Step 3(b)**, we know that the blue VLAN is available and hence employ the same to recover.

We now illustrate with the help of an example network, the 3Trees approach. The network is as shown in Figure 7.2(a) and is three edge-connected. We consider one of the tree approaches described earlier to construct the trees, namely the independent spanning trees. Figures 7.2(b) through (d) show the three independent trees rooted at node A. Figures 7.2(e) through (g) show the undirected version of the three independent trees, which denote the three desired spanning trees to be employed as VLANs. Consider the failure of link BC. Consider a packet destined to host e that is connected to switch E, arriving at node C. In the absence of the failure, the packet would have been forwarded on the red VLAN along C-B-E. Now, it is easy to see that the link failure affects both the red and blue VLANs while leaving the green VLAN unaffected. Since each VLAN is spanning, the green VLAN can be safely be used to reach E and is picked according to Step 3(a).

The 3Trees approach is designed for three edge connected networks. Although the 3Trees approach requires only VLAN re-write capability at the switches, the three edge connectivity requirement may not be met by some real-life networks. In addition, the approach requires the use of three trees, hence three VLAN tags per virtual network. As the number of VLAN tags available is only 4096, allocating three VLAN tags per network may not be preferred. For these two reasons, it is desired to have fast recovery methods that work on two edge connected networks and preferably with fewer number of VLAN tags per network. In the following sections, we develop two approaches to achieve this goal.
7.7 2Trees Approach

In an arbitrary two edge connected network, we may require $O(N)$ spanning trees such that for every link, there exists at least one spanning tree that does not contain the link. Spanning trees with this property are required only if we are restricted strictly to using VLAN re-write as the only mechanism for fast recovery. However, we may achieve fast recovery with only two spanning trees, thus two VLAN tags, if we can employ mac-in-mac encapsulation in addition to VLAN re-writing.

We consider a two edge connected network. We select an arbitrary node as the root, say $r$, and construct two link independent trees\textsuperscript{4}. We then turn the two independent trees directed towards the root into undirected trees, referred to as the red and blue trees. Figure 7.4(a) shows an example two edge connected network. Figures 7.4(b) and (c) show the red and blue undirected spanning trees. The two spanning trees were constructed by constructing two link independent trees rooted at node A, and viewing them as undirected trees. Although we treat the spanning tree as undirected, we will still use the root $r$ as an intermediate node to achieve

\textsuperscript{4}They can be constructed in $O(E)$ \cite{60}.
fast recovery. Thus, at every node we maintain the forwarding neighbor to reach root \( r \) along the red and blue trees.

Consider a packet destined to end host \( d \) at a switch. The VLAN tag in the packet is initially red to begin with. In the absence of failures, the switch forwards the packet along the red forwarding edge on which it has learnt the destination \( d \). In addition, assume that all the switches have learned all the end hosts on both the trees. Let \( f \) denote the failure bit part of the CoS field that is initialized to zero. Let \( n^R_d \) and \( n^B_d \) denote the forwarding neighbors for destination \( d \) at the switch. In addition, let \( n^R_r \) denote the forwarding neighbor to reach the root \( r \) of the spanning tree at the switch.

The procedure shown in Figure 7.5 details the forwarding actions taken at a switch. Since the 2Trees approach will employ mac-in-mac encapsulations, the destination \( d \) and tag \( T \) in the procedure refers to the MAC address and VLAN tag of the outermost header in the packet.

In **Step 1**, we assume that if the switch is the destination \( d \) as specified in the packet header but the packet also has an inner MAC header, then Step 1 is executed with the VLAN tag and destination of the inner header. If there is no inner header and the switch is connected to the host \( d \), it forwards the packet directly to the host. In all other cases, the packet is forwarded to \( n^T_d \) as described in the procedure.

**Step 2** checks the 1 bit failure information in the CoS field to see if the packet has already seen a link failure, and if so, simply drops the packet.

**Step 3** is the key step that dictates the actions for recovering from a link failure and depends on three scenarios described next.

**Scenario 1:** \( n^R_d \neq n^B_d \). As the forwarding neighbors, hence links, are different on the red and blue trees to reach destination \( d \), the packet is simply switched from red to blue spanning tree and forwarded to the neighbor on the blue tree.

For example, consider the failure of link E–B shown in Figure 7.4(d). Consider a packet at switch E that is destined to end host \( d \) attached to switch D. The destination \( d \) would have been learned over the link E–B on the red tree and on link E–F on the blue tree. As the forwarding neighbors on the red tree, B, is different
**Procedure for forwarding in 2Trees approach**

**Given:** Destination $d$, VLAN tag of packet $T \in \{R,B\}$, One bit information $f$ in CoS field of packet.

1. **Forwarding:**
   (a) If link to $n^T_d$ is available, forward packet. Go to Step 4.

2. **Check if packet has already seen a failure:**
   (a) If $f = 1$, the packet has seen a failure.
   (b) Drop packet, Go to Step 4

3. **Recovery:**
   (a) Set the failure bit $f = 1$
   (b) If $n^R_d \neq n^B_d$:
      i. Re-write VLAN tag in packet to $B$.
      ii. Go to Step 1.
   (c) If $n^R_d = n^B_d \neq n^R_r$:
      i. Re-write VLAN tag in packet to $B$.
      ii. Encap \{Destination MAC, VLAN tag, $f$\} as \{r,$R$,1\}.
      iii. Go to Step 1.
   (d) If $n^R_d = n^B_d = n^R_r$:
      i. Encap \{Destination MAC, VLAN tag,$f$\} as \{r,$B$,1\}.
      ii. Go to Step 1.

4. **Stop.**

---

Figure 7.5: Forwarding procedure in 2Trees approach.

From that on the blue tree, F, the packet is re-written with the blue VLAN tag and forwarded to F.

**Scenario 2:** $n^R_d = n^B_d \neq n^R_r$. As we have only two spanning trees, it is possible that the forwarding neighbor for a destination is the same on the red and blue trees. For example, the forwarding neighbor for $d$ (which is connected to switch D) at switch B is switch C on both red and blue trees. Thus, the packet cannot be forwarded along the second spanning tree upon a failure. Note that the spanning
trees were constructed from independent trees rooted at $r$. The independent trees have the property that even under a link failure, every node has connectivity to the root $r$. We exploit this property and use $r$ as the intermediate point through which we may reach the destination.

Consider the situation when the red forwarding neighbor to reach the root node of the spanning tree is not the same as the forwarding neighbor for $d$. Note that a spanning tree provides a unique path for every node to reach the root. Thus, if the red forwarding neighbors for a destination and root are different at a switch, then it implies the following: (a) The switch can still reach the root on the red tree; (b) The path to the root from $d$ on the red tree traverses the intermediate switch; and therefore, the destination $d$ is still connected to the root on the blue tree. Thus, we may forward the packet to the root along the red tree and forward the packet from the root to the destination along the blue tree. To achieve this, we employ mac-in-mac encapsulation. We first re-write the VLAN tag on the original packet to that of the blue spanning tree. We then add an encapsulation (outer) header to forward the packet to the root on the red tree.

Consider the failure of link B–C as shown in Figure 7.4(e). Consider a packet destined to node $d$ that is connected to switch $D$, arriving at node $B$. The forwarding neighbor for destination $d$ at switch $B$ is the same switch, $C$, on the red and blue trees. However, the forwarding neighbor for the root on the red tree is $A$. Thus upon failure of link B–C, switch $B$ would forward the packet destined to $d$ along the path B–A on the red tree. Node $A$ would decapsulate the packet and forward the inner packet to node $D$ along A–D on the blue tree.

Scenario 3: $n_{d}^R = n_{d}^B = n_{r}^R$. In this scenario, the forwarding neighbor on the red and blue trees for the destination are the same as the forwarding neighbor on the red tree for the root. Thus, we have the following properties: (a) The intermediate switch can reach the root along the blue tree; (b) As the root and the destination are on the same side of the failure on the red tree, the root and destination are still connected on the red tree under a single link failure. Thus, we may forward the packet to the root on the blue tree and then forward the packet from the root to
the destination on the red tree. To achieve this, we simply leave the original header on the packet as is, and add an encapsulation header to forward the packet to the root on the blue tree.

Consider the failure of link E–F shown in Figure 7.4(f). Consider a packet destined to end host e that is connected to switch E, arriving at node F. The forwarding neighbors on the red and blue trees to reach E are the same switch E, which is also the same as the forwarding neighbor for the root on the red tree. Thus, switch F would encapsulate the packet to the root (switch A), and forward along the blue tree. The packet follows the path F–G–D–A and gets decapsulated at switch A. The inner packet, which is destined to node e on the red tree, gets forwarded along the path A–B–E.

The 3Trees and 2Trees approaches developed thus far required that the trees be constructed simultaneously. If one desires a specific tree to be employed under no failure scenario, then it may not be possible to construct the other tree(s). Consider the example network shown in Figure 7.6(a) and assume that the spanning tree protocol is run with node A being chosen as the root. The resulting spanning tree is shown in Figure 7.6(b). It can be easily seen that a second independent spanning tree rooted at node A cannot be constructed as both links of node A are used in the spanning tree.

![Diagram](image)

Figure 7.6: Construction of a shortest path spanning tree in an example two edge connected network. (a) Example network. (b) Undirected shortest path spanning tree rooted at node A.

In the 3Trees and 2Trees approaches, one may use a separate tree for forwarding under no failure scenario by dedicating another VLAN tag, referred to as the normal VLAN tag. Thus, under this approach, the traffic under no failure scenario would be
forwarded based on the normal VLAN tag. When the packet encounters a failure, the switch would treat the packet as one that was forwarded based on the VLAN tag of the failed outgoing edge and select the corresponding backup forwarding action. If used with normal VLAN tag, the 3Trees and 2Trees approaches would require a total of four and three VLAN tags, respectively.

7.8 1Tree Approach

To overcome the limitation of the 3Trees and 2Trees approaches, we develop another approach that allows any default tree to be used for forwarding traffic under no failure scenario. We refer to this approach as the 1Tree approach.

The 1Tree approach is based on the technique for backup port assignment for IP fast re-route developed by Xi et al. (4). Using their approach as a starting point, we show that it is possible to construct a collection of trees to recover upon a link failure with mac-in-mac encapsulations. The collection of trees can all use the same VLAN tag. Thus, the 1Tree approach requires two VLAN tags: one VLAN tag to identify the spanning tree used for forwarding when there are no failures, and a second VLAN tag that is shared by a collection of trees.

We first briefly describe the ESCAP algorithm introduced in (4). We then describe how we employ it in our context. Given a two edge connected network $G$ and a primary tree rooted at a destination node, the 1Tree-IP approach computes backup ports for every node. Every node has a primary forwarding neighbor and a backup forwarding neighbor. In the context of IP networks, the packet forwarding is as follows: (1) If a packet is received from any node other than the primary forwarding neighbor, then the packet is forwarded to the primary forwarding neighbor. (2) If the primary forwarding link is unavailable due to a failure or if the packet is received from the primary forwarding neighbor, the packet is forwarded to the backup forwarding neighbor. For a given primary tree rooted at a destination node, the computation of backup forwarding neighbors at all other nodes is as follows. The links in the primary tree are considered in the depth/breadth-first manner. Consider
a link \( \ell \) that connects node \( x \) with its primary forwarding neighbor \( y \). The removal of link \( \ell \) would disconnect the primary tree into two sub-trees. One subtree would contain node \( x \), referred to as \( T(x) \), while the other would contain the destination and node \( y \), denoted as \( G \setminus T(x) \). If node \( x \) does not have a backup forwarding node assigned already, then the shortest path from \( x \) to any node in \( G \setminus T(x) \), using only the edges on \( T(x) \) except for the last hop, to reach outside \( T(x) \) is computed. The node \( z(x) \in G \setminus T(x) \) at which the directed path terminates is referred to as the exit node of \( x \). This directed path provides the backup forwarding neighbors for all the nodes in the path, and \( z(x) \) is the exit node for all nodes in the path. This procedure is repeated until the backup forwarding nodes for all the nodes are computed.

An example shortest path tree rooted at node \( A \) shown in Figure 7.7(a). The backup forwarding nodes assigned for the primary tree rooted at node \( A \) according to the ESCAP algorithm in (4) is shown in Figure 7.7(b).

![Figure 7.7: Construction of backup port assignments according to (4) on an example two edge connected network. (a) Tree rooted at node A. (b) Backup forwarding node assignments. (c) The red VLAN. (d) The blue VLAN.](image)

We now adapt the ESCAP approach to the VLAN setting. We first observe that any primary undirected tree to be used as the default VLAN can be made directed and rooted at an arbitrary node \( r \). Now, we would like to use all the backward directed arcs that ESCAP would compute on the primary tree and treat them in an undirected form, so that, we can employ it as the secondary VLAN that can
somehow be used to recover from a link failure on the primary VLAN. However, this is not straightforward as there could be cycles on the secondary VLAN. Hence, the idea is to modify the original algorithm to avoid these cycles so that we can view all the directed backup-arcs in an undirected form and employ them as a secondary VLAN. We describe the modified algorithm, denoted as $M$-ESCAP, next and refer the interested reader to the Appendix for an example of how cycles could result in the absence of any modifications to ESCAP, a detailed description of the properties of $M$-ESCAP and its proof of correctness.

On the primary tree, we construct the backup forwarding paths for each link failure, considering the links in the breadth first order as in ESCAP. The key change we make to ESCAP is as follows. Assume that we are considering the failure of the primary link for node $x$. In the backup path of $x$, let the last hop be $y \rightarrow z(x)$. Thus, $z(x)$ is the exit node of $x$. If at the time of this computation, we find that $z(x)$ does not have a backup forwarding node assigned, we assign $y$ to be the backup forwarding node. This key modification to the ESCAP algorithm guarantees that the forwarding backup node assignment would never result in a directed loop\(^5\). This key change ensures that when the directed forwarding edges are turned into undirected edges, the resultant sub-network of backup forwarding links would remain a tree. Thus, we can use the same VLAN tag on all of these trees, exploiting spatial re-use of the VLAN tag. Applying this technique on our example network, we obtain the primary tree in red as shown in Figure 7.7(c) and the forest in blue as shown in Figure 7.7(d). The idea then is to use the forest to take us across the failure to the other side of the primary spanning tree which contains the node attached the destination host.

However, since we require the spanning tree and forest to be undirected, we lose the information previously carried implicitly by the directed arcs. Thus, we need to maintain at every node $x$, the exit node for itself and for each of the neighbors of $x$ that use $x$ to reach the root $r$. These are the children of $x$ in the directed spanning

---

\(^5\)Note that the directed loop is not a concern in the IP fast rerouting. It is only a concern for our approach as we will turn the directed forwarding edges into undirected edges.
tree. So, for instance, in the example network in Figure 7.7(c), node B would need to save that its exit node is node D, and also save the exit nodes of nodes C and E\(^6\) which are nodes D and G respectively. Thus, at every node we maintain the forwarding neighbors to reach the root and all the exit nodes as detailed above.

![Network Diagram](image)

(a) Failure scenario 1 (c) Failure scenario 2

Figure 7.8: Failure scenarios when the first vlan can be arbitrarily chosen. Two failure scenarios depicting the different backup forwarding mechanisms required on the example network.

Consider a packet destined to end host \(d\) at a switch. Assume that the VLAN tag in the packet corresponds to the red tree to begin with. As before, \(n_d^R\) denotes the forwarding neighbor for destination \(d\) at the switch and \(f\), the failure bit is initialized to zero. Let \(n_r^R\) denote the forwarding neighbor to reach the root \(r\) of the spanning tree at the switch. When the network has no failures, the switch would forward the packet along the red forwarding edge on which it has learnt the destination \(d\).

Figure 7.9 shows the procedure used for forwarding packets under the 1Tree approach. As in the 2Tree approach, the destination \(d\) and VLAN tag \(T\) in the procedure correspond to the outermost header of the packet. In Step 1, we assume that if the switch is the destination \(d\) as specified in the packet’s outermost header but the packet also has inner MAC header(s), then Step 1 is executed with the VLAN tag and destination of the next inner header. If there is no inner header and the switch is connected to the host \(d\), it forwards the packet directly to the host. In all other cases, the packet is forwarded to \(n_d^T\) as described in the procedure. Now, in the event of a link failure at the current switch, denoted by \(x\), the actions for recovering from the failure depends on two scenarios.

**Scenario 1**: \(n_d^R = n_r^R\). As the forwarding neighbors, hence links, are the same to

---

\(^6\)Children of node B in Figure 7.7(a).
Procedure for forwarding in 1Tree approach

Given: Destination \( d \), VLAN tag of packet \( T \in \{ R, B \} \), One bit information \( f \) in CoS field of packet.

1. Forwarding:
   (a) If link to \( n^T_d \) is available, forward packet. Go to Step 4.

2. Check if packet has already seen a failure:
   (a) If \( f = 1 \), the packet has seen a failure.
   (b) Drop packet, Go to Step 4

3. Recovery:
   (a) Set the failure bit \( f = 1 \)
   (b) If \( n^R_d = n^R_r \):
      i. Encap \{Destination, VLAN tag, \( f \}\) as \{\( z(x) \), \( B \), 1\}.
      ii. Go to Step 1.
   (c) If \( n^R_d \neq n^R_r \):
      i. Encap \{Destination, VLAN tag, \( f \}\) as \{\( n^R_d \), \( B \), 1\}.
      ii. Encap \{Destination, VLAN tag, \( f \}\) as \{\( z(n^R_d) \), \( R \), 1\}.
      iii. Go to Step 1.

4. Stop.

Figure 7.9: Forwarding procedure in 1Tree approach.

reach destination \( d \) and the root \( r \) on the primary spanning tree, it implies that the failed link was used to reach the primary forwarding neighbor of \( x \) in the directed spanning tree rooted at \( r \). Thus, we use the blue VLAN to reach the exit node of \( x \), which by definition is outside the sub-tree rooted at \( x \), and hence guaranteed to be connected to the switch with destination host \( d \) on the red VLAN even after the link failure. To achieve this, we require an encapsulation to reach the exit node of \( x \). From there, the packet can resume normal forwarding on the red VLAN towards the switch to which the host \( d \) is attached.

For example, consider the failure of link E–B shown in Figure 7.8(a). Consider a packet arriving at switch E that is destined to end host \( d \) attached to switch D.
The destination \( d \) would have been learned over the link E–B on the red tree. As the forwarding neighbor to reach the destination on the red tree, B, is the same to reach the root A on the red tree, the packet is encapsulated so that it first reaches the exit node of E, which is node G. This can be done with an outer blue VLAN tag destined to G. Upon reaching G, the packet is forwarded on the red VLAN towards D, according to the unmodified original header.

**Scenario 2:** \( n_d^R \neq n_r^R \). As the forwarding neighbors, hence links, are different to reach destination \( d \) and the root \( r \) on the primary spanning tree, it implies that the failed link was a child of node \( x \) in the directed spanning tree rooted at \( r \). Thus, we use the primary red VLAN to reach the exit node of \( n_d^R \), then use the VLAN corresponding to the blue forest to reach \( n_d^R \). This requires two levels of encapsulation, one to reach the exit node of \( n_d^R \), and the second to reach \( n_d^R \) itself. Essentially, we have used a combination of the red spanning tree and the blue forest to tunnel to the other side of the failed link. From there on, the packet can resume normal forwarding on the primary spanning tree towards the switch to which the host \( d \) is attached.

For example, consider the failure of link A–B shown in Figure 7.8(b). Consider a packet arriving at switch A that is destined to end host \( f \) attached to switch F. The destination \( f \) would have been learned over the link A–B on the red tree. As the forwarding neighbor to reach the destination on the red tree, B, is different from that to reach the root A (itself in this case), the packet is encapsulated such that it first reaches the exit node of B, which is switch D, and then the reaches switch B. This is done with an outermost red VLAN tag with destination as D, an inner header with destination B on the blue VLAN tag, and the innermost header being the original one to reach node F on the red VLAN left untouched.

### 7.9 Performance Evaluation

We evaluate the three approaches developed in this chapter: 3Trees, 2Trees and 1Tree (3T, 2T, and 1T for short) approaches. We consider five topologies: NSFNET,
ARPANET and three hypothetical topologies Node16, Node28 and Mesh4x4, as shown in Figure 7.10. The first two of these topologies are wide area networks. All the networks considered are three edge connected, with the exception of NSFNET to which we added an additional link to make the network three edge connected. Thus, all three schemes developed in this chapter can be employed on each of these topologies for failure recovery. We are interested in the average path length of the approaches under single link failures. This will help quantify the deviation from the path lengths during normal forwarding in the absence of failures. Studying this deviation is useful since it is representative of the end-to-end delay observed by the destination due to the failure, and thus helps in provisioning buffer capacity for real time applications.

![Networks](image)

*Figure 7.10: Networks considered for performance evaluation.*

In the 3T and 2T approaches, we simply pick the red tree as the default tree. In the 3T approach, upon link failure, we check to see if the failed link is part of the blue VLAN. If so, we use the green VLAN and otherwise we simply use the blue VLAN. In case of a link failure when using the 2T approach, the re-forwarding behavior is exactly as is defined in Section 7.7. In the 1T approach, we use a shortest path tree to mimic the behavior of the spanning tree protocol as the default tree. In all these approaches, the root of the trees is chosen based on the selection criteria
outlined in Section 7.9.3.

In all these schemes, we first compare the path lengths under no failures to that using optimal shortest paths. We then evaluate the average path lengths under single link failures. Formally, we define the following path metrics that are useful in benchmarking and comparing the various schemes.

7.9.1 No Failures

We compute the the ideal average path lengths in the network when there are no failures by computing the shortest path between the two nodes in the given network topology. Note that it may not be possible to obtain these average path lengths in reality in Layer-2 networks, as spanning trees are typically constructed with one node as root. Thus, while every node may have a shortest path to the root, the path between any two nodes is not guaranteed to be the shortest. This ideal metric, however, serves as a good benchmark for comparison. Let \( P_{nd0} \) denote the shortest path length from a node \( n \) to a destination \( d \) in a given network. Then, the average ideal path length under no failures, denoted as \( SPF - No Failure (P_0) \), is given by:

\[
P_0 = \frac{1}{N(N - 1)} \sum_{d \in N} \sum_{n \in N \backslash d} P_{nd0}\tag{7.1}
\]

where \( N \) denotes the number of switches in the network.

Now, we consider the path length between two nodes on a spanning tree. Given the default spanning tree that is employed by any forwarding mechanism when the network does not have failures, let \( P'_{nd0} \) denote the path length between node \( n \) and destination \( d \). The default spanning tree for the 1T approach would be the shortest path tree rooted at the root node \( r \) denoted as \( SPT - No Failure (P'_0) \). For the 2T, 3T approaches, this would be the length on the red tree, and denoted as \( Red Tree - No Failure (P'_0) \)

\[
P_0' = \frac{1}{N(N - 1)} \sum_{d \in N} \sum_{n \in N \backslash d} P'_{nd0}\tag{7.2}
\]
It is useful to view the relation of $P'_0$ to $P_0$ as the overhead (or stretch) involved in using destination-rooted trees in IP networks to the constraint of using a single undirected spanning tree in Ethernet networks.

7.9.2 One Link Failures

We are interested in computing the average path length when a link failure affects the default path from a source to a destination, which in turn implies only failures of links present on the default spanning tree. All other single link failure scenarios do not affect the re-forwarding schemes and hence are not considered as they could falsely indicate better performance.

Let $P_{nd1}$ be the shortest path recovery length from node $n$ to $d$ averaged over all link failures that affect the default path between $n$ and $d$, computed as:

$$P_{nd1} = \frac{1}{P'_{nd0}} \sum_{f} P_{ndf} \quad (7.3)$$

where $f$ denotes a link failure and $P'_{nd0}$ signifies the number of failure scenarios that affect the default path. Let $s$ denote the switch that sees the failure $f$, also known as the point of local repair. Then, $P_{ndf}$, the path length between nodes $n$ and $d$ under failure $f$ is computed as the sum of (1) the path length from $n$ to $s$ on the default or primary VLAN, $P'_{ns0}$, and, (2) the shortest path length from $s$ to $d$ in the network that has the failed link $f$ removed. Thus, the shortest recovery path represents the path that a packet would take until it sees the failure, added to the optimum path from the point of failure to the destination. The average ideal recovery path length under one failure in the network denoted as $SPF - One failure (P_1)$ is now given by,

$$P_1 = \frac{1}{N(N-1)} \sum_{d \in N} \sum_{n \in N \setminus d} P_{nd1} \quad (7.4)$$

Much like $P_0$ in the case of no failures, this metric is an average of optimal paths that one can achieve under single link failures and hence serves as a good benchmark to evaluate each of the schemes.
We compute the path length under a single link failure for the fast recovery techniques developed as follows. The packet traverses from switch \( n \) to switch \( s \) as defined by the path in the default forwarding tree. From switch \( s \), the packet is forwarded on the recovery path dictated by the approach in question. Let \( P'_{nsf} \) denote the recovery path length from \( n \) to \( d \) under the failure scenario \( f \), then the total path length for the packet is computed as \( P'_{ns0} + P'_{sd} \) where \( P'_{sd} \) is different for each of the three approaches.

Now, let \( P'_{nd1} \) be the recovery path length from node \( n \) to \( d \) averaged over all link failures that affect the default path between \( n \) and \( d \), computed as:

\[
P'_{nd1} = \frac{1}{P'_{nd0}} \sum_{\forall f} P'_{ndf}
\]

\[
P'_{1} = \frac{1}{N(N-1)} \sum_{d \in N} \sum_{n \in N \setminus d} P'_{nd1}
\]

7.9.3 Root node selection

We highlight that there is a choice to be made for the root, the node at which the tree(s) are constructed in each of the three different approaches and that it can play a role in the performance of the various approaches. This is because we do not have the flexibility of constructing the trees per destination in any of the approaches. This was also alluded to earlier in Section 7.1 as a key difference between the approaches for Ethernet networks that retain backward learning as opposed to approaches for forwarding in IP networks or Ethernet networks that disable learning.

In choosing a root, we optimize for the scenario under which there are no failures. The rationale for this design choice is that we expect the network to be forwarding traffic under no failures more often than in the presence of failures. Thus, for each approach, we pick the node that minimizes the average distance over all source, destination pairs on the primary VLAN. This choice also implicitly assumes a uniform traffic matrix and any a priori information on the traffic matrix can be included by appropriately adding weights to corresponding source, destination pairs. Formally,
the root denoted by $r_{min}$ is picked as defined below.

$$r_{min} = \min_{r \in \mathcal{N}} \frac{1}{N(N-1)} \sum_{d \in \mathcal{N}} \sum_{n \in \mathcal{N} \setminus d} \mathcal{P}_{r_{nd0}}'$$ (7.7)

where $\mathcal{P}_{r_{nd0}}'$ corresponds to the distance from node $n$ to node $d$ on the primary VLAN (SPT in the 1T approach, Red Tree in the other two approaches) for a fixed root $r$.

In our earlier descriptions of path lengths, we have simply dropped the superscript of the root node that we chose ($r_{min}$) according to Equation 7.7 in the interest of clarity.

7.9.4 Performance results

Before we delve into the performance results of the path lengths on the three approaches, we first briefly study the impact of the root selection. Figure 7.11 shows the average path length under single link failures in each of the three approaches on one particular network, ARPANET for different choices of the root node. The bar is white denotes the optimal root chosen according to Equation 7.7. We observe that although the root selection has not been optimized for the failure performance, the variance in the average single link failure path length is marginal across different root node selections in all the approaches. We observe similar trends in all other networks as well but leave the figures for brevity.

![Figure 7.11: Impact of root node selection on single link failure performance of the three approaches.](image)

We now discuss the performance of path lengths under the choice of a corresponding $r_{min}$ in each of the three approaches. The average path length across all
the nodes in the network for the 1Tree, 2Tree, and 3Tree fast recovery techniques developed in this chapter are shown in Tables 7.1 – 7.3. Some interesting observations we can draw from the results are discussed next.

We study the $P'_0$ metric across the 1T, 2T and 3T approaches. This metric represents the average path length on the primary tree in the absence of failures. We see in Tables 7.1, 7.2 and 7.3 that the difference among the approaches is marginal across all the considered networks. The interesting take from this observation is that the choice of the default tree under no failure could be different from the shortest path tree constructed by the spanning tree algorithms without significantly increasing the path length when the network does not have any failures. This is because, even in the case of the tree constructed by spanning tree algorithms, the algorithm merely optimizes the paths from all nodes to the root and, thus does not necessarily do well when all $s,d$ pairs are accounted for. Hence, it may not be necessary to dedicate an extra VLAN tag to use the shortest path tree with respect to root or a specific spanning tree as the default VLAN in the 3T and 2T approaches. This may be lucrative as these approaches involve no/fewer encapsulations as opposed to the 1T approach. However, for the cases in which a default tree has to be fixed for reasons besides path length optimizations (such as policy, economics etc.), the 1T approach allows for such flexibility absent in the other approaches.

Next, we study $P'_1$, the key performance metric for evaluating the average path length under single link failures. We see from Tables 7.1, 7.2 and 7.3 that in the 1T and 3T approaches, we have a stretch of $\approx 1.3x$ with respect to $P_1$, the average of the optimal recovery path lengths under single link failures. The 2T approach on the other hand, has a stretch of $\approx 1.8x$ with respect to $P_1$. This is due to the fact that the independent trees in the 2T approach are not constrained in any fashion. They could each be of arbitrary depth and hence re-forwarding paths that need to use the root (which is the case for the majority of failures) as a via-point becomes lengthy. In the 3T approach, although we employ independent trees, we avoid detours involving the root thus mitigating the stretch involved. In the 1T approach, since the default
Table 7.1: Average back up path lengths using the 1T approach.

<table>
<thead>
<tr>
<th></th>
<th>ARPANET</th>
<th>NSFNET</th>
<th>Node-16</th>
<th>Node-28</th>
<th>Mesh-4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPF - No Failure (( P_0 ))</td>
<td>2.79</td>
<td>2.08</td>
<td>2.51</td>
<td>3.57</td>
<td>2.13</td>
</tr>
<tr>
<td>SPT - No failure (( P'_0 ))</td>
<td>3.76</td>
<td>3.01</td>
<td>3.63</td>
<td>4.93</td>
<td>3.27</td>
</tr>
<tr>
<td>SPF - One failure (( P_1 ))</td>
<td>4.93</td>
<td>3.82</td>
<td>4.68</td>
<td>6.35</td>
<td>3.72</td>
</tr>
<tr>
<td>1T - One failure (( P'_1 ))</td>
<td>6.48</td>
<td>4.99</td>
<td>6.43</td>
<td>9.32</td>
<td>4.95</td>
</tr>
</tbody>
</table>
Table 7.2: Average back up path lengths using the 2T approach.

<table>
<thead>
<tr>
<th></th>
<th>ARPANET</th>
<th>NSFNET</th>
<th>Node-16</th>
<th>Node-28</th>
<th>Mesh-4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPF - No Failure</strong> ((P_0))</td>
<td>2.79</td>
<td>2.08</td>
<td>2.51</td>
<td>3.57</td>
<td>2.13</td>
</tr>
<tr>
<td><strong>Red Tree - No failure</strong> ((P'_0))</td>
<td>4.11</td>
<td>3.14</td>
<td>4.17</td>
<td>5.99</td>
<td>3.27</td>
</tr>
<tr>
<td><strong>SPF - One failure</strong> ((P_1))</td>
<td>4.90</td>
<td>3.94</td>
<td>5.00</td>
<td>7.08</td>
<td>3.72</td>
</tr>
<tr>
<td><strong>2T - One failure</strong> ((P'_1))</td>
<td>10.59</td>
<td>6.54</td>
<td>7.53</td>
<td>11.95</td>
<td>7.78</td>
</tr>
</tbody>
</table>
Table 7.3: Average back up path lengths using the 3T approach.

<table>
<thead>
<tr>
<th></th>
<th>ARPANET</th>
<th>NSFNET</th>
<th>Node-16</th>
<th>Node-28</th>
<th>Mesh-4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPF - No Failure ($P_0$)</td>
<td>2.79</td>
<td>2.08</td>
<td>2.51</td>
<td>3.57</td>
<td>2.13</td>
</tr>
<tr>
<td>Red Tree - No failure ($P'_0$)</td>
<td>4.19</td>
<td>3.08</td>
<td>3.63</td>
<td>5.10</td>
<td>3.33</td>
</tr>
<tr>
<td>SPF - One failure ($P_1$)</td>
<td>5.13</td>
<td>3.92</td>
<td>4.68</td>
<td>6.38</td>
<td>3.75</td>
</tr>
<tr>
<td>3T - One failure ($P'_1$)</td>
<td>6.42</td>
<td>5.14</td>
<td>6.22</td>
<td>8.89</td>
<td>5.12</td>
</tr>
</tbody>
</table>
tree is an SPT, the height of the tree is kept low which helps keep the detour paths involving the root to shorter lengths. Hence, we suggest that the 2Trees approach be employed only if the network is sparsely (less than three-edge) connected and two header encapsulations (required in the 1Tree approach) may need to be avoided for packet re-forwarding.

Based on the performance results, we observe that the 3Tree approach provides a good compromise in terms of the path lengths achieved and the overhead involved in fast recovery. Between the 2T and 1T approaches, we observe that the 1T approach provides better path lengths even under single failure scenarios compared to the 2Tree approach. The performance difference is approximately 40% between the two approaches. The tradeoff is in terms of employing two levels of encapsulation in case of the 1T approach versus one level of encapsulation in the 2T approach.

7.10 Appendix

The primary tree used in the ESCAP algorithm which when made undirected is referred to as the red VLAN. Similarly making all the backup arcs in ESCAP as undirected, we get sub-graphs that could be disconnected from one another. We refer to this collection as the blue VLAN. We first show that the blue VLAN could have cycles in it. This is not a concern in ESCAP since the incoming link information and the fact that the primary tree is directed and rooted at a node helps avoid looping on the backup arcs. However, when we make them undirected so that they can be used symmetrically in Ethernet networks, and employ a VLAN ID, we cannot afford to have a cycle as this can cause inconsistencies in the backward learning mechanism on the VLAN and potential broadcast storm problems as well. In this Appendix, we show how such cycles can be avoided by making a small modification to the original ESCAP algorithm.

Lemma 6 If there exists an undirected cycle $C$ on the blue VLAN, then the directed arcs that make up the cycle must also form a directed cycle $\tilde{C}$.
Proof We prove the result by contradiction. Let’s say we have an undirected cycle. Now, let’s look at the direction of the arcs that created the undirected cycle. Assume that it is not a directed cycle. Then, there is at least one node on the cycle that has two outgoing arcs. However, this is not possible because each node has at most one outgoing arc for backup. The root has none while all other nodes have exactly one. Thus, we are only concerned with the possibility of a directed cycle. □

Figure 7.12: Example network (a) Example network (b) Primary and backup arcs on the example network.

Figure 7.12 shows an example network alongside its primary tree and backup port assignments. It can be seen that the nodes $n_1, n_2, n_3$ are present on a cycle on the blue VLAN. In order to avoid such directed cycles, we now propose the following modification to the ESCAP algorithm.

**Modification to ESCAP:** Whenever a node $z$ gets picked as the exit node during ESCAP, say through link $y \rightarrow z$ and Node $z$ does not have a backup neighbor assigned thus far, we force $y$ to be the exit node for $z$. This modification($M$) to the original ESCAP algorithm for single link failures is denoted as $M$-ESCAP.

We will show that $M$-ESCAP is (1) One of the several outputs that the original ESCAP algorithm would compute, and thus, does not violate any of the algorithm’s properties, and (2) Avoids the formation of directed cycles of length greater than two, thus ensuring that the undirected graph of the backup arcs is a collection of trees.

**Theorem 13** $M$-ESCAP is simply one of the several possible outputs of the ESCAP algorithm.
Before we prove the theorem, it is worth noting that the original algorithm can produce several valid outputs (namely the set of backup forwarding ports) since it relies on tree traversal techniques such as breadth-first search (BFS) and/or depth-first search (DFS) to compute the backup forwarding neighbors. The outputs of such tree traversal algorithms can differ depending on the implementation (data structures used to represent the graph). In our proofs, we will assume that the primary tree in ESCAP is explored using a DFS and the backup forwarding ports are computed using a BFS.

**Proof** Consider a link, \( y \rightarrow z \) that is chosen as an exit link for some node (maybe \( y \) itself or some other node \( x \)). The link \( y \rightarrow z \) is not present on the primary tree by definition. Now, the original ESCAP and \( M \)-ESCAP differ only in the case in which node \( z \) does not already have a backup neighbor assigned. In such a scenario, we force \( y \) to be the exit node for \( z \) thus creating directed cycles of length two. This is illustrated in Figures 7.13(a), (b) below.

![Figure 7.13](image)

(a) Exit link \( y \rightarrow z \) in ESCAP  (b) Forcing link \( z \rightarrow y \) in \( M \)-ESCAP.

Figure 7.13: Illustration of \( M \)-ESCAP: Forcing the backup forwarding neighbor for an exit node whose backup has not yet been defined.

We now consider the later stage at which node \( z \) would have its backup path computed in the original ESCAP algorithm. Such a computation would be instigated either for the failure of node \( z \)’s primary forwarding link (or) for the failure of a primary forwarding link of some ancestor of \( z \) which tries to use \( z \) to exit the primary tree. Without loss of generality, let us call this node that instigates the backup path computation of node \( z \) as node \( w \).

We consider the computations in ESCAP for \( T(w) \). Now, given that the link \( z \rightarrow y \) is not on the primary tree, we have two possibilities for the backup path
computation for node $z$.

1. The link $z - y$ is outside $T(w)$ and hence safe to use.

2. The link $z - y$ is inside $T(w)$ and hence cannot be used.

**Case 1:** The first possibility ensures that node $z$ would pick node $y$ or some other 1-hop neighbor, the link to whom is not on the primary tree as the exit link. Hence, $\mathcal{M}$-ESCAP is simply one of the choices in ESCAP as desired and we are done. We note that node $z$ will not pick a longer path in ESCAP since $y$ is a neighbor that is only one hop away.

![Figure 7.14: Illustration of a scenario that cannot happen in ESCAP. Nodes $y$ and $z$ are both in $T(w)$. Node $z$ has no backup arc defined yet but node $y$ has its backup arc (and exit link) defined as $y \rightarrow z$.](image)

**Case 2:** We now show that the second possibility will never occur in ESCAP which will complete the proof. Consider Figure 7.14 for reference. We know that node $y$ has a backup arc and exit link defined at an earlier stage of ESCAP, it implies that some ancestor of $y$ (or $y$ itself) is using $y$ to exit the primary tree. Now, this ancestor, say node $x$ could either be in $T(w)$ or an ancestor of $w$ on the primary tree.

**Case 2(a)** Node $x$ cannot be in $T(w)$ since node $w$ itself does not have a backup path yet, and thus, no link failure within $T(w)$ will have been considered in the DFS on the primary tree.

**Case 2(b)** Consider now the case where node $x$ is an ancestor of node $w$ and is using the link $y \rightarrow z$ present in $T(w)$ as its exit link. Then, if node $x$ were to use $y$ to exit the primary tree, then it must also use node $w$ since the backup path of any node will have to stay on the primary tree until the last hop on the exit link, $y \rightarrow z$. 
But, since node \( z \) and hence \( w \)'s backup path is yet to be computed, node \( w \) cannot already have a backup path leading to a contradiction.

Hence, the possibility of link \( z - y \) being inside \( T(w) \) will never occur completing the proof.

\[ \square \]

**Theorem 14** \( \mathcal{M} \text{-ESCAP} \) guarantees that there is no directed cycle consisting of backup arcs of length greater than two.

**Proof** We prove this theorem by contradiction. Assume that there exists a directed cycle of length \( k > 2 \). Let the nodes on the cycle be denoted as \( n_1, n_2 \ldots n_k \). Now all links on this cycle cannot also be on the primary tree (in the opposite direction) by the definition of a tree. Thus, there exists at least one exit link on this cycle. Let us call the set of exit links on this cycle as \( E \) and denote the exit links as \( e_1, e_2, \ldots, e_k \) which appear in the same order on the directed cycle. Consider Figure 7.15 for reference which shows a directed cycle of backup arcs with the exit links marked on the cycle.

![Figure 7.15](image)

Figure 7.15: Illustration for the contradiction of the existence of a directed cycle in \( \mathcal{M} \text{-ESCAP} \).

Consider the stage at which the backup arc for node \( n_1 \) which is exit link \( e_1 \) is computed. Now, according to \( \mathcal{M} \text{-ESCAP} \), the only reason why \( n_2 \) will not pick \( n_1 \) as its backup forwarding neighbor is because it already has a backup path and exit link computed. Also, since every node has exactly one backup forwarding neighbor in ESCAP and hence, also in \( \mathcal{M} \text{-ESCAP} \) from Theorem 1, we know that the exit link for \( n_2 \) should also belong to the directed cycle and hence is \( e_2 \). Now, by extension of the same argument, we can see that \( n_{i+1} \) should also have been assigned an exit
link. Continuing the argument in this fashion, we arrive at last exit link on the directed cycle, $e_k$. It is easy to see that if a directed cycle as in Figure 7.15 were to exist, then, node $n_{j+1}$ will have to have its exit link as $e_1$. This would imply that $n_1$ already has a backup neighbor assigned which contradicts the assumption we started with; namely that we are at the stage at which the exit link for node $n_1$ is to be computed. We note that this proof argument holds for any number of exit links ($1 \leq |\mathcal{E}_l| \leq k$) on the cycle.

$\square$

**Corollary** The undirected graph of the set of directed backup arcs does not contain any cycles. Thus, the blue VLAN is a forest (of trees) as required.
This dissertation develops several graph algorithms that find applications in network tomography and fault tolerance.

In Chapters 2 and 3, we studied the problem of identifying additive link metrics. In Chapter 2, we developed the necessary theory for identifiability of additive link metrics in undirected and directed networks. We showed that three-edge connectivity is both necessary and sufficient to identify all link metrics in undirected networks using one monitor employing only monitoring cycles. We developed an $O(|N|^{2})$ algorithm which uses three link-independent trees to compute the linearly independent cycles in any three-edge connected network. For networks that do not satisfy the necessary conditions, we showed that the minimum number of monitors, their placement and the linearly independent cycles and paths may be identified. For networks with symmetric directed links, we showed that the sum of the number of monitors and the number of links for which the edges need to be known equals the number of nodes in the network, irrespective of the connectivity in the network.

In Chapter 3, we defined and derived the link rank (the maximum number of linearly independent cycles for a given placement of monitors) of arbitrary undirected networks. We developed a linear time algorithm to compute the rank and a polynomial time algorithm to compute the cycles that achieve the rank. Further, we also pinpointed the exact links whose metrics can and cannot be identified by developing sophisticated graph decomposition techniques. While Chapter 2 showed how to address rank deficiency using more monitors, Chapter 3 elucidated how to achieve the same result by specifying the links whose metrics we would require as a priori information.

In both chapters, we exploit the graph structure inherent in the problem to
avoid expensive matrix inversions leading to much better bounds on computational complexity. This is the reason we were able to compute the link rank in $O(|L|)$ time when matrix inversion complexity is typically super-quadratic /cubic.

Future research could involve extending these results to networks with directed links, where links may not be bi-directional, i.e., if $i \rightarrow j$ exists, $j \rightarrow i$ does not necessarily exist and, establishing results for more realistic models that account for noise and time variance.

Chapters 4 and 5 were on edge-independent spanning trees. In Chapter 4, an old graph theoretical result on independent spanning trees, the implication conjecture thought to be closed in (63) is opened up again by a counter-example we provide and as a result, the implication conjecture and whether four edge independent spanning trees can be constructed in four edge connected graphs remain open problems at this time. Based on our experience in developing the counterexample, it is not apparent if a general approach to compute edge independent spanning trees can be derived just from the corresponding vertex independent spanning trees (or vice-versa), without the knowledge of how the vertex independent spanning trees were constructed as assumed in (63).

In Chapter 5, we provide an algorithm to construct three edge independent spanning trees in three edge connected graphs. Given that a generic algorithm for deriving edge independent trees from vertex independent trees (or vice-versa) is not known as of this writing, we believe that the construction of three edge independent trees is a significant result, not just because it partially answers an outstanding graph theoretical problem, but for the sheer number of applications it provides in various areas as was seen across the chapters of this dissertation.

Chapters 6 and 7 were on achieving fast recovery from link failures in IP/Ethernet networks. In Chapter 6, we employ three link-independent trees to be used for multipath routing and fast re-routing in an IP network. We develop a routing scheme that is capable of disjoint multipath routing using only the destination address in the packet header. We also develop three fast re-routing approaches using the trees, all of which are guaranteed to recover from arbitrary two link fail-
ures. Further, the routing table entries are limited to at most four per node and very minimal packet overhead is required. Through simulations, we show that the path lengths obtained by using the three trees is close to that obtained with a previous tunneling approach, even though the latter employs seven routing table entries per node and does not provide multipath routing capability.

In Chapter 7, we develop three different techniques for achieving fast recovery from single link failures in Ethernet networks. Two of the approaches (2Tree, 1Tree) are applicable to networks that are at least two edge connected and one of them (3Tree) is applicable to networks that are at least three edge connected. We show that the approaches provide a tradeoff in terms of path length performance and the techniques required for achieving fast recovery, such as the number of VLAN tags, dependence on VLAN rewrite, and mac-in-mac encapsulations. Based on the performance results, the approach employing three spanning trees offers a better tradeoff in path length across the considered networks.
REFERENCES


[105] “802.1ah - Provider Backbone Bridges,” IEEE Standard, Online.

