Making Networks Robust to Component Failures

MAKING NETWORKS ROBUST TO COMPONENT FAILURES

A Dissertation Presented

by

DANIEL P. GYLLSTROM

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2014

School of Computer Science

MAKING NETWORKS ROBUST TO COMPONENT FAILURES

A Dissertation Presented

by

DANIEL P. GYLLSTROM

Approved as to style and content by:

Jim Kurose, Chair

Prashant Shenoy, Member

Deepak Ganesan, Member

Lixin Gao, Member

Lori A. Clarke, Chair
School of Computer Science

ACKNOWLEDGMENTS

The help and support of family, friends, and colleagues made this thesis possible. Starting with my wife, Sarah, thank you for your love, support, and reminding me to enjoy the fun things in life when research was a struggle. Next, thank you to my daughter Sophia. Coming home to you was (is) the best part of my a day. You forced me to create some structure to my work days (mom you were right!); I can't help but think that if you were born a few years ago I would have graduated a year or two earlier.

Thank you mom and dad for your love and constant encouragement, especially during tough times. Your support guided me through the program. I hope to be the same parent to my kids. To my sisters Amanda, Lisa, and Sara thank you for your unconditional love and support. JP, Dave, and Devon thank you sharing a beer with me whenever I asked (selfless).

To my second family – Cecile, Vladimir, Grandma Lola, Felicia, Adam, and Hannah – thank you for your care and support. Avon, CT will always be a haven of relaxation and sleep.

Thank you Jim Kurose for being my mentor, role model, and advisor. If I never stepped foot in your graduate Networking course, I would have left the program. You injected energy and curiosity to research that re-inspired me to stay in the Ph.D program. You gave me the independence I wanted and a guiding hand when necessary. I will always aspire to have your enthusiasm and approach to both teaching and research.

Thank you Leeanne Leclerc for believing in me, keeping me in order (you even reminded me to buy Sarah a Christmas present), and helping me through the program. It was always a treat to stop by your office.

Thank you to my friends and colleagues in the Networks Lab. I was lucky to have such a bright group of friends to help me work through research challenges, share a cup of coffee, and geek out on networking research.

Moe, Karl, D. Brent, Oggy, et al. thank you for being my chilled-out entertainers.

ABSTRACT

MAKING NETWORKS ROBUST TO COMPONENT FAILURES

MAY 2014

DANIEL P. GYLLSTROM

B.Sc., TRINITY COLLEGE

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Jim Kurose

In this thesis, we consider instances of component failure in the Internet and in networked cyber-physical systems, such as the communication network used by the modern electric power grid (termed the *smart grid*). We design algorithms that make these networks more robust to various component failures, including failed routers, failures of links connecting routers, and failed sensors. This thesis divides into three parts: recovery from malicious or misconfigured nodes injecting false information into a distributed system (e.g., the Internet), placing smart grid sensors to provide measurement error detection, and fast recovery from link failures in a smart grid communication network.

First, we consider the problem of malicious or misconfigured nodes that inject and spread incorrect state throughout a distributed system. Such false state can degrade

the performance of a distributed system or render it unusable. For example, in the case of network routing algorithms, false state corresponding to a node incorrectly declaring a cost of 0 to all destinations (maliciously or due to misconfiguration) can quickly spread through the network. This causes other nodes to (incorrectly) route via the misconfigured node, resulting in suboptimal routing and network congestion. We propose three algorithms for efficient recovery in such scenarios and evaluate their efficacy.

The last two parts of this thesis consider robustness in the context of the electric power grid. We study the use and placement of a sensor, called a Phasor Measurement Unit (PMU), currently being deployed in electric power grids worldwide. PMUs provide voltage and current measurements at a sampling rate orders of magnitude higher than the status quo. As a result, PMUs can both drastically improve existing power grid operations and enable an entirely new set of applications, such as the reliable integration of renewable energy resources. However, PMU applications require *correct* (addressed in thesis part 2) and *timely* (covered in thesis part 3) PMU data. Without these guarantees, smart grid operators and applications may make incorrect decisions and take corresponding (incorrect) actions.

The second part of this thesis addresses PMU measurement errors, which have been observed in practice. We formulate a set of PMU placement problems that aim to satisfy two constraints: place PMUs "near" each other to allow for measurement error detection and use the minimal number of PMUs to infer the state of the maximum number of system buses and transmission lines. For each PMU placement problem, we prove it is NP-Complete, propose a simple greedy approximation algorithm, and evaluate our greedy solutions.

In the last part of this thesis, we design algorithms for fast recovery from link failures in a smart grid communication network. We propose, design, and evaluate solutions to all three aspects of link failure recovery: (a) link failure detection,

(b) algorithms for pre-computing backup multicast trees, and (c) fast backup tree installation.

To address (a), we design link-failure detection and reporting mechanisms that use OpenFlow to detect link failures when and where they occur *inside* the network. OpenFlow is an open source framework that cleanly separates the control and data planes for use in network management and control. For part (b), we formulate a new problem, MULTICAST RECYCLING, that pre-computes backup multicast trees that aim to minimize control plane signaling overhead. We prove MULTICAST RECYCLING is at least NP-hard and present a corresponding approximation algorithm. Lastly, two control plane algorithms are proposed that signal data plane switches to install pre-computed backup trees. An optimized version of each installation algorithm is designed that finds a near minimum set of forwarding rules by sharing forwarding rules across multicast groups. This optimization reduces backup tree install time and associated control state. We implement these algorithms using the POX open-source OpenFlow controller and evaluate them using the Mininet emulator, quantifying control plane signaling and installation time.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Communication network components (routers, links, and sensors) fail. These failures can cause widespread network service disruption and outages, and potentially critical errors for network applications. *In this thesis, we examine how networks – traditional networks and networked cyber-physical systems, such as the electric power grid – can be made more robust to component failures.*

1.1 Thesis Overview

1.1.1 Component Failures in Communication Networks

We consider three separate but related problems in this thesis: node (i.e., switch or router) failure in traditional networks such as the Internet or wireless sensor networks, the failure of critical sensors that measure voltage and current throughout the smart grid, and link failures in a smart grid communication network. The term *smart grid* refers to modern and future electric power grids that automate power grid operations using sensors and wide-area communication.

For distributed network algorithms, a malicious or misconfigured node can inject and spread incorrect state throughout the distributed system. Such false state can degrade the performance of the network or render it unusable. For example, in 1997 a significant portion of Internet traffic was routed through a single misconfigured router that had spread false routing state to several Internet routers. As a result, a large potion of the Internet became inoperable for several hours [64].

Component failure in a smart grid can be especially catastrophic. For example, if smart grid sensors or links in its supporting communication network fail, smart grid applications can make incorrect decisions and take corresponding (incorrect) actions. Critical smart grid applications required to operate and manage a power grid are especially vulnerable to such failures because typically these applications have strict data delivery requirements, needing both ultra low latency and assurance that data is received correctly. In the worst case, component failure can lead to a cascade of power grid failures like the August 2003 blackout in the USA [2] and the recent power grid failures in India that left hundreds of millions of people without power [79].

1.1.2 Approaches to Making Networks More Robust to Failures

For many distributed systems, recovery algorithms operate on-demand (as opposed to being preplanned) because algorithm and system state is typically distributed throughout the network of nodes. As a result, fast convergence time and low control message overhead are key requirements for efficient recovery from component failure. In order to make the problem of on-demand recovery in a distributed system concrete, we investigate distance vector routing as an instance of this problem where nodes must recover from incorrectly injected state information. Distance vector forms the basis for many routing algorithms widely used in the Internet (e.g., BGP, a path-vector algorithm) and in multi-hop wireless networks (e.g., AODV, diffusion routing).

In the first technical chapter of this thesis, we design, develop, and evaluate three different approaches for correctly recovering from the injection of false distance vector routing state (e.g., a compromised node incorrectly claiming a distance of 0 to all destinations). Such false state, in turn, may propagate to other routers through the normal execution of distance vector routing, causing other nodes to (incorrectly) route via the misconfigured node, making this a network-wide problem. Recovery is correct if the routing tables of all nodes have converged to a global state where, for each node,

all compromised nodes are removed as a destination and no least cost path routes through a compromised node.

The second and third thesis chapters consider robustness from component failure in the context of the smart grid. Because reliability is a key requirement for the smart grid, each chapter focuses on preplanned approaches to failure recovery.

In our second thesis chapter, we study the placement of a sensor, called a Phasor Measurement Unit (PMU), currently being deployed in electric power grids worldwide. PMUs provide voltage and current measurements at a sampling rate orders of magnitude higher than the status quo. As a result, PMUs can both drastically improve existing power grid operations and enable an entirely new set of applications, such as the reliable integration of renewable energy resources. We formulate a set of problems that consider PMU measurement errors, which have been observed in practice. Specifically, we specify four PMU placement problems that aim to satisfy two constraints: place PMUs "near" each other to allow for measurement error detection and use the minimal number of PMUs to infer the state of the maximum number of system buses and transmission lines. For each PMU placement problem, we prove it is NP-Complete, propose a simple greedy approximation algorithm, and evaluate our greedy solutions.

In our final technical chapter, we design algorithms that provide fast recovery from link failures in a smart grid communication network. We propose, design, and evaluate solutions to all three aspects of link failure recovery: (a) link failure detection, (b) algorithms for pre-computing backup multicast trees, and (c) fast backup tree installation. Because this requires modifying network switches and routers, we use OpenFlow – an open standard that cleanly separates the control and data planes for use in network management and control – to program data plane forwarding using novel control plane algorithms.

To address (a), we design link-failure detection and reporting mechanisms that use OpenFlow to detect link failures when and where they occur *inside* the network. For part (b), we formulate a new problem, Multicast Recycling, that aims to pre-compute backup multicast trees that minimize control plane signaling overhead. We prove Multicast Recycling is at least NP-hard and present a corresponding approximation algorithm. Lastly, two control plane algorithms are proposed that signal data plane switches to install pre-computed backup trees. An optimized version of each installation algorithm is designed that finds a near minimum set of forwarding rules by sharing forwarding rules across multicast groups. This optimization reduces backup tree install time and control state. We implement these algorithms using the POX open-source OpenFlow controller [57] and evaluate them using the Mininet emulator [50], quantifying control plane signaling and installation time.

1.2 Thesis Contributions

The main contributions of this thesis are:

- We design, develop, and evaluate three different algorithms – 2nd-Best, Purge, and CPR – for correctly recovering from the injection of false routing state in distance vector routing. 2nd-Best performs localized state invalidation, followed by network-wide recovery using the traditional distance vector algorithm. Purge first globally invalidates false state and then uses distance vector routing to recompute distance vectors. CPR takes and stores local routing table snapshots at each router, and then uses a rollback mechanism to implement recovery. We prove the correctness of each algorithm for scenarios of single and multiple compromised nodes.

- We use simulations and analysis to evaluate 2nd-Best, Purge, and CPR in terms of control message overhead and convergence time. We find that

2ND-BEST performs poorly due to routing loops. Over topologies with fixed link weights, PURGE performs nearly as well as CPR even though our simulations and analysis assume near perfect conditions for CPR. Over more realistic scenarios in which link weights can change, we find that PURGE yields lower message complexity and faster convergence time than CPR and 2ND-BEST.

- We define four PMU placement problems, three of which are completely new, that place PMUs at a subset of electric power grid buses. Two PMU placement problems consider measurement error detection by requiring PMUs to be placed "near" each other to allow for their measurements to be cross-validated. For each PMU placement problem, we prove it is NP-Complete and propose a simple greedy approximation algorithm.

- We prove our greedy approximations for PMU placement are correct and give complexity bounds for each. Through simulations over synthetic topologies generated using real portions of the North American electric power grid as templates, we find that our greedy approximations yield results that are close to optimal: on average, within 97% of optimal. We also find that imposing our requirement of cross-validation to ensure PMU measurement error detection comes at small marginal cost: on average, only 5% fewer power grid buses are observed (covered) when PMU placements require cross-validation versus placements that do not.

- We propose, implement, and evaluate a suite of algorithms for fast recovery from link failures in a smart grid communication network: PCOUNT, BUNCHY, PROACTIVE, REACTIVE, and MERGER. PCOUNT uses OpenFlow to accurately detect link failures inside the network, rather than using slower end-to-end measurements. Then, we define a new problem, MULTICAST RECYCLING, that computes backup multicast trees with the aim of minimizing control plane sig-

naling overhead. This problem is shown to be at least NP-hard, motivating the design of an approximation, BUNCHY. Next, we design two algorithms – PROACTIVE and REACTIVE – for fast backup tree installation. PROACTIVE pre-installs backup tree forwarding rules and activates these rules after a link failure is detected, while, REACTIVE installs backup trees *after* a link a failure is detected. Lastly, we present MERGER, an algorithm that can be applied to PROACTIVE and REACTIVE to speed backup tree installations and reduce the amount of pre-installed forwarding state. MERGER does so using local optimization to create a near minimal set of forwarding rules by "merging" forwarding rules in cases where multiple multicast trees have common forwarding behavior.

- We use Mininet [50] emulations to evaluate our algorithms over communication networks based on real portions of the power grid. We find that PCOUNT provides fast and accurate link loss estimates: after sampling only 75 packets the 95% confidence interval is within 15% of the true loss probability. Additionally, we find PROACTIVE yields faster recovery than REACTIVE (REACTIVE sends up to 10 times more control messages than PROACTIVE) but at the cost of storage overhead at each switch (pre-installed backup trees can account for as much as 35% of the capacity of a conventional OpenFlow switch [21]). Finally, we observe that MERGER reduces control plane messaging and the amount of pre-installed forwarding state by a factor of 2 to 2.5 when compared to a standard multicast implementation, resulting in faster installation and manageable sized flow tables.

1.3 Thesis Outline

The rest of this thesis is organized as follows. We present algorithms for recovery from false routing state in distributed routing algorithms in Chapter 2. In Chapter 3, we formulate PMU placement problems that provide measurement error detection.

Chapter 4 presents our algorithms for fast recovery from link failures in a smart grid communication network. We conclude, in Chapter 5, with a summary and discussion of open problems emerging from this thesis.

CHAPTER 2

RECOVERY FROM FALSE ROUTING STATE IN DISTRIBUTED ROUTING ALGORITHMS

2.1 Introduction

Malicious and misconfigured nodes can degrade the performance of a distributed system by injecting incorrect state information. Such false state can then be further propagated through the system either directly in its original form or indirectly, e.g., by diffusing computations initially using this false state. In this chapter, we consider the problem of removing such false state from a distributed system.

In order to make the false-state-removal problem concrete, we investigate distance vector routing as an instance of this problem. Distance vector forms the basis for many routing algorithms widely used in the Internet (e.g., BGP, a path-vector algorithm) and in multi-hop wireless networks (e.g., AODV, diffusion routing). However, distance vector is vulnerable to compromised nodes that can potentially flood a network with false routing information, resulting in erroneous least cost paths, packet loss, and congestion. Such scenarios have occurred in practice. For example, in 1997 a significant portion of Internet traffic was routed through a single misconfigured router, rendering a large part of the Internet inoperable for several hours [64]. Distance vector currently has no mechanism to recover from such scenarios. Instead, human operators are left to manually reconfigure routers. It is in this context that we propose and evaluate automated solutions for recovery.

In this chapter, we design, develop, and evaluate three different approaches for correctly recovering from the injection of false routing state (e.g., a compromised node

incorrectly claiming a distance of 0 to all destinations). Such false state, in turn, may propagate to other routers through the normal execution of distance vector routing, making this a network-wide problem. Recovery is correct if the routing tables in all nodes have converged to a global state in which all nodes have removed each compromised node as a destination, and no node has a least cost path to any destination that routes through a compromised node.

Specifically, we develop three novel distributed recovery algorithms: 2ND-BEST, PURGE, and CPR. 2ND-BEST performs localized state invalidation, followed by network-wide recovery. Nodes directly adjacent to a compromised node locally select alternate paths that avoid the compromised node; the traditional distributed distance vector algorithm is then executed to remove remaining false state using these new distance vectors. The PURGE algorithm performs global false state invalidation by using diffusing computations to invalidate distance vector entries (network-wide) that routed through a compromised node. As in 2ND-BEST, traditional distance vector routing is then used to recompute distance vectors. CPR uses snapshots of each routing table (taken and stored locally at each router) and a rollback mechanism to implement recovery. Although our solutions are tailored to distance vector routing, we believe they represent approaches that are applicable to other diffusing distributed computations.

For each algorithm, we prove correctness, derive communication complexity bounds, and evaluate its efficiency in terms of message overhead and convergence time via simulation. Our analysis and simulations show that when considering topologies in which link weights remain fixed, CPR outperforms both PURGE and 2ND-BEST (at the cost of checkpoint memory). This is because CPR can efficiently remove all false state by simply rolling back to a checkpoint immediately preceding the injection of false routing state. In scenarios where link weights can change, PURGE outperforms CPR and 2ND-BEST. CPR performs poorly because, following rollback, it must

process the valid link weight changes that occurred since the false routing state was injected; 2ND-BEST and PURGE, however, can make use of computations subsequent to the injection of false routing state that did not depend on the false routing state. We will see, however, that 2ND-BEST performance suffers because of the so-called count-to-infinity problem.

Recovery from false routing state has similarities to the problem of recovering from malicious transactions [6, 54] in distributed databases. Our problem is also similar to that of rollback in optimistic parallel simulation [40]. However, we are unaware of any existing solutions to the problem of recovering from false routing state. A related problem to the one considered in this chapter is that of discovering misconfigured nodes. In Section 2.2, we discuss existing solutions to this problem. In fact, the output of these algorithms serve as input to the recovery algorithms proposed in this chapter.

This chapter has six sections. In Section 2.2 we define the false-state-removal problem and state our assumptions. We present our three recovery algorithms in Section 4.3. Then, in Section 2.4, we briefly state the results of our message complexity analysis, leaving the details to Appendix A.3. Section 2.5 describes our simulation study. We detail related work in Section 2.6 and conclude the chapter in Section 2.7. The research described here has been published in [34].

2.2 Problem Formulation

We consider distance vector routing [11] over arbitrary network topologies. We model a network as an undirected graph, $G = (V, E)$, with a link weight function $w : E \to \mathbb{N}$. [1] Each node, v, maintains the following state as part of distance vector:

[1] Recovery is simple with link state routing: each node uses its complete topology map to compute new least cost paths that avoid all compromised nodes. Thus we do not consider link state routing in this chapter.

a vector of all adjacent nodes ($adj(v)$), a vector of least cost distances to all nodes in G ($\overrightarrow{min}_v$), and a *distance matrix* that contains distances to every node in the network via each adjacent node ($dmatrix_v$).

For simplicity, we present our recovery algorithms in the case of a single compromised node. We describe the necessary extensions to handle multiple compromised nodes in Section 2.3.5. We assume that the identity of the compromised node is provided by a different algorithm, and thus do not consider this problem in this thesis. Examples of such algorithms include [25, 26, 28, 61, 65, 71]. Specifically, we assume that at time t_b, this algorithm is used to notify all neighbors of the compromised node. Let t' be the time the node was compromised.

For each of our algorithms, the goal is for all nodes to recover "correctly": all nodes should remove the compromised nodes as a destination and find new least cost distances that do not use a compromised node. If the network becomes disconnected as a result of removing the compromised node, all nodes need only compute new least cost distances to all other nodes within their connected component.

For simplicity, let $\overline{v}$ denote the compromised node, let $\overrightarrow{old}$ refer to $\overrightarrow{min}_{\overline{v}}$ before $\overline{v}$ was compromised, and let $\overrightarrow{bad}$ denote $\overrightarrow{min}_{\overline{v}}$ after $\overline{v}$ has been compromised. Intuitively, $\overrightarrow{old}$ and $\overrightarrow{bad}$ are snapshots of the compromised node's least cost vector taken at two different timesteps: $\overrightarrow{old}$ marks the snapshot taken before $\overline{v}$ was compromised and $\overrightarrow{bad}$ represents a snapshot taken after $\overline{v}$ was compromised.

Table 2.1 summarizes the notation used in this chapter.

Abbreviation	Meaning
$\overrightarrow{min}_i$	node i's the least cost vector
$dmatrix_i$	node i' distance matrix
DV	Distance Vector
t_b	time the compromised node is detected
t'	time the compromised node was compromised
$\overrightarrow{bad}$	compromised node's least cost vector at and after t
$\overrightarrow{old}$	compromised node's least cost vector at and before t'
$\overline{v}$	compromised node
$adj(v)$	nodes adjacent to v in G'

Table 2.1. Table of abbreviations.

2.3 Recovery Algorithms

In this section we propose three new recovery algorithms: 2ND-BEST, PURGE, and CPR. With one exception, the input and output of each algorithm is the same. [2]

- **Input:** Undirected graph, $G = (V, E)$, with weight function $w : E \to \mathbb{N}$. $\forall v \in V$, $\overrightarrow{min}_v$ and $dmatrix_v$ are computed (using distance vector). Also, each $v \in adj(\overline{v})$ is notified that $\overline{v}$ was compromised.

- **Output:** Undirected graph, $G' = (V', E')$, where $V' = V - \{\overline{v}\}$, $E' = E - \{(\bar{v}, v_i) \mid v_i \in adj(\bar{v})\}$, and link weight function $w : E \to \mathbb{N}$. $\overrightarrow{min}_v$ and $dmatrix_v$ are computed via the algorithms discussed below $\forall v \in V'$.

Before we describe each recovery algorithm, we outline a preprocessing procedure common to all three recovery algorithms. Correctness proofs for 2ND-BEST, PURGE, and CPR can be found in Appendix A.2.

[2] Additionally, as input CPR requires that each $v \in adj(\overline{v})$ is notified of the time, t', in which $\overline{v}$ was compromised.

2.3.1 Preprocessing

All three recovery algorithms share a common preprocessing procedure. The procedure removes $\overline{v}$ as a destination and finds the node IDs in each connected component. This is implemented using diffusing computations [23] initiated at each $v \in adj(\overline{v})$. A diffusing computation is a distributed algorithm started at a source node which grows by sending queries along a spanning tree, constructed simultaneously as the queries propagate through the network. When the computation reaches the leaves of the spanning tree, replies travel back along the tree towards the source, causing the tree to shrink. The computation eventually terminates when the source receives replies from each of its children in the tree.

In our case, each diffusing computation message contains a vector of node IDs. When a node receives a diffusing computation message, the node adds its ID to the vector and removes $\overline{v}$ as a destination. At the end of the diffusing computation, each $v \in adj(\overline{v})$ has a vector that includes all nodes in v's connected component. Finally, each $v \in adj(\overline{v})$ broadcasts the vector of node IDs to all nodes in their connected component. In the case where removing $\overline{v}$ partitions the network, each node will only compute shortest paths to nodes in the vector.

Consider the example in Figure 2.1 where $\overline{v}$ is the compromised node. When i receives the notification that $\overline{v}$ has been compromised, i removes $\overline{v}$ as a destination and then initiates a diffusing computation. i creates a vector and adds its node ID to the vector. i sends a message containing this vector to j and k. Upon receiving i's message, j and k both remove $\overline{v}$ as a destination and add their own ID to the message's vector. Finally, l and d receive a message from j and k, respectively. l and d add their node own ID to the message's vector and remove $\overline{v}$ as a destination. Then, l and d send an ACK message back to j and k, respectively, with the complete list of node IDs. Eventually when i receives the ACKs from j and k, i has a complete

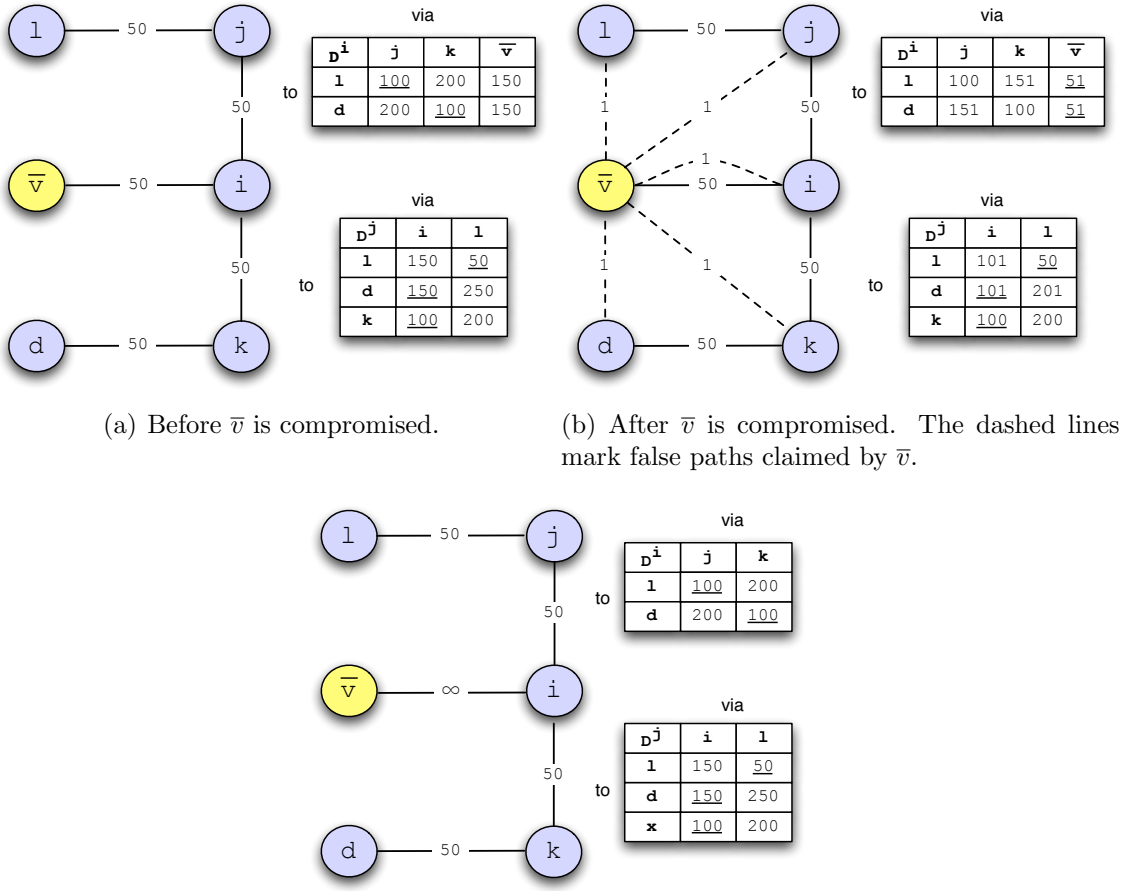
list of nodes in its connected component. Finally, i broadcasts the vector of node IDs in its connected component.

2.3.2 The 2nd Best Algorithm

2ND-BEST invalidates state locally and then uses distance vector to implement network-wide recovery. Following the preprocessing described in Section 2.3.1, each neighbor of the compromised node locally invalidates state by selecting the least cost pre-existing alternate path that does not use the compromised node as the first hop. The resulting distance vectors trigger the execution of traditional distance vector to remove the remaining false state. Algorithm A.1.1 in the Appendix gives a complete specification of 2ND-BEST.

We trace the execution of 2ND-BEST using the example in Figure 2.1. In Figure 2.1(b), i uses $\overline{v}$ to reach nodes l and d. j uses i to reach all nodes except l. Notice that when j uses i to reach d, it transitively uses $\overrightarrow{bad}$ (e.g., uses path $j - i - \overline{v} - d$ to d). After the preprocessing completes, i selects a new neighbor to route through to reach l and d by finding its new smallest distance in $dmatrix_i$ to these destinations: i selects the routes via j to l with a cost of 100 and i picks the route via k to reach d with cost of 100. (No changes are required to route to j and k because i uses its direct link to these two nodes). Then, using traditional distance vector i sends $\overrightarrow{min}_i$ to j and k. When j receives $\overrightarrow{min}_i$, j must modify its distance to d because $\overrightarrow{min}_i$ indicates that i's least cost to d is now 100. j's new distance value to d becomes 150, using the path $j - i - k - l$. j then sends a message sharing $\overrightarrow{min}_j$ with its neighbors. From this point, recovery proceeds according by using traditional distance vector.

2ND-BEST is simple and makes no synchronization assumptions. However, 2ND-BEST is vulnerable to the count-to-infinity problem. Because each node only has local information, the new shortest paths may continue to use $\overline{v}$. For example, if $w(k, d) = 400$ in Figure 2.1, a count-to-infinity scenario would arise. After notification



(a) Before $\overline{v}$ is compromised.

(b) After $\overline{v}$ is compromised. The dashed lines mark false paths claimed by $\overline{v}$.

(c) After recovery.

Figure 2.1. Three snapshots of a graph, G, where $\overline{v}$ is the compromised node. Parts of i and j's distance matrix are displayed to the right of each sub-figure. The least cost values are underlined.

of $\overline{v}$'s compromise, i would select the route via j to reach d with cost 151 (by consulting $dmatrix_i$), using a path that does not actually exist in G ($i - j - i - \overline{v} - d$), since j has removed $\overline{v}$ as a neighbor. When i sends $\overrightarrow{min}_i$ to j, j selects the route via i to d with cost 201. Again, the path $j - i - j - i - \overline{v} - d$ does not exist. In the next iteration, i picks the route via j having a cost of 251. This process continues until each node finds their correct least cost to d. We will see in our simulation study that the count-to-infinity problem can incur significant message and time costs.

2.3.3 The Purge Algorithm

PURGE globally invalidates all false state using a diffusing computation and then uses distance vector to compute new distance values that avoid all invalidated paths. Recall that diffusing computations preserve the decentralized nature of distance vector. The diffusing computation is initiated at the neighbors of $\overline{v}$ because only these nodes are aware if $\overline{v}$ is used an intermediary node. The diffusing computations spread from $\overline{v}$'s neighbors to the network edge, invalidating false state at each node along the way. Then ACKs travel back from the network edge to the neighbors of $\overline{v}$, indicating that the diffusing computation is complete. See Algorithm A.1.2 and A.1.3 in the Appendix for a complete specification of this diffusing computation.

Next, PURGE uses distance vector to recompute least cost paths invalidated by the diffusing computations. In order to initiate the distance vector computation, each node is required to send a message after diffusing computations complete, even if no new least cost is found. Without this step, distance vector may not correctly compute new least cost paths invalidated by the diffusing computations. For example, consider the following the scenario when the diffusing computations complete: a node i and all of i's neighbors have least cost of ∞ to destination node a. Without forcing i and its neighbors to send a message after the diffusing computations complete, neither i nor i's neighbors may never update their least cost to a because they may never receive a non-∞ cost to a.

In Figure 2.1, the diffusing computation executes as follows. First, i sets its distance to l and d to ∞ (thereby invalidating i's path to l and d) because i uses $\overline{v}$ to route these nodes. Then, i sends a message to j and k containing l and d as invalidated destinations. When j receives i's message, j checks if it routes via i to reach l or d. Because j uses i to reach d, j sets its distance estimate to d to ∞. j does not modify its least cost to l because j does not route via i to reach l. Next, j sends a message that includes d as an invalidated destination. l performs the same

steps as j. After this point, the diffusing computation ACKs travel back towards i. When i receives an ACK, the diffusing computation is complete. At this point, i needs to compute new least costs to node l and d because i's distance estimates to these destinations are ∞. i uses $dmatrix_i$ to select its new route to l (which is via j) and uses $dmatrix_i$ to find i's new route to d (which is via k). Both new paths have cost 100. Finally, i sends $\overrightarrow{min}_i$ to its neighbors, triggering the execution of distance vector to recompute the remaining distance vectors.

Note that a consequence of the diffusing computation is that not only is all $\overrightarrow{bad}$ state deleted, but all $\overrightarrow{old}$ state as well. Consider the case when $\overline{v}$ is detected before node i receives $\overrightarrow{bad}$. It is possible that i uses $\overrightarrow{old}$ to reach a destination, d. In this case, the diffusing computation will set i's distance to d to ∞.

An advantage of PURGE is that it operates without the need for any clock synchronization. We will find that CPR, unlike PURGE, either requires extra computation to maintain logical clocks or assumes clocks are loosely synchronized. Also, PURGE's diffusing computations ensure that the count-to-infinity problem does not occur by removing false state from the entire network. However, globally invalidating false state can be wasteful if valid alternate paths are locally available.

2.3.4 The CPR Algorithm

CPR[3] is our third and final recovery algorithm. Unlike 2ND-BEST and PURGE, CPR only requires that clocks across different nodes be loosely synchronized i.e. the maximum clock offset between any two nodes is assumed to be bounded. For ease of explanation, we describe CPR as if the clocks at different nodes are perfectly synchronized. Extensions to handle loosely synchronized clocks should be clear. Accordingly, we assume that all neighbors of $\overline{v}$, are notified of the time, t', at which $\overline{v}$ was com-

[3]The name is an abbreviation for **C**heck**P**oint and **R**ollback.

promised. At the end of this section we comment on how the clock synchronization requirement assumption can be dropped by using logical clocks.

For each node, $i \in G$, CPR adds a time dimension to $\overrightarrow{min}_i$ and $dmatrix_i$, which CPR then uses to locally archive a complete history of values. Once the compromised node is discovered, the archive allows the system to rollback to a system snapshot from a time before $\overline{v}$ was compromised. From this point, CPR needs to remove $\overline{v}$ and $\overrightarrow{old}$ and update stale distance values resulting from link weight changes. We describe each algorithm step in detail.

Step 1: Create a $\overrightarrow{min}$ and *dmatrix* archive. We define a *snapshot* of a data structure to be a copy of all current distance values along with a timestamp. [4] The timestamp marks the time at which that set of distance values start being used. $\overrightarrow{min}$ and *dmatrix* are the only data structures that need to be archived. This approach is similar to ones used in temporal databases [41, 55].

Our distributed archive algorithm is quite simple. Each node has a choice of archiving at a given frequency (e.g., every m timesteps) or after some number of distance value changes (e.g., each time a distance value changes). Each node must choose the same option, which is specified as an input parameter to CPR. A node archives independently of all other nodes. A side effect of independent archiving, is that even with perfectly synchronized clocks, the union of all snapshots may not constitute a globally consistent snapshot. For example, a link weight change event may only have propagated through part of the network, in which case the snapshot for some nodes will reflect this link weight change (i.e., among nodes that have learned of the event) while for other nodes no local snapshot will reflect the occurrence of this event. We will see that a globally consistent snapshot is not required for correctness.

[4] In practice, we only archive distance values that have changed. Thus each distance value is associated with its own timestamp.

Step 2: Rolling back to a valid snapshot. Rollback is implemented using diffusing computations. Neighbors of the compromised node independently select a snapshot to roll back to, such that the snapshot is the most recent one taken before t'. Each such node, i, rolls back to this snapshot by restoring the $\overrightarrow{min}_i$ and $dmatrix_i$ values from the snapshot. Then, i initiates a diffusing computation to inform all other nodes to do the same. If a node has already rolled back and receives an additional rollback message, it is ignored. (Note that this rollback algorithm ensures that no reinstated distance value uses $\overrightarrow{bad}$ because every node rolls back to a snapshot with a timestamp less that t'). A pseudo-code specification of this rollback algorithm can be found in the Appendix (Algorithm A.1.4).

Step 3: Steps after rollback. After Step 2 completes, the algorithm in Section 2.3.1 is executed. There are two issues to address. First, some nodes may be using $\overrightarrow{old}$. Second, some nodes may have stale state as a result of link weight changes that occurred during $[t', t_b]$ and consequently are not reflected in the snapshot. To resolve these issues, each neighbor, i, of $\overline{v}$, sets its distance to $\overline{v}$ to ∞ and then selects new least cost values that avoid the compromised node, triggering the execution of distance vector to update the remaining distance vectors. That is, for any destination, d, where i routes via $\overline{v}$ to reach d, i uses $dmatrix_i$ to find a new least cost to d. If a new least costs value is used, i sends a distance vector message to its neighbors. Otherwise, i sends no message. Messages sent trigger the execution of distance vector.

During the execution of distance vector, each node uses the most recent link weights of its adjacent links. Thus, if the same link changes cost multiple times during $[t', t_b]$, we ignore all changes but the most recent one. Algorithm A.1.5 specifies Step 3 of CPR.

In the example from Figure 2.1, the global state after rolling back is nearly the same as the snapshot depicted in Figure 2.1(c): the only difference between the actual system state and that depicted in Figure 2.1(c) is that in the former $(i,\overline{v}) = 50$ rather

than ∞. Step 3 in CPR makes this change. Because no nodes use $\overrightarrow{old}$, no other changes take place.

Rather than using an iterative process to remove false state (like in 2ND-BEST and PURGE), CPR does so in one diffusing computation. However, CPR incurs storage overhead resulting from periodic snapshots of $\overrightarrow{min}$ and $dmatrix$. Also, after rolling back, stale state may exist if link weight changes occur during $[t', t_b]$. This can be expensive to update. Finally, unlike PURGE and 2ND-BEST, CPR requires loosely synchronized clocks because without a bound on the clock offset, nodes may rollback to highly inconsistent local snapshots. Although correct, this would severely degrade CPR performance.

Using Logical Clock Timestamps. We can use Lamport's clock algorithm [49] to assign timestamps based on logical, rather than physical, clocks. This allows us to drop the inconvenient assumption of loosely synchronized clocks. Here we briefly outline how the stated CPR algorithm can be modified to use Lamport timestamps to create and restore system snapshots.

First, CPR is modified to create network-wide snapshots using diffusing computations instead of each node creating snapshots independently. Here each node records the logical timestamp when creating a checkpoint. The logical clock values are determined using the "happened before" relation defined by Lamport [49], where we limit events to be messages sent and received, and by piggybacking each node's logical clock value with each message it sends.

The second change to CPR is in how each node determines the snapshot to restore during the roll back process (i.e., Step 2 above). Starting with each $i \in adj(\overline{v})$, i determines the logical timestamp of the snapshot it wishes to restore. This requires that the detection algorithm specifies either $\overrightarrow{bad}$ or the logical time $\overline{v}$ was compromised. In the latter case, finding the appropriate snapshot to restore is straightforward. If only $\overrightarrow{bad}$ is provided, i must additionally record each $\overrightarrow{min}$ vector received (as a part

of standard distance vector messaging) along with the corresponding Lamport timestamp. By doing so, this archive can be searched to find the logical timestamp in which $\overrightarrow{bad}$ was first received at i. Let t_l denote this timestamp. Next, i initiates a diffusing computation instructing all other nodes to roll back to their most recent snapshot taken before t_l. After this point, the diffusing computations continue as described in Step 2 above and Step 3 remains unchanged.

Rolling back using logical timestamps does not guarantee that all $\overrightarrow{bad}$ state is removed because Lamport timestamps only provide partial ordering of events. With logical clocks, CPR can be thought of as a best-effort approach to quickly removing false routing state by rolling back in time. CPR is still correct with logical clocks for the reasons described in its correctness proof (Corollary A.6).

2.3.5 Multiple Compromised Nodes

Here we detail the necessary changes to each of our recovery algorithms when multiple nodes are compromised. Since we make the same changes to all three algorithms, we do not refer to a specific algorithm in this section. Let $\overline{V}$ refer to the set of nodes compromised at time t'.

In the case where multiple nodes are simultaneously compromised, each recovery algorithm is modified such that for each $\overline{v} \in \overline{V}$, all $adj(\overline{v})$ are notified that $\overline{v}$ was compromised. From this point, the changes to each algorithm are straightforward. For example, the diffusing computations described in Section 2.3.1 are initiated at the neighbor nodes of each node in $\overline{V}$. [5]

More changes are required to handle the case where an additional node is compromised during the execution of a recovery algorithm. Specifically, when another node is compromised, $\overline{v}_2$, we make the following change to the distance vector computa-

[5]For CPR, t' is set to the time the first node is compromised.

tion of each recovery algorithm. [6] If a node, i, receives a distance vector message which includes a distance value to destination $\overline{v}_2$, then i ignores said distance value and processes the remaining distance values (if any exist) to all other destinations (e.g., where $d \neq \overline{v}_2$) normally. If the message contains no distance information for any other destination $d \neq \overline{v}_2$, then i ignores the message. Because $\overline{v}_2$'s compromise triggers a diffusing computation to remove $\overline{v}_2$ as a destination, each node eventually learns the identity of $\overline{v}_2$, thereby allowing the node execute the specified changes to distance vector.

Without this change it is possible that the recovery algorithm will not terminate. Consider the case of two compromised nodes, $\overline{v}_1$ and $\overline{v}_2$, where $\overline{v}_2$ is compromised during the recovery triggered by $\overline{v}_1$'s compromise. In this case, two executions of the recovery algorithm are triggered: one when $\overline{v}_1$ is compromised and the other when $\overline{v}_2$ is compromised. Recall that all three recovery algorithms set all link weights to $\overline{v}_1$ to ∞ (e.g., $(v_i, \overline{v}_1) = \infty, \forall v_i \in adj(\overline{v}_1)$). If the first distance vector execution triggered by $\overline{v}_1$'s compromise is not modified to terminate least cost computations to $\overline{v}_2$, the distance vector step of the recovery algorithm would never complete because the least cost to $\overline{v}_2$ is ∞.

2.4 Analysis of Algorithms

Here we summarize the results from our analysis, the detailed proofs can be found in Appendix A.3. Using a synchronous communication model, we derive communication complexity bounds for each algorithm. Our analysis assumes: a graph with unit link weights of 1, that only a single node is compromised, and that the compromised node falsely claims a cost of 1 to every node in the graph. For graphs with fixed link weights, we find that the communication complexity of all three algorithms is

[6] Recall that each of our recovery algorithms use distance vector to complete their computation.

bounded above by $O(mnd)$ where d is the diameter, n is the number of nodes, and m the maximum out-degree of any node.

In the second part of our analysis, we consider graphs where link weights can change. Again, we assume a graph with unit link weights of 1 and a single compromised node that declares a cost of 1 to every node. Additionally, we let link weights increase between the time the malicious node is compromised and the time at which error recovery is initiated. We assume that across all network links, the total increase in link weights is w units. We find that CPR incurs additional overhead (not experienced by 2ND-BEST and PURGE) because CPR must update stale state after rolling back. 2ND-BEST and PURGE avoid the issue of stale state because neither algorithm rolls back in time. As a result, the message complexity for 2ND-BEST and PURGE is still bounded by $O(mnd)$ when link weights can change, while CPR is not. CPR's upper bound becomes $O(mnd) + O(wn^2)$.

2.5 Simulation Study

In this section, we use simulations to characterize the performance of each of our three recovery algorithms in terms of message and time overhead. Our goal is to illustrate the relative performance of our recovery algorithms over different topology types (e.g., Erdös-Rényi graphs, Internet-like graphs) and different network conditions (e.g., fixed link weighs, changing link weights).

We build a custom simulator with a synchronous communication model as described in Section A.3. All algorithms are deterministic under this communication model. The synchronous communication model, although simple, yields interesting insights into the performance of each of the recovery algorithms. We find the same trends hold when using a more general asynchronous communication model but, for ease of exposition, we only present the results found using synchronous communication.

We simulate the following scenario: [7]

1. Before t', $\forall v \in V$ $\overrightarrow{min}_v$ and $dmatrix_v$ are correctly computed.

2. At time t', $\overline{v}$ is compromised and advertises a $\overrightarrow{bad}$ (a vector with a cost of 1 to *every* node in the network) to its neighboring nodes.

3. $\overrightarrow{bad}$ spreads for a specified number of hops (this varies by simulation). Variable k refers to the number of hops that $\overrightarrow{bad}$ has spread.

4. At time t, some node $v \in V$ notifies all $v \in adj(\overline{v})$ that $\overline{v}$ was compromised. [8]

The message and time overhead are measured in step (4) above. The pre-computation described in Section 2.3.1, is not counted towards message and time overhead because the same exact pre-computation steps are executed by all three recovery algorithms. We describe our simulation scenario for multiple compromised nodes in Section 2.5.1.4.

2.5.1 Simulations using Graphs with Fixed Link Weights

In the next five simulations, we evaluate our recovery algorithms over different topology types in the case where link weights remain fixed.

2.5.1.1 Simulation 1: Erdös-Rényi Graphs with Fixed Unit Link Weights

We start with a simplified setting and consider Erdös-Rényi graphs with parameters n and p. n is the number of graph nodes and p is the probability that link (i, j) exists where $i, j \in V$. The link weight of each edge in the graph is set to 50. We iterate over different values of k. For each k, we generate an Erdös-Rényi graph,

---

[7] In Section 2.5.1.4 we consider the case of multiple compromised nodes. In that simulation we modify our simulation scenario to consider a set of compromised nodes, $\overline{V}$, instead of $\overline{v}$.

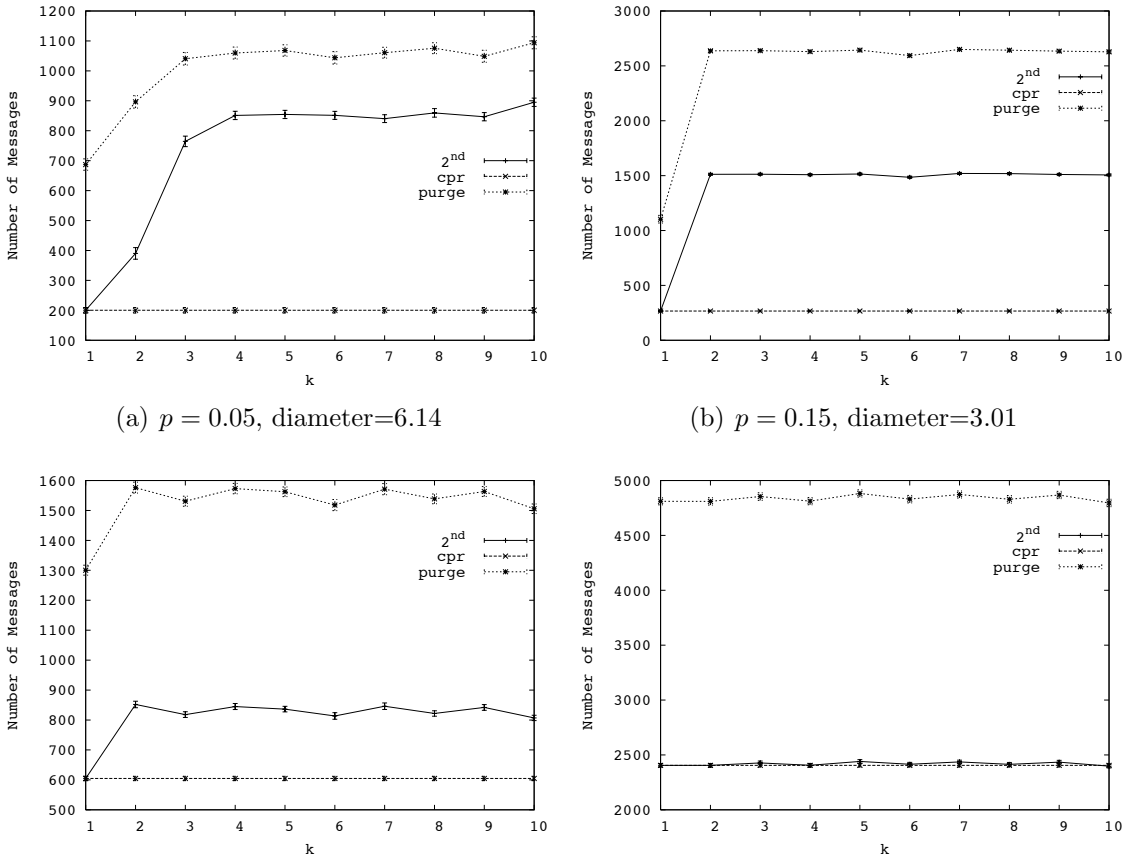
[8] For CPR this node also indicates the time, t', $\overline{v}$ was compromised.

$G = (V, E)$, with parameters n and p. Then we select a $\overline{v} \in V$ uniformly at random and simulate the scenario described above, using $\overline{v}$ as the compromised node. In total we sample 20 unique nodes for each G. We set $n = 100$, $p = \{0.05, 0.15, 0.25, 0.50\}$, and let $k = \{1, 2, ...10\}$. Each data point is an average over 600 runs (20 runs over 30 topologies). We then plot the 90% confidence interval.

For each of our recovery algorithms, Figure 2.2 shows the message overhead for different values of k. We conclude that CPR outperforms PURGE and 2ND-BEST across all topologies. CPR performs well because $\overrightarrow{bad}$ is removed using a single diffusing computation, while the other algorithms remove $\overrightarrow{bad}$ state through distance vector's iterative process. CPR's global state after rolling back is almost the same as the final recovered state.

2ND-BEST recovery can be understood as follows. By Corollary A.9 and A.10 in Section A.3.1, distance values increase from their initial value until they reach their final (correct) value. Any intermediate, non-final, distance value uses $\overrightarrow{bad}$ or $\overrightarrow{old}$. Because $\overrightarrow{bad}$ and $\overrightarrow{old}$ no longer exist during recovery, these intermediate values must correspond to routing loops. Table 2.2 shows that there are few pairwise routing loops during 2ND-BEST recovery in the network scenarios generated in Simulation 1, indicating that 2ND-BEST distance values quickly count up to their final value. [9] Although no pairwise routing loops exist during PURGE recovery, PURGE incurs overhead in performing network-wide state invalidation. Roughly, 50% of PURGE's messages come from these diffusing computations. For these reasons, PURGE has higher message overhead than 2ND-BEST.

[9] We compute this metric as follows. After each simulation timestep, we count all pairwise routing loops over all source-destination pairs and then sum all of these values.



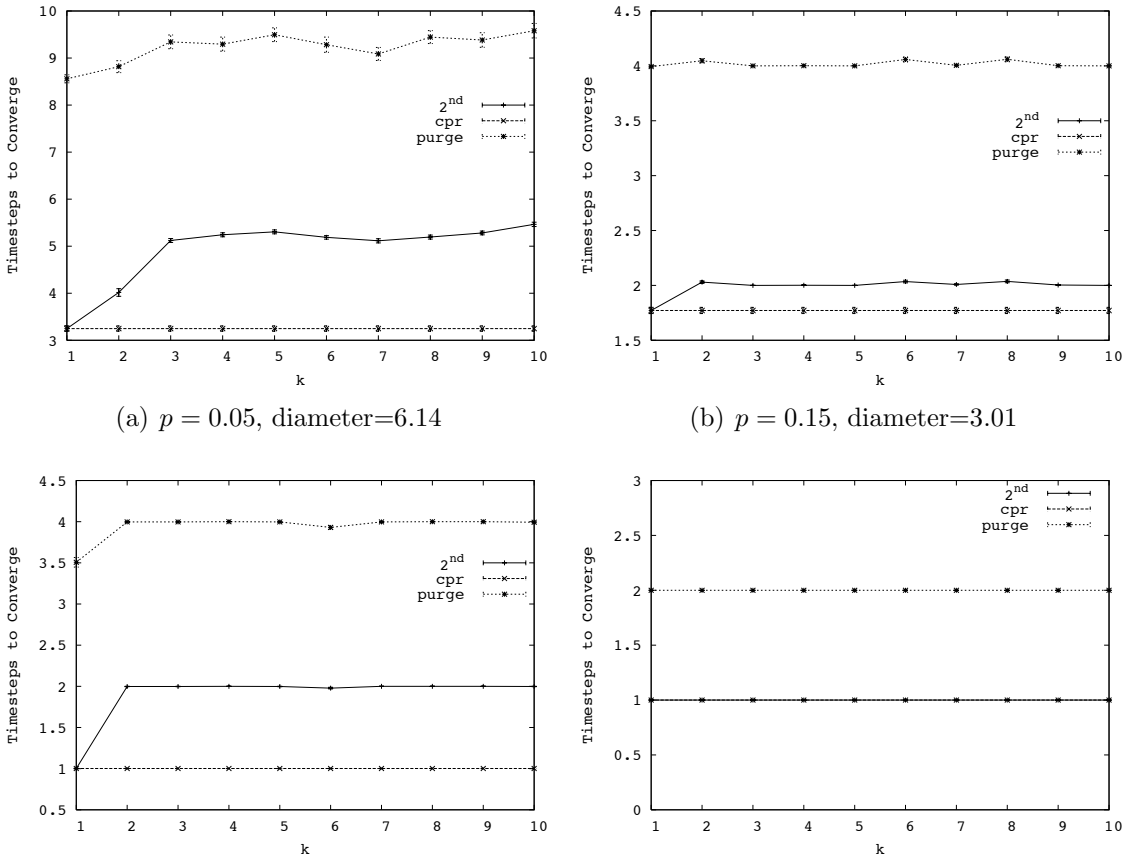
(a) $p = 0.05$, diameter=6.14

(b) $p = 0.15$, diameter=3.01

(c) $p = 0.25$, diameter=2.99

(d) $p = 0.50$, diameter=2

Figure 2.2. Simulation 1: message overhead as a function of the number of hops false routing state has spread from the compromised node (k), over Erdös-Rényi graphs with fixed link weights. Note the y-axes have different scales.



(a) $p = 0.05$, diameter=6.14

(b) $p = 0.15$, diameter=3.01

(c) $p = 0.25$, diameter=2.99

(d) $p = 0.50$, diameter=2

Figure 2.3. Simulation 1: time overhead as a function of the number of hops false routing state has spread from the compromised node (k), over Erdös-Rényi graphs with fixed link weights. Note the different scales of the y-axes.

	$k=1$	$k=2$	$k=3$	$k=4-10$
$p=0.05$	0	14	87	92
$p=0.15$	0	7	8	9
$p=0.25$	0	0	0	0
$p=0.50$	0	0	0	0

Table 2.2. Average number pairwise routing loops for 2ND-BEST in Simulation 1.

	$k=1$	$k=2$	$k=3$	$k=4-10$
$p=0.05$	554	1303	9239	12641
$p=0.15$	319	698	5514	7935
$p=0.25$	280	446	3510	5440
$p=0.50$	114	234	2063	2892

Table 2.3. Average number pairwise routing loops for 2ND-BEST in Simulation 2.

Figure 2.3 shows the time overhead for the same p values. The trends for time overhead match the trends we observe for message overhead. [10]

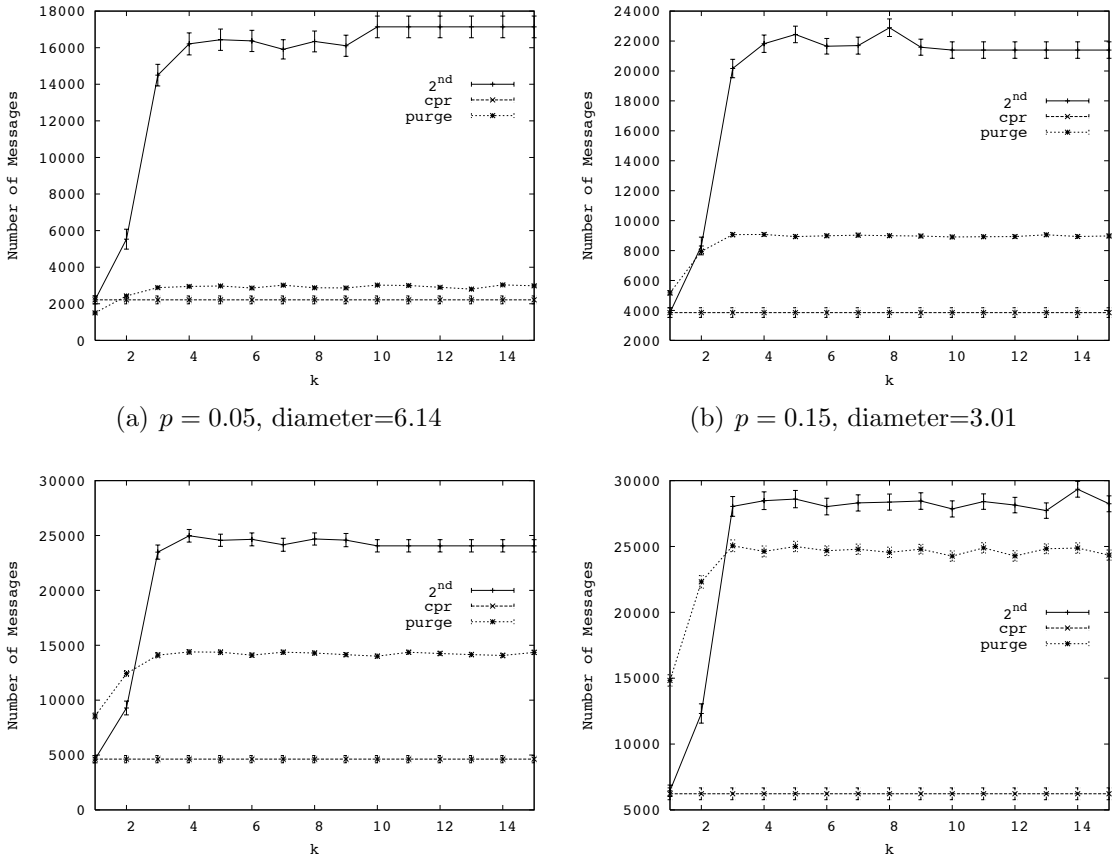
PURGE and 2ND-BEST message overhead increases with larger k. Larger k imply that false state has propagated further in the network, implying more paths to repair, and therefore increased messaging. For values of k greater than a graph's diameter, the message overhead remains constant, as expected.

2.5.1.2 Simulation 2: Erdös-Rényi Graphs with Fixed but Randomly Chosen Link Weights

The simulation setup is identical to Simulation 1 with one exception: link weights are selected uniformly at random between $[1, n]$, rather than using a fixed link weight of 50.

Figure 2.4 show the message overhead for different k where $p = \{0.05, 0.15, 0.25, 0.50\}$. In striking contrast to Simulation 1, PURGE outperforms 2ND-BEST for most values

[10]For the remaining simulations, we omit time overhead plots because time overhead follows the same trends as message overhead.



(a) $p = 0.05$, diameter=6.14

(b) $p = 0.15$, diameter=3.01

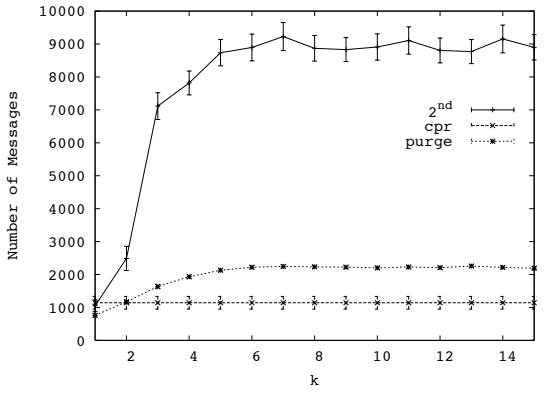
(c) $p = 0.25$, diameter=2.99

(d) $p = 0.50$, diameter=2

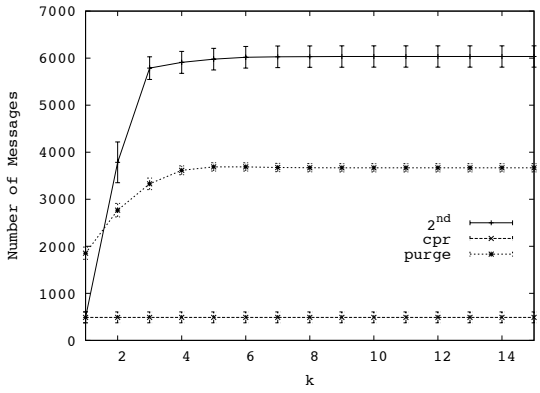
Figure 2.4. Simulation 2: message overhead as a function of k, the number of hops false routing state has spread from the compromised node. Erdös-Rényi graph with link weights selected randomly from $[1, 100]$ are used. Note the different scales of the y-axes.

of k. 2ND-BEST performs poorly because the count-to-infinity problem: Table 2.3 shows the large average number of pairwise routing loops in this simulation, an indicator of the occurrence of count-to-infinity problem. In the few cases (e.g., $k = 1$ for $p = 0.15$, $p = 0.25$ and $p = 0.50$) that 2ND-BEST performs better than PURGE, 2ND-BEST has few routing loops.

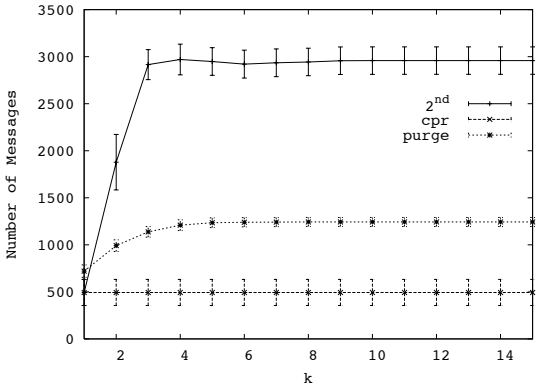
No routing loops are found with PURGE. CPR performs well for the same reasons described in Section 2.5.1.1.



(a) GT-ITM, $n = 156$, diameter=14.133



(b) Rocketfuel 6461, $n = 141$, diameter=8



(c) Rocketfuel 3867, $n = 79$, diameter=10

Figure 2.5. Simulation 3: Internet-like graph message overhead as a function of k, the number of hops false routing state has spread from the compromised node.

In addition, we counted the number of epochs in which at least one pairwise routing loop existed. For 2ND-BEST (across all topologies), on average, all but the last three timesteps had at least one routing loop. This suggests that the count-to-infinity problem dominates the cost for 2ND-BEST.

2.5.1.3 Simulation 3: Internet-like Topologies

Thus far, we studied the performance of our recovery algorithms over Erdös-Rényi graphs, which have provided us with useful intuition about the performance of each algorithm. In this simulation, we simulate our algorithms over Internet-like topologies downloaded from the Rocketfuel website [3] and generated using GT-ITM [1]. The

Rocketfuel topologies have inferred edge weights. For each Rocketfuel topology, we let each node be the compromised node and average over all of these cases for each value of k. For GT-ITM, we used the parameters specified in Heckmann et al [37] for the 154-node AT&T topology described in Section 4 of [37]. For the GT-ITM topologies, we use the same criteria specified in Simulation 1 to generate each data point.

The results, shown in Figure 2.5, follow the same pattern as in Simulation 2. In the cases where 2ND-BEST performs poorly, the count-to-infinity problem dominates the cost, as evidenced by the number of pairwise routing loops. In the few cases that 2ND-BEST performs better than PURGE, there are few pairwise routing loops.

2.5.1.4 Simulation 4: Multiple Compromised Nodes

In this simulation, we evaluate our recovery algorithms when multiple nodes are compromised. Our simulation setup is different from what we have used to this point: we fix $k = \infty$ and vary the number of compromised nodes. Specifically, for each topology we create $m = \{1, 2, ..., 15\}$ compromised nodes, each of which is selected uniformly at random (without replacement). We then simulate the scenario described at the start of Section 2.5 with one modification: m nodes are compromised during $[t', t' + 10]$. The simulation is setup so that the outside algorithm identifies all m compromised node at time t. After running the simulation for all possible values for m, we generate a new topology and repeat the above procedure. We continue sampling topologies until the 90% confidence interval for message overhead falls within 10% of the mean message overhead.

First, we perform this simulation using Erdös-Rényi graphs with fixed link weights. The message overhead results are shown in Figure 2.6(a) for $p = 0.05$ and $n = 100$.

[11] The relative performance of the three algorithms is consistent with the results from Simulation 1, in which we had a single compromised node. As in Simulation 1, 2ND-BEST and CPR have few pairwise routing loops (Figure 2.6(b)). In fact, there is more than an order of magnitude fewer pairwise routing loops in this simulation when compared to the results for the same simulation scenario of m compromised nodes using Erdös-Rényi graphs with random link weights (Figure 2.7(b)). Few routing loops imply that 2ND-BEST and CPR (after rolling back) quickly count up to correct least costs. In contrast, PURGE has high message overhead because PURGE globally invalidates false state before computing new least cost paths, rather than directly using alternate paths that are immediately available when recovery begins at time t.

2ND-BEST and PURGE message overhead are nearly constant for $m \geq 8$ because at that point $\overrightarrow{bad}$ state has saturated G. Figure 2.6 shows the number of least cost paths, per node, that use $\overrightarrow{bad}$ or $\overrightarrow{old}$ at time t (e.g., after $\overrightarrow{bad}$ state has propagated k hops from $\overline{v}$). The number of least cost paths that use $\overrightarrow{bad}$ is nearly constant for $m \geq 8$.

In contrast, CPR message overhead increases with the number of compromised nodes. After rolling back, CPR must remove all compromised nodes and all stale state (e.g., $\overrightarrow{old}$) associated with each $\overline{v}$. As seen in Figure 2.6(c), the amount of $\overrightarrow{old}$ state increases as the number of compromised nodes increase.

Next, we perform the same simulation using Erdös-Rényi graphs with with link weights selected uniformly at random from $[1, 100]$. We only show the results for $p = .05$ and $n = 100$ because the trends are consistent for other values of p. The message overhead results for this simulation are shown in Figure 2.7(a). PURGE performs best because, unlike 2ND-BEST and CPR, PURGE does not suffer from the

[11] We do not include the results for $p = \{0.15, 0.25, 0.50\}$ because they are consistent with the results for $p = 0.05$.

count-to-infinity problem. Below, we explain the performance of each algorithm in detail.

Consistent with Simulation 2 and 3, 2ND-BEST performs poorly because of the count-to-infinity problem. Figure 2.7(b) shows that a significant number of pairwise routing loops occur during 2ND-BEST recovery. 2ND-BEST message overhead remains constant when $m \geq 6$ because at this point $\overrightarrow{bad}$ state has saturated the network. Figure 2.7(c) confirms this: the number of effected least cost paths remains constant (at 80) for all $m \geq 6$.

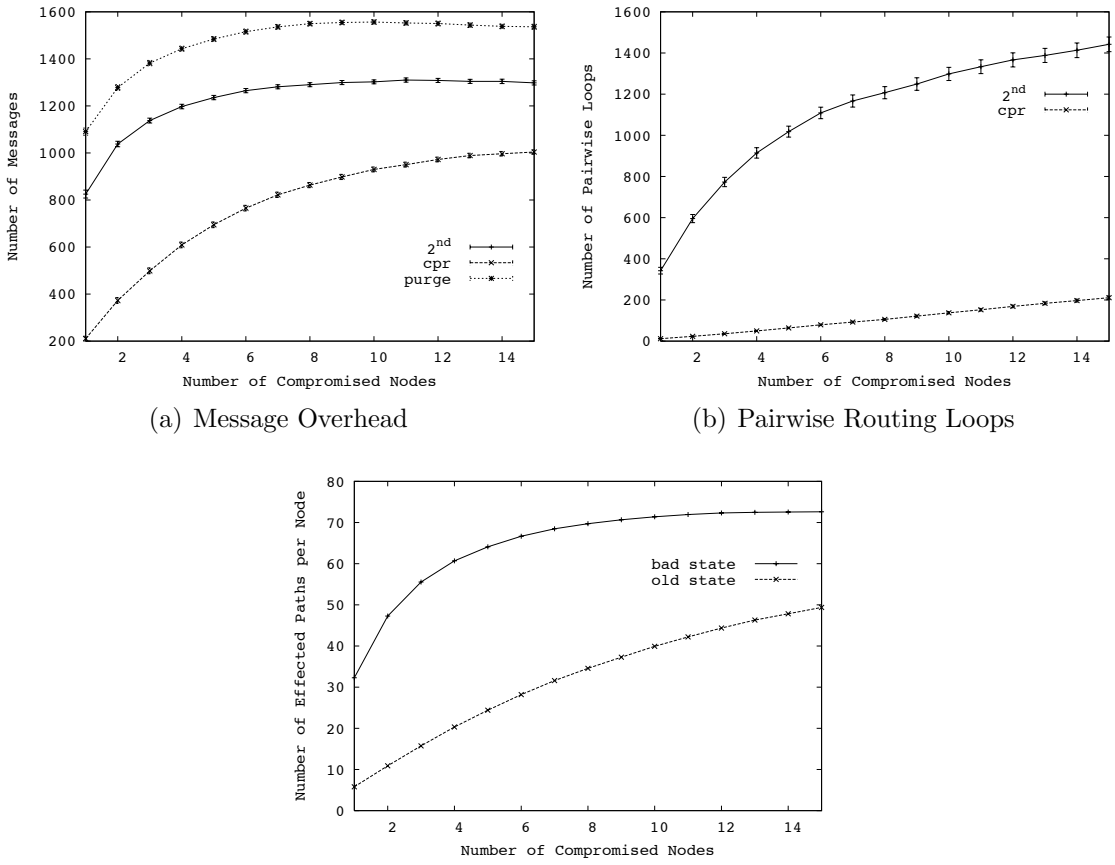
CPR message overhead increases with the number of compromised nodes because the amount of $\overrightarrow{old}$ state increases as the number of compromised nodes increase (Figure 2.7(c)). More $\overrightarrow{old}$ state results in more routing loops – as shown in Figure 2.7(b) – causing increased message overhead.

PURGE performs well because unlike CPR and 2ND-BEST, no routing loops occur during recovery. Surprisingly, PURGE's message overhead decreases when $m \geq 5$. Although more least cost paths need to be computed with larger m, the message overhead decreases because the residual graph, G', – resulting from the removal of all m compromised nodes – is smaller than G. As a result, there are m fewer destinations and m fewer nodes sending messages during the recovery process.

Finally, we simulated the same scenario of m compromised node using the Internet-like graphs from Simulation 3. The results were consistent with those for Erdös-Rényi graphs with random link weights.

2.5.1.5 Simulation 5: Adding Poisoned Reverse

Poisoned reverse is a common heuristic used to remove routing loops in distance vector routing. Poisoned reverse works as follows. When a node x routes through y to reach a destination w, x will advertise to y that its cost to reach w is ∞. In doing so, this prevents y from using x as its first-hop node to reach w, thereby

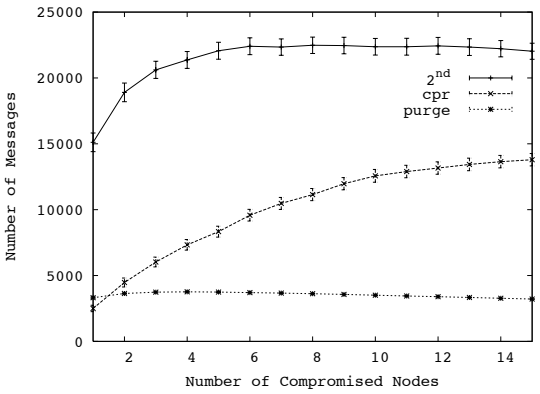


(a) Message Overhead

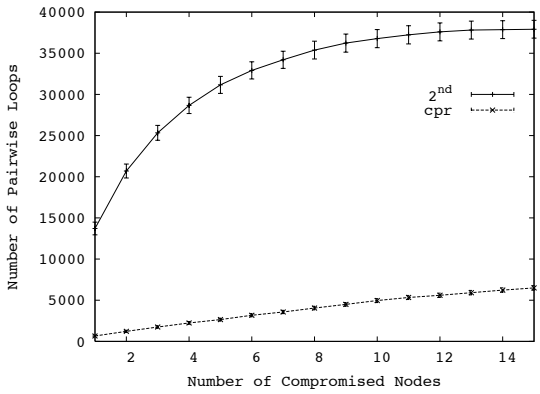
(b) Pairwise Routing Loops

(c) Number of Effected Least Cost Paths

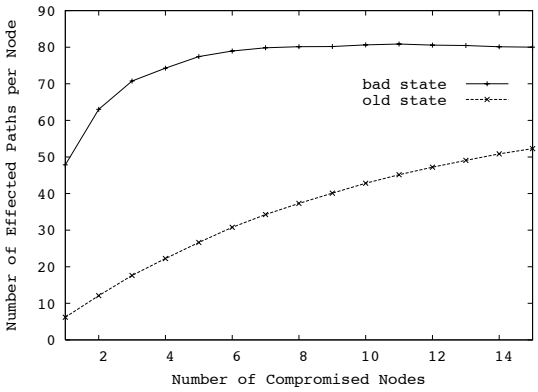
Figure 2.6. Simulation 4: simulations with multiple compromised nodes using Erdös-Rényi graphs with fixed link weights, $p = .05$, $n = 100$, and diameter=6.14. Results for different metrics as a function of the number of compromised nodes are shown.



(a) Message Overhead



(b) Pairwise Routing Loops



(c) Number of Effected Least Cost Paths

Figure 2.7. Simulation 4: multiple compromised nodes simulations over Erdös-Rényi graphs with link weights selected uniformly at random from $[1, 100]$, $p = .05$, $n = 100$, and diameter=6.14.

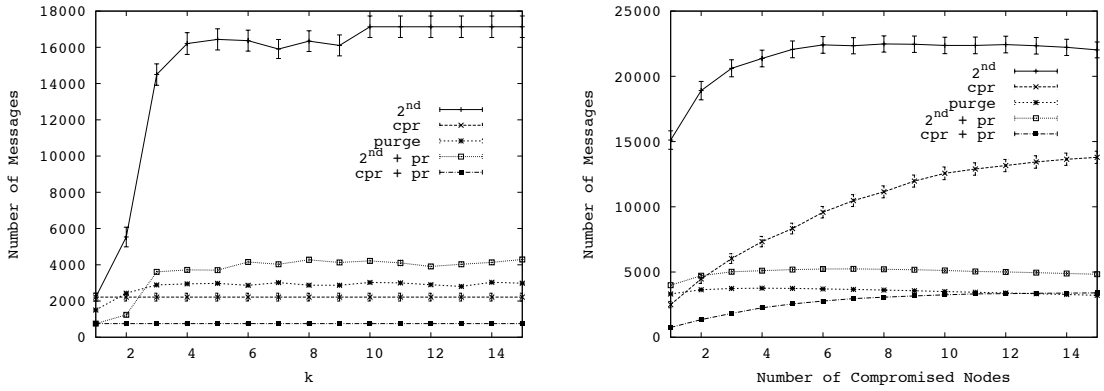
eliminating a possible routing loop between x and y. However, poisoned reverse only eliminates routing loops between two immediately adjacent nodes [48]. Here we study the benefits of applying poisoned reverse to 2ND-BEST and CPR.

We repeat Simulations 2, 3, and 4 using poisoned reverse with 2ND-BEST and CPR. We do not apply poisoned reverse to PURGE because no routing loops (resulting from the removal of $\overline{v}$) exist during PURGE's recovery. Additionally, we do not repeat Simulation 1 using poisoned reverse because we observed few routing loops in that simulation.

The results from repeating Simulation 2 using poisoned reverse are shown for one representative topology in Figure 2.8(a), where 2ND-BEST+PR and CPR+PR refer to each respective algorithm using poisoned reverse. CPR+PR has modest gains over standard CPR because few routing loops occur with CPR. On other hand, 2ND-BEST+PR sees a significant decrease in message overhead when compared to the standard 2ND-BEST algorithm because poisoned reverse removes the many pairwise routing loops that occur during 2ND-BEST recovery. However, 2ND-BEST+PR still performs worse than CPR+PR and PURGE. When compared to CPR+PR, the same reasons described in Simulation 2 account for 2ND-BEST+PR's poor performance.

Comparing PURGE and 2ND-BEST+PR yields interesting insights into the two different approaches for eliminating routing loops: PURGE prevents routing loops using diffusing computations and 2ND-BEST+PR uses poisoned reverse. Because PURGE has lower message complexity than 2ND-BEST+PR and poisoned reverse only eliminates pairwise routing loops, it suggests that PURGE removes routing loops larger than 2.

Repeating Simulation 3 using poisoned reverse yields the same trends as repeating Simulation 2 with poisoned reverse. Finally, we consider poisoned reverse in the case of multiple compromised nodes (e.g., we repeat Simulation 4). 2ND-BEST+PR and CPR+PR over Erdös-Rényi graphs with unit link weights perform only slightly better



(a) Single Compromised Node (b) Multiple Compromised nodes

Figure 2.8. Simulation 5 plots. Algorithms run over Erdös-Rényi graphs with random link weights, $n = 100$, $p = .05$, and average diameter=6.14. 2ND-BEST+PR refers to 2ND-BEST using poisoned reverse. Likewise, CPR+PR is CPR using poisoned reverse.

than the basic version of each algorithm, respectively. This is expected because few pairwise routing loops occur in this scenario.

Like the single compromised node scenario, in the case of multiple compromised nodes, 2ND-BEST+PR and CPR+PR over Erdös-Rényi graphs with random link weights provide significant improvements over the basic version of each algorithm. Particularly for 2ND-BEST, we observed many pairwise loops in Simulation 4 (Figure 2.7(b)). This accounts for the effectiveness of poisoned reverse in this simulation. Despite the significant improvements, 2ND-BEST+PR still performs worse than CPR+PR and PURGE. CPR+PR performs best among all the recovery algorithms because, as we have discussed, rolling back to a network-wide checkpoint is more efficient than using distance vector's iterative procedure. Furthermore, poisoned reverse helps CPR+PR reduce the count-to-infinity problem, improving CPR's effectiveness in the face of multiple compromised nodes.

2.5.2 Simulations using Graphs with Changing Link Weights

So far, we have evaluated our algorithms over different topologies with fixed link weights in scenarios with single and multiple compromised nodes. We found that CPR using poisoned reverse outperforms the other algorithms because CPR removes false routing state with a single diffusing computation, rather than using an iterative distance vector process as in 2ND-BEST and PURGE, and poisoned reverse removes all pairwise routing loops that occur during CPR recovery.

In the next three simulations we evaluate our algorithms over graphs with changing link weights. We introduce link weight changes between the time $\overline{v}$ is compromised and when $\overline{v}$ is discovered (e.g., during $[t', t_b]$). In particular, let there be λ link weight changes per timestep, where λ is deterministic. To create a link weight change event, we choose a link (except for all $(v, \bar{v})$ links) whose link will change equiprobably among all links. The new link weight is selected uniformly at random from $[1, n]$.

2.5.2.1 Simulation 6: Effects of Link Weight Changes

Except for λ, our simulation setup is identical to the one in Simulation 2. We let $\lambda = \{1, 4, 8\}$. In order to isolate the effects of link weights changes, we assume that CPR checkpoints at each timestep.

Figure 2.9 shows PURGE yields the lowest message overhead for $p = .05$, but only slightly lower than CPR. CPR's message overhead increases with larger k because there are more link weight change events to process. After CPR rolls back, it must process all link weight changes that occurred in $[t', t_b]$. In contrast, 2ND-BEST and PURGE process some of the link weight change events during the interval $[t', t_b]$ as part of normal distance vector execution. In our simulation setup, these messages are not counted because they do not occur in Step 4 (i.e., as part of the recovery process) of our simulation scenario described in Section 2.5.

Our analysis further indicates that 2ND-BEST performance suffers because of the count-to-infinity problem. The gap between 2ND-BEST and the other algorithms shrinks as λ increases because as λ increases, link weight changes have a larger effect on message overhead.

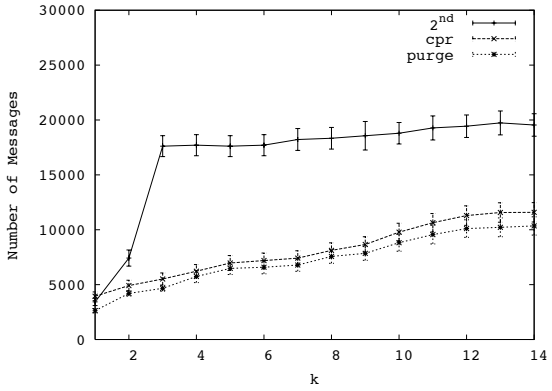
With larger p values, λ has a smaller effect on message complexity because more alternate paths are available. Thus when $p = 0.15$ and $\lambda = 1$, most of PURGE's recovery effort is towards removing $\overrightarrow{bad}$ state, rather than processing link weight changes. Because CPR removes $\overrightarrow{bad}$ using a single diffusing computation and there are few link weight changes, CPR has lower message overhead than PURGE in this case. As λ increases, CPR has higher message overhead than PURGE: there are more link weight changes to process and CPR must process all such link weight changes, while PURGE processes some link weight changes during the interval $[t', t_b]$ as part of normal distance vector execution.

2.5.2.2 Simulation 7: Applying Poisoned Reverse Heuristic

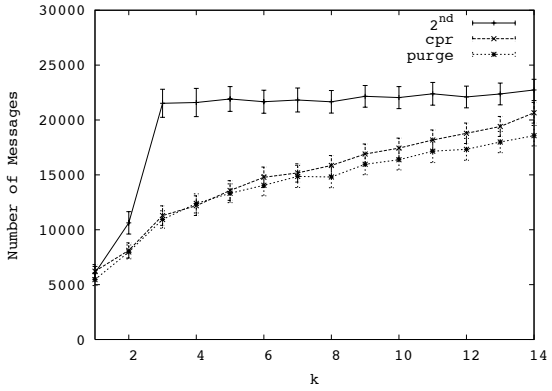
In this simulation, we apply poisoned reverse to each algorithm and repeat Simulation 6. Because PURGE's diffusing computations only eliminate routing loops corresponding to $\overrightarrow{bad}$ state, PURGE is vulnerable to routing loops stemming from link weight changes. Thus, contrary to Simulation 5, poisoned reverse improves PURGE performance. The results are shown in Figure 2.10. Results for different p values yield the same trends.

All three algorithms using poisoned reverse show remarkable performance gains. As confirmed by our profiling numbers, the improvements are significant because routing loops are more pervasive when link weights change. Accordingly, the poisoned reverse optimization yields greater benefits as λ increases.

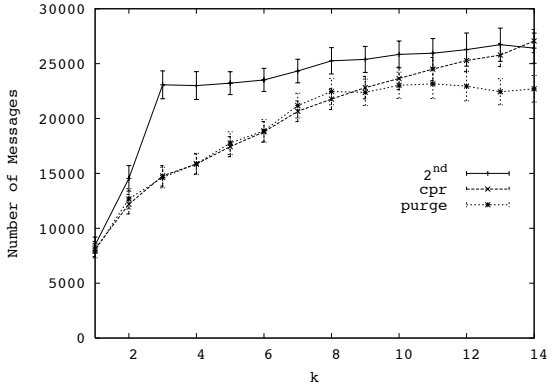
PURGE+PR removes all routing loops including loops with more than two nodes, while 2ND-BEST+PR does not. For this reason, PURGE+PR has lower message



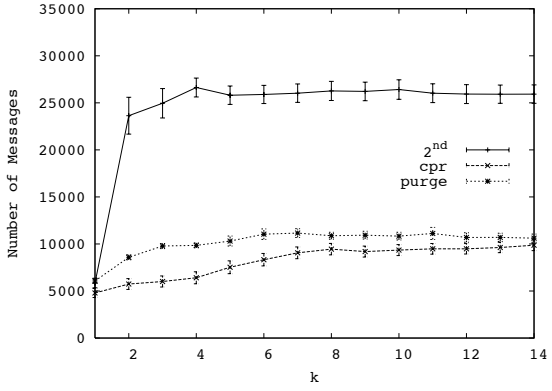
(a) $p = 0.05$, diameter=6.14, $\lambda = 1$



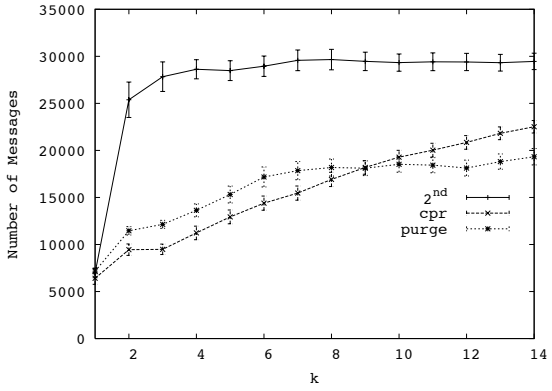
(b) $p = 0.05$, diameter=6.14, $\lambda = 4$



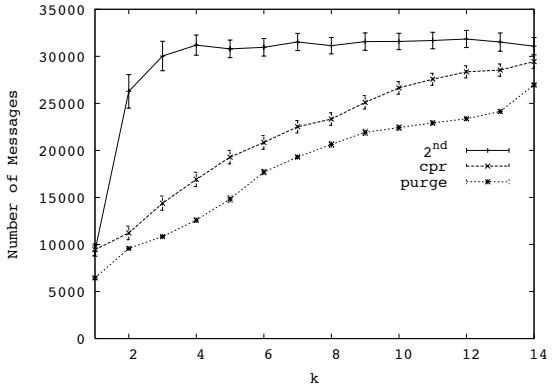
(c) $p = 0.05$, diameter=6.14, $\lambda = 8$



(d) $p = 0.15$, diameter=3.01, $\lambda = 1$



(e) $p = 0.15$, diameter=3.01, $\lambda = 4$



(f) $p = 0.15$, diameter=3.01, $\lambda = 8$

Figure 2.9. Simulation 6: Message overhead as a function of the number of hops false routing state has spread from the compromised node (k) for $p = \{0.05, 0.15\}$ Erdös-Rényi with link weights selected randomly with different λ values.

complexity. CPR+PR has the lowest message complexity. In this simulation, the benefits of rolling back to a global snapshot taken before $\overline{v}$ was compromised outweigh the message overhead required to update stale state pertaining to link weight changes that occurred during $[t', t_b]$. As λ increases, the performance gap decreases because CPR+PR must process all link weight changes that occurred in $[t', t_b]$ while 2ND-BEST+PR and PURGE+PR process some link weight change events during $[t', t_b]$ as part of normal distance vector execution.

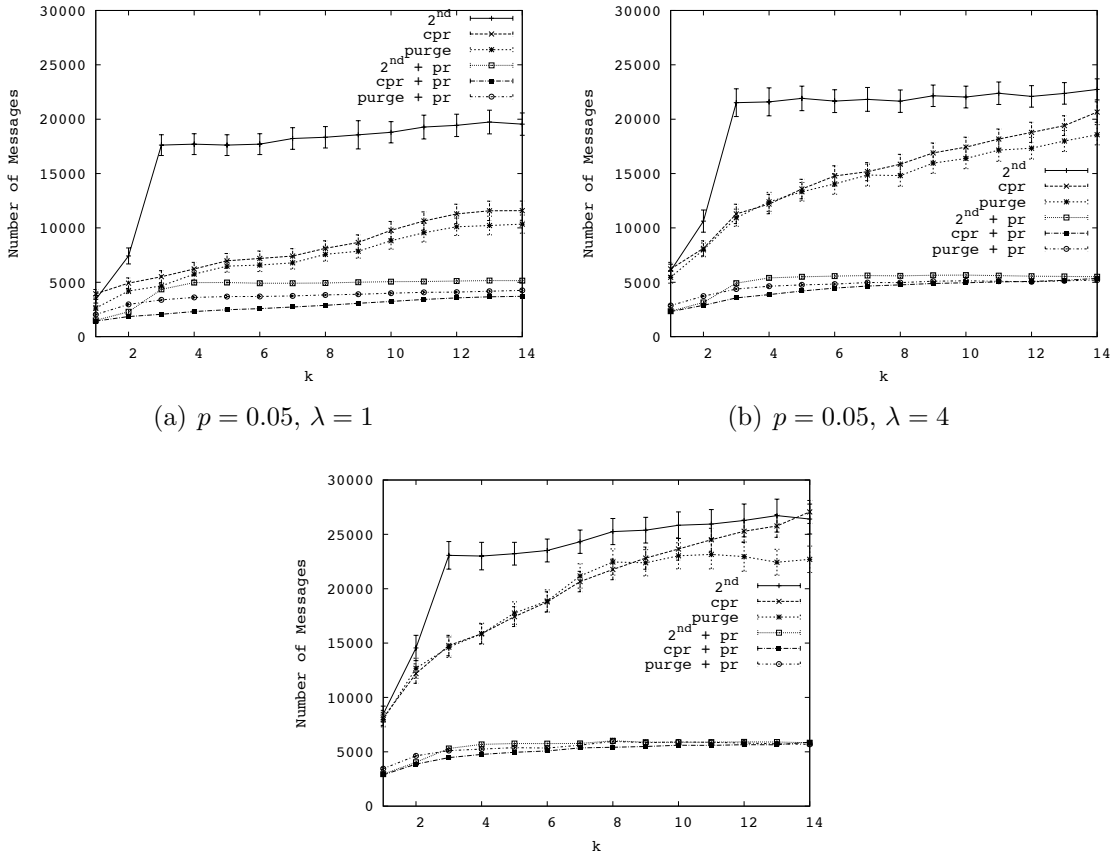
However, CPR+PR only achieves such strong results by making two optimistic assumptions: we assume perfectly synchronized clocks and checkpointing occurs at each timestep. In the next simulation we relax the checkpointing assumption.

2.5.2.3 Simulation 8: Effects of Checkpoint Frequency

In this simulation we study the trade-off between message overhead and storage overhead for CPR. To this end, we vary the frequency at which CPR checkpoints and fix the interval $[t', t_b]$. Otherwise, our simulation setup is the same as Simulation 6.

Figure 2.11 shows the results for an Erdös-Rényi graph with link weights selected uniformly at random between $[1, n]$, $n = 100$, $p = .05$, $\lambda = \{1, 4, 8\}$ and $k = 2$. We plot message overhead against the number of timesteps CPR must rollback, z. CPR's message overhead increases with larger z because as z increases there are more link weight change events to process. 2ND-BEST and PURGE have constant message overhead because they operate independent of z.

We conclude that as the frequency of CPR snapshots decreases, CPR incurs higher message overhead. Therefore, when choosing the frequency of checkpoints, the trade-off between storage and message overhead must be carefully considered.

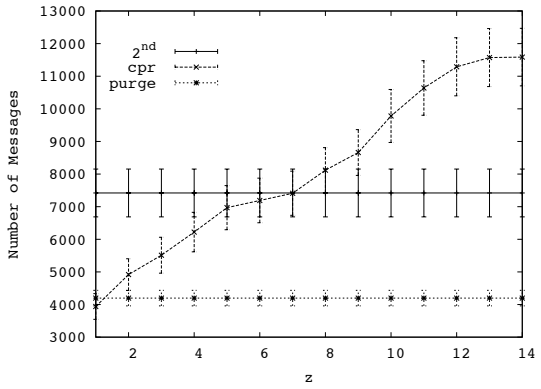


(a) $p = 0.05$, $\lambda = 1$

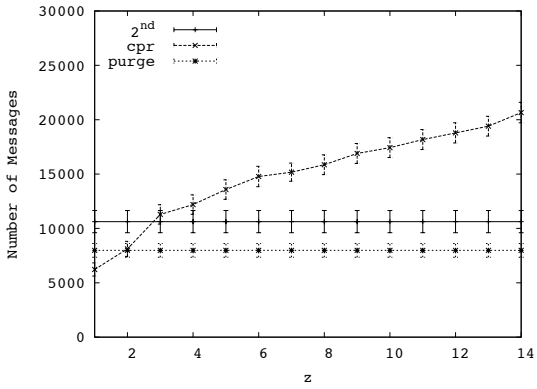
(b) $p = 0.05$, $\lambda = 4$

(c) $p = 0.05$, $\lambda = 8$

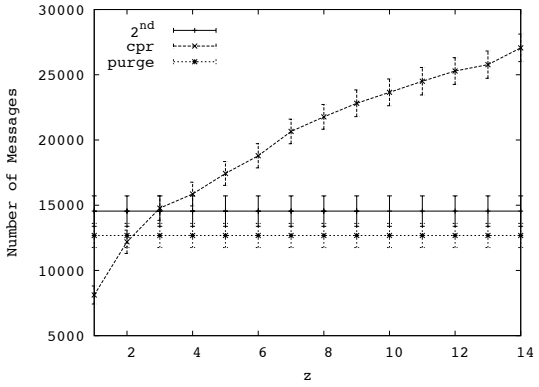
Figure 2.10. Plots for Simulation 7 using Erdös-Rényi graphs with link weights selected uniformly at random, $p = 0.05$, average diameter is 6.14, and $\lambda = \{1, 4, 8\}$. Message overhead is plotted as a function of k, the number of hops false routing state has spread from the compromised node. The curves for 2ND-BEST+PR, PURGE+PR, and CPR+PR refer to each algorithm using poisoned reverse, respectively.



(a) $p = 0.05$, $k = 2, \lambda = 1$



(b) $p = 0.05$, $k = 2, \lambda = 4$



(c) $p = 0.05$, $k = 2, \lambda = 8$

Figure 2.11. Simulation 8: message overhead for $p = 0.05$ Erdös-Rényi with link weights selected uniformly random with different λ values. z refers to the number of timesteps CPR must rollback. Note the y-axes have different scales.

2.5.3 Summary of Simulation Results

Our results show CPR using poisoned reverse yields the lowest message and time overhead in all scenarios. CPR benefits from removing false state with a single diffusing computation. Also, applying poisoned reverse significantly reduces CPR message complexity by eliminating pairwise routing loops resulting from link weight changes. However, CPR has storage overhead, requires loosely synchronized clocks, and requires the time $\overline{v}$ was compromised.

2ND-BEST's performance is determined by the count-to-infinity problem. In the case of Erdös-Rényi graphs with fixed unit link weights, the count-to-infinity problem was minimal, helping 2ND-BEST perform better than PURGE. For all other topologies, poisoned reverse significantly improves 2ND-BEST performance because routing loops are pervasive. Still, 2ND-BEST using poisoned reverse is not as efficient as CPR using poisoned reverse and PURGE.

In cases where link weights change, we found that PURGE using poisoned reverse is only slightly worse than CPR+PR. Unlike CPR, PURGE makes use of computations that follow the injection of false state, that do not depend on false routing state. Because PURGE does not make the assumptions that CPR requires, PURGE using poisoned reverse is a suitable alternative for topologies with link weight changes.

Finally, we found that an additional challenge with CPR is setting the parameter which determines checkpoint frequency. Frequent checkpointing yields lower message and time overhead at the cost of more storage overhead. Ultimately, application-specific factors must be considered when setting this parameter.

2.6 Related Work

To the best our knowledge no existing approach exists to address recovery from false routing state in distance vector routing. However, our problem is similar to that of recovering from malicious but committed database transactions. Liu et al. [6] and

Ammann et al [54] develop algorithms to restore a database to a valid state after a malicious transaction has been identified. PURGE's algorithm to globally invalidate false state can be interpreted as a distributed implementation of the dependency graph approach by Liu et al. [54]. Additionally, if we treat link weight change events that occur after the compromised node has been discovered as database transactions, we face a similar design decision as in [6]: do we wait until recovery is complete before applying link weight changes or do we allow the link weight changes to execute concurrently?

Database crash recovery [62] and message passing systems [25] both use snapshots to restore the system in the event of a failure. In both problem domains, the snapshot algorithms are careful to ensure snapshots are globally consistent. In our setting, consistent global snapshots are not required for CPR, since distance vector routing only requires that all initial distance estimates be non-negative.

Garcia-Lunes-Aceves's DUAL algorithm [32] uses diffusing computations to coordinate least cost updates in order to prevent routing loops. In our case, CPR and the prepossessing procedure (Section 2.3.1) use diffusing computations for purposes other than updating least costs (e.g., rollback to a checkpoint in the case of CPR and remove $\overline{v}$ as a destination during preprocessing). Like DUAL, the purpose of PURGE's diffusing computations is to prevent routing loops. However, PURGE's diffusing computations do not verify that new least costs preserve loop free routing (as with DUAL) but instead globally invalidate false routing state.

Jefferson [40] proposes a solution to synchronize distributed systems called Time Warp. Time Warp is a form of optimistic concurrency control and, as such, occasionally requires rolling back to a checkpoint. Time Warp does so by "unsending" each message sent after the time the checkpoint was taken. With our CPR algorithm, a node does not need to explicitly "unsend" messages after rolling back. Instead, each

node sends its $\overrightarrow{min}$ taken at the time of the snapshot, which implicitly undoes the effects of any messages sent after the snapshot timestamp.

2.7 Conclusions

In this chapter, we developed methods for recovery in scenarios where a malicious node injects false state into a distributed system. We studied an instance of this problem in distance vector routing. We presented and evaluated three new algorithms for recovery in such scenarios. Among our three algorithms, our results showed that CPR – a checkpoint-rollback based algorithm – yields the lowest message and time overhead over topologies with fixed link weights. However, CPR had storage overhead and required either loosely synchronized clocks or synchronization through logical clocks. In the case of topologies where links weights can change, PURGE performed best by avoiding the problems that plagued CPR and 2ND-BEST. Unlike CPR, PURGE has no stale state to update because PURGE does not rollback in time. The count-to-infinity problem resulted in high message overhead for 2ND-BEST, while PURGE eliminated the count-to-infinity problem by globally purging false state before finding new least cost paths.

CHAPTER 3

PMU SENSOR PLACEMENT FOR MEASUREMENT ERROR DETECTION IN THE SMART GRID

3.1 Introduction

This chapter considers placing electric power grid sensors, called phasor measurement units (PMUs), to enable measurement error detection. Significant investments have been made to deploy PMUs on electric power grids worldwide. PMUs provide *synchronized* voltage and current measurements at a sampling rate orders of magnitude higher than the status quo: 10 to 60 samples per second rather than one sample every 1 to 4 seconds. This allows system operators to directly measure the state of the electric power grid in real-time, rather than relying on imprecise state estimation. Consequently, PMUs have the potential to enable an entirely new set of applications for the power grid: protection and control during abnormal conditions, real-time distributed control, postmortem analysis of system faults, advanced state estimators for system monitoring, and the reliable integration of renewable energy resources [13].

An electric power system consists of a set of buses – electric substations, power generation centers, or aggregation points of electrical loads – and transmission lines connecting those buses. The state of a power system is defined by the voltage phasor – the magnitude and phase angle of electrical sine waves – of all system buses and the current phasor of all transmission lines. PMUs placed on buses provide real-time measurements of these system variables. However, because PMUs are expensive, they cannot be deployed on all system buses [9][22]. Fortunately, the voltage phasor at a system bus can, at times, be determined (termed *observed* in this thesis) even

when a PMU is not placed at that bus, by applying Ohm's and Kirchhoff's laws on the measurements taken by a PMU placed at some nearby system bus [9][14]. Specifically, with correct placement of enough PMUs at a subset of system buses, the entire system state can be determined.

In this chapter, we study two sets of PMU placement problems. The first problem set consists of FullObserve and MaxObserve, and considers maximizing the observability of the network via PMU placement. FullObserve considers the minimum number of PMUs needed to observe all system buses, while MaxObserve considers the maximum number of buses that can be observed with a given number of PMUs. A bus is said to be *observed* if there is a PMU placed at it or if its voltage phasor can be calculated using Ohm's or Kirchhoff's Law. Although FullObserve is well studied [9, 14, 36, 60, 77], existing work considers only networks consisting solely of zero-injection buses, an unrealistic assumption in practice, while we generalize the problem formulation to include mixtures of zero and non-zero-injection buses. Additionally, our approach for analyzing FullObserve provides the foundation with which to present the other three new (but related) PMU placement problems.

The second set of placement problems considers PMU placements that support PMU error detection. PMU measurement errors have been recorded in actual systems [75]. One method of detecting these errors is to deploy PMUs "near" each other, thus enabling them to *cross-validate* each-other's measurements. FullObserve-XV aims to minimize the number of PMUs needed to observe all buses while insuring PMU cross-validation, and MaxObserve-XV computes the maximum number of observed buses for a given number of PMUs, while insuring PMU cross-validation.

We make the following contributions in this chapter:

- We formulate two PMU placement problems, which (broadly) aim at maximizing observed buses while minimizing the number of PMUs used. Our formula-

tion extends previously studied systems by considering both zero and non-zero-injection buses.

- We formally define graph-theoretic rules for PMU cross-validation. Using these rules, we formulate two additional PMU placement problems that seek to maximize the number of observed buses while minimizing the number of PMUs used under the condition that the PMUs are cross-validated.

- We prove that all four PMU placement problems are NP-Complete. This represents our most important contribution.

- Given the proven complexity of these problems, we evaluate heuristic approaches for solving these problems. For each problem, we describe a greedy algorithm, and prove that each greedy algorithm has polynomial running time.

- Using simulations, we evaluate the performance of our greedy approximation algorithms over synthetic and actual IEEE bus systems. We find that the greedy algorithms yield a PMU placement that is, on average, within 97% optimal. Additionally, we find that the cross-validation constraints have limited effects on observability: on average our greedy algorithm that places PMUs according to the cross-validation rules observes only 5.7% fewer nodes than the same algorithm that does not consider cross-validation.

The rest of this chapter is organized as follows. In Section 4.2 we introduce our modeling assumptions, notation, and observability and cross-validation rules. In Section 3.3 we formulate and prove the complexity of our four PMU placement problems. Section 3.4 presents the approximation algorithms for each problem and Section 3.5 considers our simulation-based evaluation. We conclude with a review of related work (Section 3.6) and concluding remarks (Section 3.7).

3.2 Preliminaries

In this section we introduce notation and underlying assumptions (Section 3.2.1), and define our observability (Section 3.2.2) and cross-validation (Section 3.2.3) rules.

3.2.1 Assumptions, Notation, and Terminology

We model a power grid as an undirected graph $G = (V, E)$. Each $v \in V$ represents a bus. $V = V_Z \cup V_I$, where V_Z is the set of all zero-injection buses and V_I is the set of all non-zero-injection buses. A bus is zero-injection if it has no load nor generator [80]. All other buses are non-zero-injection, which we refer to as injection buses. Each $(u, v) \in E$ is a transmission line connecting buses u and v.

Consistent with the conventions in [9, 14, 18, 60, 77, 78], we make the following assumptions about PMU placements and buses. First, a PMU can only be placed on a bus. Second, a PMU on a bus measures the voltage phasor at the bus and the current phasor of all transmission lines connected to it.

Using the same notation as Brueni and Heath [14], we define two Γ functions. For $v \in V$ let $\Gamma(v)$ be the set of v's neighbors in G, and $\Gamma[v] = \Gamma(v) \cup \{v\}$. A PMU placement $\Phi_G \subseteq V$ is a set of nodes at which PMUs are placed, and $\Phi_G^R \subseteq V$ is the set of observed nodes for graph G with placement Φ_G (see definition of observability below). $k^* = \min\{|\Phi_G| : \Phi_G^R = V\}$ denotes the minimum number of PMUs needed to observe the entire network. Where the graph G is clear from the context, we drop the G subscript.

For convenience, we refer to any node with a PMU as a *PMU node*. Additionally, for a given PMU placement we shall say that a set $W \subseteq V$ is observed if all nodes in the set are observed, and if $W = V$ we refer to the graph as *fully observed*.

3.2.2 Observability Rules

We use the simplified observability rules stated by Brueni and Heath [14]. For completeness, we restate the rules here:

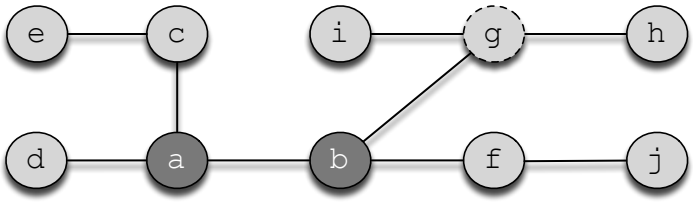


Figure 3.1. Example power system graph. PMU nodes (a, b) are indicated with darker shading. Injection nodes have solid borders while zero-injection nodes (g) have dashed borders.

1. **Observability Rule 1 (O1)**. *If node v is a PMU node, then $v \cup \Gamma(v)$ is observed.*

2. **Observability Rule 2 (O2)**. *If a zero-injection node, v, is observed and $\Gamma(v) \backslash \{u\}$ is observed for some $u \in \Gamma(v)$, then $v \cup \Gamma(v)$ is observed.*

Consider the example in Figure 3.1, where the shaded nodes are PMU nodes and g is the only zero-injection node. Nodes $a - d$ are observed by applying O1 at the PMU at a, and nodes a, b, f and g are observed by applying O1 at b. e cannot be observed via c because c does not have a PMU (O1 does not apply) and is an injection node (O2 does not apply). Similarly, j is not observed via f. Finally, although $g \in V_Z$, O2 cannot be applied at g because g has two unobserved neighbors i, h, so they remain unobserved.

Since O2 only applies with zero-injection nodes, the number of zero-injection nodes can greatly affect system observability. For example, consider the case where c and f are *zero-injection* nodes. $a - d$, g and f are still observed as before, as O1 makes not conditions on the node type. Additionally, since now $c, f \in V_Z$ and each has a single unobserved neighbor, we can apply O2 at each of them to observe e, j, respectively. We evaluate the effect of increasing the number of zero-injection nodes on observability in our simulations (Section 3.5).

3.2.3 Cross-Validation Rules

Cross-validation formalizes the intuitive notion of placing PMUs "near" each other to allow for measurement error detection. From Vanfretti et al. [75], PMU measurements can be cross-validated when: (1) a voltage phasor of a non-PMU bus can be computed by PMU data from two different buses or (2) the current phasor of a transmission line can be computed from PMU data from two different buses. [1]

For convenience, we say a PMU is cross-validated even though it is actually the PMU data at a node that is cross-validated. A PMU is *cross-validated* if one of the rules below is satisfied [75]:

1. **Cross-Validation Rule 1 (XV1)**. *If two PMU nodes are adjacent, then the PMUs cross-validate each other.*

2. **Cross-Validation Rule 2 (XV2)**. *If two PMU nodes have a common neighbor, then the PMUs cross-validate each other.*

In short, the cross-validation rules require that *the PMU is within two hops of another PMU*. For example, in Figure 3.1, the PMUs at a and b cross-validate each other by XV1.

XV1 derives from the fact that both PMUs are measuring the current phasor of the transmission line connecting the two PMU nodes. XV2 is more subtle. Using the notation specified in XV2, when computing the voltage phasor of an element in $\Gamma(u) \cap \Gamma(v)$ the voltage equations include variables to account for measurement error (e.g., angle bias) [74]. When the PMUs are two hops from each other (i.e., have a common neighbor), there are more equations than unknowns, allowing for measurement error detection. Otherwise, the number of unknown variables exceeds the number of equations, which eliminates the possibility of detecting measurement errors [74].

[1] Vanfretti et al. [75] use the term "redundancy" instead of cross-validation.

3.3 Four NP-Complete PMU Placement Problems

In this section we define four PMU placement problems (FullObserve, MaxObserve, FullObserve-XV, and MaxObserve-XV) and prove their NP-Completeness. FullObserve-XV and MaxObserve-XV both consider measurement error detection, while FullObserve and MaxObserve do not. We begin with a general overview of NP-Completeness, as well as a high-level description of the proof strategy used in this chapter (Section 3.3.1). In the remainder of Section 3.3 we present and prove the NP-Completeness of four PMU placement problems, in the following order: FullObserve (Section 3.3.2), MaxObserve (Section 3.3.3), FullObserve-XV (Section 3.3.4), and MaxObserve-XV (Section 3.3.5).

In all four problems we are only concerned with computing the voltage phasors of each bus (i.e., observing the buses). Using the values of the voltage phasors, Ohm's Law can be easily applied to compute the current phasors of each transmission line. Also, we consider networks with both injection and zero-injection buses. For similar proofs for purely zero-injection systems, see Appendix B.

3.3.1 NP-Completeness Overview and Proof Strategy

Before proving that our PMU placement problems are NP-Complete (abbreviated NPC), we provide some background on NP-Completeness. NPC problems are the hardest problems in complexity class $\mathcal{NP}$. It is generally assumed that solving NPC problems is hard, meaning that any algorithm that solves an NPC problem has exponential running time as function of the input size. It is important to clarify that despite being NPC, a *specific* problem instance might be efficiently solvable. This is either due to the special structure of the specific instance or because the input size is small, yielding a small exponent. For example, in Section 3.5 we are able to solve FullObserve for small IEEE bus topologies due to their small size. Thus, by establishing that our PMU placement problems are NPC, we claim that there

exist bus topologies for which these problems are difficult to solve (i.e., no known polynomial-time algorithm exists to solve those case).

To prove our problems are NPC, we follow the standard three-step reduction procedure. For a decision problem Π, we first show $\Pi \in \mathcal{NP}$. Second, we select a known NPC problem, denoted Π', and construct a polynomial-time transformation, f, that maps any instance of Π' to an instance of Π. Finally, we must ensure that for this f, $x \in \Pi' \Leftrightarrow f(x) \in \Pi$ [33].

Next, we outline the proof strategy we use throughout this section. In Sections 3.3.2 through Section 3.3.5 we use slight variations of the approach presented by Brueni and Heath in [14] to prove the problems we consider here are NPC. In general we found their scheme to be elegantly extensible for proving many properties of PMU placements.

In [14], the authors prove NP-Completeness by reduction from planar 3-SAT (P3SAT). A 3-SAT formula, ϕ, is a boolean formula in conjunctive normal form (CNF) such that each clause contains at most 3 literals. For any 3-SAT formula ϕ with the sets of variables $\{v_1, v_2, \ldots, v_r\}$ and clauses $\{c_1, c_2, \ldots, c_s\}$, $G(\phi)$ is the bipartite graph $G(\phi) = (V(\phi), E(\phi))$ defined as follows:

$$\begin{aligned} V(\phi) &= \{v_i \mid 1 \leq i \leq r\} \cup \{c_j \mid 1 \leq j \leq s\} \\ E(\phi) &= \{(v_i, c_j) \mid v_i \in c_j \ \ or \ \ \overline{v_i} \in c_j\}. \end{aligned}$$

Note that edges pass only between v_i and c_j nodes, and so the graph is bipartite. P3SAT is a 3-SAT formula such that $G(\phi)$ is planar [53]. For example, P3SAT formula

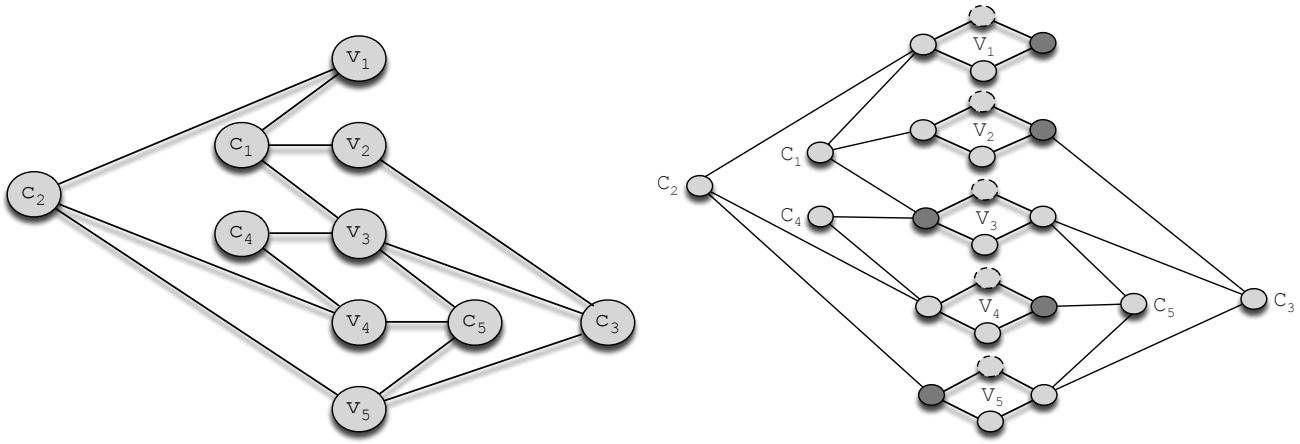
$$\begin{aligned} \varphi &= (\overline{v_1} \vee v_2 \vee v_3) \wedge (\overline{v_1} \vee \overline{v_4} \vee v_5) \wedge (\overline{v_2} \vee \overline{v_3} \vee \overline{v_5}) \\ &\quad \wedge (v_3 \vee \overline{v_4}) \wedge (\overline{v_3} \vee v_4 \vee \overline{v_5}) \end{aligned} \tag{3.1}$$

has graph $G(\varphi)$ shown in Figure 3.2(a). Discovering a satisfying assignment for P3SAT is an NPC problem, and so it can be used in a reduction to prove the complexity of the problems we address here. Note that in this work we will use φ to denote a specific P3SAT formula, while ϕ will be used to denote a generic P3SAT formula.

Following the approach in [14], for P3SAT formula, ϕ, we replace each variable node and each clause node in $G(\phi)$ with a specially constructed set of nodes, termed a *gadget*. In this work, all variable gadgets will have the same structure, and all clause gadgets have the same structure (that is different from the variable gadget structure), and we denote the resulting graph as $H(\phi)$. In $H(\phi)$, each *variable* gadget has a subset of nodes that semantically represent assigning "True" to that variable, and a subset of nodes that represent assigning it "False". When a PMU is placed at one of these nodes, this is interpreted as assigning a truth value to the P3SAT variable corresponding with that gadget. Thus, we use the PMU placement to determine a consistent truth value for each P3SAT variable. Also, clause gadgets are connected to variable gadgets at either "True" or "False" (but never both) nodes, in such a way that the clause is satisfied if and only if *at least one* of those nodes has a PMU.

Although the structure of our proofs is adapted from [14], the variable and clause gadgets we use to correspond to the P3SAT formula are novel, thus leading to a different set of proofs. Our work here demonstrates how the approach from [14] can be extended, using new variable and clause gadgets, to address a wide array of PMU placement problems.

While we assume $G(\phi)$ is planar, we make no such claim regarding $H(\phi)$, though in practice all graphs used in our proofs are indeed planar. The proof of NPC rests on the fact that solving the underlying ϕ formula is NPC. In what follows, for a given PMU placement problem Π, we prove Π is NPC by showing that a PMU placement



(a) $G(\varphi)$ formed from φ in Equation (3.1). (b) Graph formed from φ formula in Theorem 3.1 proof.

Figure 3.2. The figure in (a) shows $G(\varphi) = (V(\varphi), E(\varphi))$ using example formula, φ, from Equation (3.1). (b) shows the new graph formed by replacing each variable node in $G(\varphi)$ – as specified by the Theorem 3.1 proof – with the Figure 3.3(a) variable gadget.

in $H(\phi)$, Φ, can be interpreted semantically as describing a satisfying assignment for ϕ iff $\Phi \in \Pi$. Since P3SAT is NPC, this proves Π is NPC as well.

3.3.2 The FullObserve Problem

The FullObserve problem has been addressed in the literature (e.g., the PMUP problem in [14], and the PDS problem in [36]) but only for purely zero-injection bus systems. Here we consider networks with mixtures of injection and zero-injection buses, and modify the NPC proof of PMUP in [14] to handle this mixture.

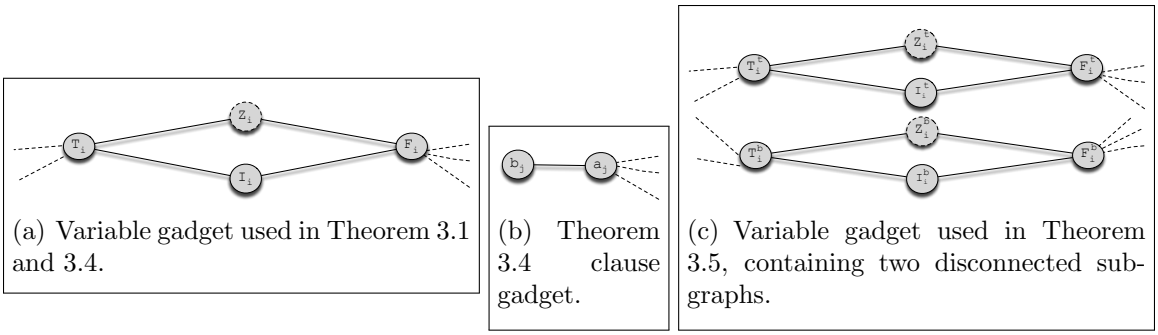
FullObserve Optimization Problem:

Input: Graph $G = (V, E)$ where $V = V_Z \cup V_I$ and $V_Z \neq \emptyset$. [2]

Output: A placement of PMUs, Φ_G, such that $\Phi_G^R = V$ and Φ_G is minimal.

FullObserve Decision Problem:

[2]We include the condition that $V_Z \neq \emptyset$ because otherwise FullObserve reduces to Vertex-Cover, making the NP-Completeness proof trivial.



(a) Variable gadget used in Theorem 3.1 and 3.4.

(b) Theorem 3.4 clause gadget.

(c) Variable gadget used in Theorem 3.5, containing two disconnected subgraphs.

Figure 3.3. Gadgets used in Theorem 3.1 - 3.7. Z_i in Figure 3.3(a), Z_i^t in Figure 3.3(c), and Z_i^b in Figure 3.3(c) are the only zero-injection nodes. The dashed edges in Figure 3.3(a) and Figure 3.3(c) are connections to clause gadgets. Likewise, the dashed edges in Figure (b) are connections to variable gadgets. In Figure 3.3(c), superscript, t, denotes nodes in the upper subgraph and superscript, b, indexes nodes in the lower subgraph.

Instance: Graph $G = (V, E)$ where $V = V_Z \cup V_I$, $V_Z \neq \emptyset$, k PMUs such that $k \geq 1$.

Question: Is there a Φ_G such that $|\Phi_G| \leq k$ and $\Phi_G^R = V$?

Theorem 3.1. FULLOBSERVE *is NP-Complete.*

Proof Idea: We introduce a problem-specific variable gadget. We show that in order to observe all nodes, PMUs must be placed on variable gadgets, specifically on nodes that semantically correspond to True and False values that satisfy the corresponding P3SAT formula.

For our first problem, we use a single node as a clause gadget denoted a_j, and the subgraph shown in Figure 3.3(a) as the variable gadget. Note that in the variable gadget, all the nodes are injection nodes except for Z_i. For this subgraph, we state the following simple lemma:

Lemma 3.2. *Consider the gadget shown in Figure 3.3(a), possibly with additional edges connected to T_i and/or F_i. Then (a) nodes I_i, Z_i are not observed if there is no PMU on the gadget, and (b) all the nodes in the gadget are observed with a single PMU iff the PMU is placed on either T_i or F_i.*

Proof. (a) If there is no PMU on the gadget, O1 cannot be applied at any of the nodes, and so we must resort to O2. We assume no edges connected to I_i, Z_i from outside the gadget, and since $T_i, F_i \in V_I$, we cannot apply O2 at them, which concludes our proof.

(b) In one direction, if we have a PMU placed at T_i, from O1 we can observe Z_i, I_i. Since Z_i is zero-injection and one neighbor, T_i has been observed, from O2 at Z_i we can observe F_i. The same holds for placing a PMU at F_i, due to symmetry.

In the other direction, by placing a PMU at I_i (Z_i) we observe T_i and F_i via O1. However, since $F_i, T_i \notin V_Z$, O2 cannot be applied at either of them, so Z_i (I_i) will not be observed. □

Proof of Theorem 3.1. We start by arguing that FULLOBSERVE $\in \mathcal{NP}$. First, non-deterministically select k nodes in which to place PMUs. Using the rules specified in Section 3.2.2, determining the number of observed nodes can be done in linear time.

To show FULLOBSERVE is NP-hard, we reduce from P3SAT. Let ϕ be an arbitrary P3SAT formula with variables $\{v_1, v_2, \ldots, v_r\}$ and the set of clauses $\{c_1, c_2, \ldots, c_s\}$, and $G(\phi)$ the corresponding planar graph. We use $G(\phi)$ to construct a new graph $H_0(\phi) = (V_0(\phi), E_0(\phi))$ by replacing each variable node in $G(\phi)$ with the variable gadget shown in Figure 3.3(a). The clause nodes consist of a single node (i.e., are the same as in $G(\phi)$). We denote the node corresponding to c_j as a_j. All clause nodes are injection nodes. In the remainder of this proof we let $H := H_0(\phi)$. In total, V_Z contains all Z_i nodes for $1 \leq i \leq r$, and all other nodes are in V_I. The edges connecting clause nodes with variable gadgets express which variables are in each clause: for each clause node a_j, $(T_i, a_j) \in E_0(\phi) \Leftrightarrow v_i \in c_j$, and $(F_i, a_j) \in E_0(\phi) \Leftrightarrow \overline{v_i} \in c_j$. As a result, the following observation holds:

Observation 3.3. *For a given truth assignment and a corresponding PMU placement, a clause c_j is satisfied iff a_j is attached to a node in a variable gadget with a PMU.*

The resulting graph for the example given in Figure 3.2(a) is shown in Figure **??**. Nodes with a dashed border are zero-injection nodes. [3] The corresponding formula for this graph, φ, is satisfied by truth assignment A_φ: $\overline{v_1}, \overline{v_2}, v_3, \overline{v_4}$, and $\overline{v_5}$ are True. This corresponds to the dark shaded nodes in Figure 3.2(b). While this construction generates a graph with very specific structure, in Section 3.3.6, we detail how to extend our proof to consider graphs with a wider range of structures.

With this construct in place, we move on to our proof. We show that ϕ is satisfiable if and only if $k = r = |\Phi_H|$ PMUs can be placed on H such that $\Phi_H^R = V$.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. Then, consider the placement Φ_H such that for each variable gadget V_i, $T_i \in \Phi_H \Leftrightarrow v_i = True$ in A_ϕ, and $F_i \in \Phi_H \Leftrightarrow v_i = False$. From Lemma 3.2(b) we know that all nodes in variable gadgets are observed by such a placement. From Observation 3.3, all clause nodes are observed because our PMU assignment is based on a satisfying assignment. Thus, we have shown that $\Phi_H^R = V$.

($\Leftarrow$) Suppose there is a placement of r PMUs, Φ_H, such that $\Phi_H^R = V$. From Lemma 3.2(a) we know that for each V_i with no PMU, at least two nodes are not observed, so each V_i must have a PMU placed in it. Since we have only r PMUs, that means one PMU per gadget. From Lemma 3.2(b) we know this PMU must be placed on T_i or F_i, since otherwise the gadget will not be fully observed. Note that these nodes are all in V_I.

Since we assume the graph is fully observed, all a_j are observed by Φ_H. Because we just concluded that PMUs are placed only on injection nodes in the variable gadgets, each clause node a_j can only be observed via application of O1 at T_i/F_i nodes to which it is attached – specifically, a_j is attached to a node with a PMU. From Observation

[3] Throughout this chapter, nodes with dashed borders denote zero-injection nodes.

3.3 this means that all clauses are satisfied by the semantic interpretation of our PMU placement, which concludes our proof. □

3.3.3 The MaxObserve Problem

MaxObserve is a variation of FullObserve: rather than consider the minimum number of PMUs required for full system observability, MaxObserve finds the maximum number of nodes that can be observed using a fixed number of PMUs.

MaxObserve Optimization Problem:

Input: Graph $G = (V, E)$ where $V = V_Z \cup V_I$, k PMUs such that $1 \leq k < k^*$.

Output: A placement of k PMUs, Φ_G, such that $|\Phi_G^R|$ is maximum.

MaxObserve Decision Problem:

Instance: Graph $G = (V, E)$ where $V = V_Z \cup V_I$, k PMUs such that $1 \leq k < k^*$.

Question: For a given $m < |V|$, is there a Φ_G such that $|\Phi_G| \leq k$ and $m \leq |\Phi_G^R| < |V|$?

Theorem 3.4. MaxObserve *is NP-Complete.*

Proof Idea: First, we construct problem-specific gadgets for variables and clauses. We then demonstrate that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. Next we show that as a result of this, the problem of observing m nodes in this graph reduces to Theorem 3.1.

Proof. MaxObserve $\in \mathcal{NP}$ using the same argument in the proof for Theorem 3.1.

Next, we reduce from P3SAT as in the proof for Theorem 3.1, where ϕ is an arbitrary P3SAT formula. We create a new graph $H_1(\phi) = (V_1(\phi), E_1(\phi))$ which is identical to $H_0(\phi)$ from the previous proof, except that each clause node in $H_0(\phi)$ is replaced with the clause gadget shown in Figure 3.3(b), comprising of two injection nodes. As before, the edges connecting clause nodes with variable gadgets express which variables are in each clause: for each clause node a_j, $(T_i, a_j) \in E_1(\phi) \Leftrightarrow v_i \in c_j$, and $(F_i, a_j) \in E_1(\phi) \Leftrightarrow \overline{v_i} \in c_j$. Note that Observation 3.3 holds here as well.

We are now ready to show MaxObserve is NP-hard. For convenience, we let $H := H_1(\phi)$. Recall ϕ has r variables and s clauses. Here we consider the instance of MaxObserve where $k = r$ and $m = 4r + s$, and show that ϕ is satisfiable if and only if $r = |\Phi_H|$ PMUs can be placed on H such that $m \leq |\Phi_H^R| < |V|$. In Section 3.3.6 we discuss how to extend this proof for any larger value of m and different $\frac{|V_Z|}{|V_I|}$ ratios.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. Then, consider the placement Φ_H such that for each variable gadget V_i, $T_i \in \Phi_H \Leftrightarrow v_i = True$ in A_ϕ, and $F_i \in \Phi_H \Leftrightarrow v_i = False$. In the proof for Theorem 3.1 we demonstrated such a placement will observe all nodes in $H_0(\phi) \subset H_1(\phi)$, and using the same argument it can easily be checked that these nodes are still observed in $H_1(\phi)$. Each b_j node remains unobserved because each $a_j \in V_I$ and consequently O2 cannot be applied at a_j. Since $|H_0(\phi)| = 4r + s = m$, we have observed the required nodes.

($\Leftarrow$) We begin by proving that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. By construction, each PMU is either on a clause gadget or a variable gadget, but not both. Let $0 \leq t \leq r$ be the number of PMUs on clause gadgets, we wish to show that for the given placement $t = 0$. First, note that *at least* $\max(s - t, 0)$ clause gadgets are without PMUs, and that for each such clause (by construction) at least one node (b_i) is not observed. Next, from Lemma 3.2(a) we know that for each variable gadget without a PMU, at least two nodes are not observed.

Denote the *unobserved* nodes for a given PMU placement as Φ_H^-. Thus, we get $|\Phi_H^-| \geq 2t + \max((s - t), 0)$. However, since m nodes are observed and $|V| - m \leq s$, we get $|\Phi_H^-| \leq s$, so we know $s \geq 2t + \max((s - t), 0)$. We consider two cases:

- $s \geq t$: then we get $s \geq t + s \Rightarrow t = 0$.
- $s < t$: then we get $s \geq 2t$, and since we assume here $0 \leq s < t$ this leads to a contradiction and so this case cannot occur.

Thus, the r PMUs must be on nodes in variable gadgets. Note that the variable gadgets in $H_1(\phi)$ have the same structure as in $H_0(\phi)$. We return to this point shortly.

Earlier we noted that for each clause gadget without a PMU, the corresponding b_j node is unobserved, which comes to s nodes. To observe $m = 4r + s$ nodes, we will need to observe all the remaining nodes. Thus, we have reduced the problem to that of observing all of $H_0(\phi) \subset H_1(\phi)$. Our proof for Theorem 3.1 demonstrated this can only be done by placing PMUs at nodes corresponding to a satisfying assignment of ϕ, and so our proof is complete. □

3.3.4 The FullObserve-XV Problem

The FullObserve-XV optimization and decision problems are defined as follows:

FullObserve-XV Optimization Problem:

Input: Graph $G = (V, E)$ where $V = V_Z \cup V_I$.

Output: A placement of PMUs, Φ_G, such that $\Phi_G^R = V$, and Φ_G is minimal under the condition that each $v \in \Phi_G$ is cross-validated according to the rules specified in Section 3.2.3.

FullObserve-XV Decision Problem:

Instance: Graph $G = (V, E)$ where $V = V_Z \cup V_I$, k PMUs such that $k \geq 1$.

Question: Is there a Φ_G such that $|\Phi_G| \leq k$ and $\Phi_G^R = V$ under the condition that each $v \in \Phi_G$ is cross-validated?

Theorem 3.5. FullObserve-XV *is NP-Complete.*

Proof Idea: We show FullObserve-XV is NP-hard by reducing from P3SAT. We create a single-node gadget for clauses (as for FullObserve) and the gadget shown in Figure 3.3(c) for each variable. Each variable gadget here comprises of two disconnected components, and there are two T_i and two F_i nodes, one in each component. First, we show that each variable gadget must have 2 PMUs for the

entire graph to be observed, one PMU for each subgraph. Then, we show that cross-validation constraints force PMUs to be placed on both T nodes or both F nodes. Finally, we show how to use the PMU placement to derive a satisfying P3SAT truth assignment.

Lemma 3.6. *Consider the gadget shown in Figure 3.3(c), possibly with additional nodes attached to T_i and/or F_i nodes. (a) nodes I_i^t, Z_i^t are not observed if there is no PMU on V_i^t, and (b) all the nodes in V_i^t are observed with a single PMU iff the PMU is placed on either T_i^t or F_i^t. Due to symmetry, the same holds when considering V_i^b.*

Proof. The proof is straightforward from the proof of Lemma 3.2, since both V_i^t and V_i^b are identical to the gadget from Figure 3.3(a), which Lemma 3.2 refers to. □

Proof of Theorem 3.5. First, we argue that FullObserve-XV $\in \mathcal{NP}$. Given a FullObserve-XV solution, we use the polynomial time algorithm described in our proof for Theorem 3.1 to determine if all nodes are observed. Then, for each PMU node we run a breadth-first search, stopping at depth 2, to check that the cross-validation rules are satisfied.

To show FullObserve-XV is NP-hard, we reduce from P3SAT. Our reduction is similar to the one used in Theorem 3.1. We start with the same P3SAT formula ϕ with variables $\{v_1, v_2, \ldots, v_r\}$ and the set of clauses $\{c_1, c_2, \ldots, c_s\}$.

For this problem, we construct $H_2(\phi)$ in the following manner. We use the single-node clause gadgets as in $H_0(\phi)$, and as before, the edges connecting clause nodes with variable gadgets shown in Figure 3.3(c) express which variables are in each clause: for each clause node a_j, $(T_i^t, a_j), (T_i^b, a_j) \in E_1(\phi) \Leftrightarrow v_i \in c_j$, and $(F_i^t, a_j), (F_i^b, a_j) \in E_1(\phi) \Leftrightarrow \overline{v_i} \in c_j$. For notational simplicity, we shall use H to refer to $H_2(\phi)$. Note that once again, by construction Observation 3.3 holds for H.

Moving on, we now show that ϕ is satisfiable if and only if $k = 2r$ PMUs can be placed on H such that H is fully observed under the condition that all PMUs are

cross-validated, and that $2r$ PMUs are the minimal bound for observing the graph with cross-validation.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. For each $1 \leq i \leq r$, if $v_i = True$ in A_ϕ we place a PMU at T_i^b and at T_i^t of the variable gadget V_i. Otherwise, we place a PMU at F_i^b and at F_i^t of this gadget. From the fact that A_ϕ is satisfying and Observation 3.3, we know the PMU nodes in V_i must be adjacent to some clause node[4], making T_i^t (F_i^t) two hops away from T_i^b (F_i^b). Therefore, all PMUs are cross-validated by XV2.

Assignment Φ_H observes all $v \in V$: from Lemma 3.6(b) we know the assignment fully observes all the variable gadgets. From Observation 3.3 we know all clause nodes are adjacent to a node with a PMU, so they are observed via O1, which concludes this direction of the theorem.

($\Leftarrow$) Suppose Φ_G observes all nodes in H under the condition that each PMU is cross-validated, and that $|\Phi_H| = 2r$. We want to show that ϕ is satisfiable by the truth assignment derived from Φ_H. We do so following a similar method as for the previous Theorems.

From Lemma 3.6(a) we know that each component in each variable gadget must have at least one PMU in order for the entire graph to be observed. Since we have $2r$ PMUs and $2r$ components, each component will have a single PMU. This also means there are no PMUs on clause gadgets.

From Lemma 3.6(b) we know that full observability will require PMUs be on either T or F nodes in each variable gadget. As a result, cross-validation constraints require for each variable gadget that both PMUs are either on T_i^t, T_i^b or F_i^t, F_i^b. This is because any T_i^t (F_i^t) is four hops or more away from any other T/F node. Since

[4]Each variable must be used in at least a single clause, or it is not considered part of the formula. If there is a variable that has no impact on the truth value of ϕ, we always place the PMUs on two nodes (both T or both F) that are adjacent to a clause node.

we assume the clause nodes are all observed and we know no PMUs are on clause nodes, from Observation 3.3 this means the PMU placement satisfies all clauses, which concludes our proof. □

3.3.5 The MaxObserve-XV Problem

The MaxObserve-XV optimization and decision problems are defined below:

MaxObserve-XV Optimization Problem:

Input: Graph $G = (V, E)$ where $V = V_Z \cup V_I$ and k PMUs such that $1 \leq k < k^*$.

Output: A placement of k PMUs, Φ_G, such that $|\Phi_G^R|$ is maximum under the condition that each $v \in \Phi_G$ is cross-validated according to the rules specified in Section 3.2.3.

MaxObserve-XV Decision Problem:

Instance: Graph $G = (V, E)$ where $V = V_Z \cup V_I$, k PMUs such that $1 \leq k < k^*$, and some $m < |V|$.

Question: Is there a Φ_G such that $|\Phi_G| \leq k$ and $m \leq |\Phi_G^R| < |V|$ under the condition that each $v \in \Phi_G$ is cross-validated?

Theorem 3.7. MaxObserve-XV *is NP-Complete.*

Proof Idea: We show MaxObserve-XV is NP-hard by reducing from P3SAT. Our proof is a combination of the NP-hardness proofs for MaxObserve and FullObserve-XV. From a P3SAT formula, ϕ, we create a graph $G = (V, E)$ with the clause gadgets from MaxObserve (Figure 3.3(b)) and the variable gadgets from FullObserve-XV (Figure 3.3(c)). The edges in G are identical the ones the graph created in our reduction for FullObserve-XV.

We show that any solution that observes $m = |V| - s$ nodes must place the PMUs exclusively on nodes in the variable gadgets. As a result, we show 1 node in each clause gadget – b_j for clause C_j – is not observed, yielding a total s unobserved nodes.

This implies all other nodes must be observed, and thus reduces our problem to the scenario considered in Theorem 3.5, which is already proven.

Proof. MaxObserve-XV is easily in $\mathcal{NP}$. We verify a MaxObserve-XV solution using the same polynomial time algorithm described in our proof for Theorem 3.5.

We reduce from P3SAT to show MaxObserve-XV is NP-hard. Our reduction is a combination of the reductions used for MaxObserve and FullObserve-XV. Given a P3SAT formula, ϕ, with variables $\{v_1, v_2, \ldots, v_r\}$ and the set of clauses $\{c_1, c_2, \ldots, c_s\}$, we form a new graph, $H_3(\phi) = (V(\phi), E(\phi))$ as follows. Each clause c_j corresponds to the clause gadget from MaxObserve (Figure 3.3(b)) and the variable gadgets from FullObserve-XV (Figure 3.3(c)). As in Theorem 3.5, we refer to the upper subgraph of variable gadget, V_i, as V_i^t and the lower subgraph as V_i^b. Also, we denote here $H := H_3(\phi)$.

Let $k = 2r$ and $m = 8r + s = |V| - s$. As in our NP-hardness proof for MaxObserve, m includes all nodes in H except b_j of each clause gadget. We need to show that ϕ is satisfiable if and only if $2r$ cross-validated PMUs can be placed on H such that $m \leq |\Phi_H^R| < |V|$.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. For each $1 \leq i \leq r$, if $v_i = True$ in A_ϕ we place a PMU at T_i^b and at T_i^t of the variable gadget V_i. Otherwise, we place a PMU at F_i^b and at F_i^t of this gadget. In either case, the PMU nodes in V_i must be adjacent to a clause node, making T_i^t (F_i^t) two hops away from T_i^b (F_i^b)[5]. Therefore, all PMUs are cross-validated by XV2.

This placement of $2r$ PMUs, Φ_H, is exactly the same one derived from ϕ's satisfying instance in Theorem 3.5. Since Φ_H only has PMUs on variable gadgets, all a_j nodes are observed by the same argument used in Theorem 3.5. Thus, at least $8r + s$ nodes are observed in H. Because no PMU in Φ_H is placed on a clause gadget, C_j,

[5]See previous note on FullObserve-XV

and O2 cannot be applied at a_j since $a_j \in V_I$, we know that no b_j is observed. We conclude that exactly m nodes are observed with Φ_H.

($\Leftarrow$) We begin by proving that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. Assume that there are $1 < t \leq r$ variable gadgets without a PMU. Then, at most t PMUs are on nodes in clause gadgets, so *at least* $\max(s-t, 0)$ clause gadgets are without PMUs. We want to show here that for $m = 8r + s$, $t = 0$.

To prove this, we rely on the following observations:

- As shown in Theorem 3.5, a variable gadget's subgraph with no PMU has at least 2 unobserved nodes.
- In any clause gadget C_j, b_j nodes cannot be observed if there is no PMU somewhere in C_j.

Thus, given some t, $|\Phi_H^-| \geq 2t + \max(s-t, 0)$, where Φ_H^- denotes the unobserved nodes in H. Since $|V| - m \leq s$, we know $|\Phi_H^-| \leq s$ and thus $s \geq 2t + \max(s-t, 0)$. We consider two cases:

- $s \geq t$: then we get $s \geq s + t \Rightarrow t = 0$.
- $s < t$: then we get $s \geq 2t$, and since we assume here $0 \leq s < t$ this leads to a contradiction and so this case cannot occur.

Thus, we have concluded that the $2r$ PMUs must be on variable gadgets, leaving all clause gadgets without PMUs. We now observe that for each clause gadget C_j, such a placement of PMUs cannot observe nodes of type b_j, which amounts to a total of s unobserved nodes – the allowable bound. This means that all other nodes in H must be observed in order for the requirement to be met. Specifically this is exactly all the nodes in $H_2(\phi)$ from the Theorem 3.5 proof. Since PMUs can only be placed on variable gadgets – all of which are included $H_2(\phi)$ – we have reduced the problem

to the problem in Theorem 3.5. We use the Theorem 3.5 proof to determine that all clauses in ϕ are satisfied by the truth assignment derived from Φ_H. □

3.3.6 Proving NPC for Additional Topologies

A quick review of our NPC proofs reveals that the graphs are carefully constructed regarding our selection of $|V_Z|, |V_I|$ and (where relevant) m. From a purely theoretical standpoint this is sufficient to prove that the class of problems is NPC. However, we argue that the NPC of these problems holds for a much wider range of topologies. To support this claim, in this section we show that slight adjustments to the variable and/or clause gadgets can generate a wide selection of graphs – changing $|V_Z|, |V_I|$ and (where relevant) m and $m/|V|$ – in which the same proofs from Section 3.3.2 - Section 3.3.5 can be applied. We present the outline for new gadget constructions and leave the detailed analysis to the reader.

The *number of injection nodes*, $|V_I|$, for each of our four problem definitions can be increased by introducing new variable gadgets. For FullObserve and MaxObserve, we use the variable gadget shown in Figure 3.4(a) in place of the original variable gadget (Figure 3.3(a)). Our proofs for Theorem 3.1 and Theorem 3.4 can remain largely unchanged because the same PMU placement described in each NP-Completeness proof observes these newly introduced nodes. [6] For FullObserve-XV and MaxObserve-XV we increase $|V_I|$ using the variable gadget shown in Figure 3.4(b). The PMU placements described in the proofs for Theorem 3.5 and Theorem 3.7 observe all newly introduced nodes in Figure 3.4(b).

Similarly, the *number of zero-injection nodes* $|V_Z|$ can modified by changing the variable gadgets. FullObserve and MaxObserve – using the variable gadget shown in Figure 3.5(a) – and FullObserve-XV and MaxObserve-XV – using the variable gadget shown in Figure 3.5(a) – are easily extended to include more zero-

[6] The PMU on a T_i or F_i node observes $I_{i1}, I_{i2}, \ldots, I_{ip}$ via O1.

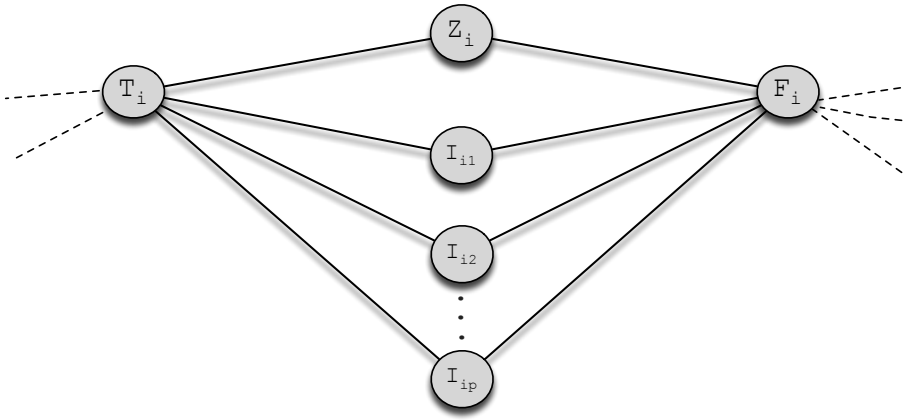
injection nodes. By repeatedly applying O2 at the newly introduced zero-injection nodes, all variable gadget nodes are observed using the same PMU placement described in the NP-Completeness proofs for each problem. For this reason, our proofs only require slight modifications.

In the MaxObserve-XV and MaxObserve proofs we demonstrated NPC for $m = |V| - s$. In order to increase the size of $|V|$ while keeping m the same, we replace each clause gadget, C_j for $1 \leq j \leq s$, with a new clause gadget, C'_j, shown in Figure 3.6. Note that all C'_j nodes are injection nodes. [7] In this new clause gadget, placing a PMU on any node but a_j results in the observation of at most 3 nodes. Using this simple insight, we can easily argue that more nodes are always observed by placing a PMU on the variable gadget rather than at a clause gadget. Then, we can argue that PMUs are only placed on variable gadgets and finally leverage the argument from Theorem 3.4 to show MaxObserve is NP-Complete for any $\frac{m}{|V|}$. A similar argument can be made for MaxObserve-XV.

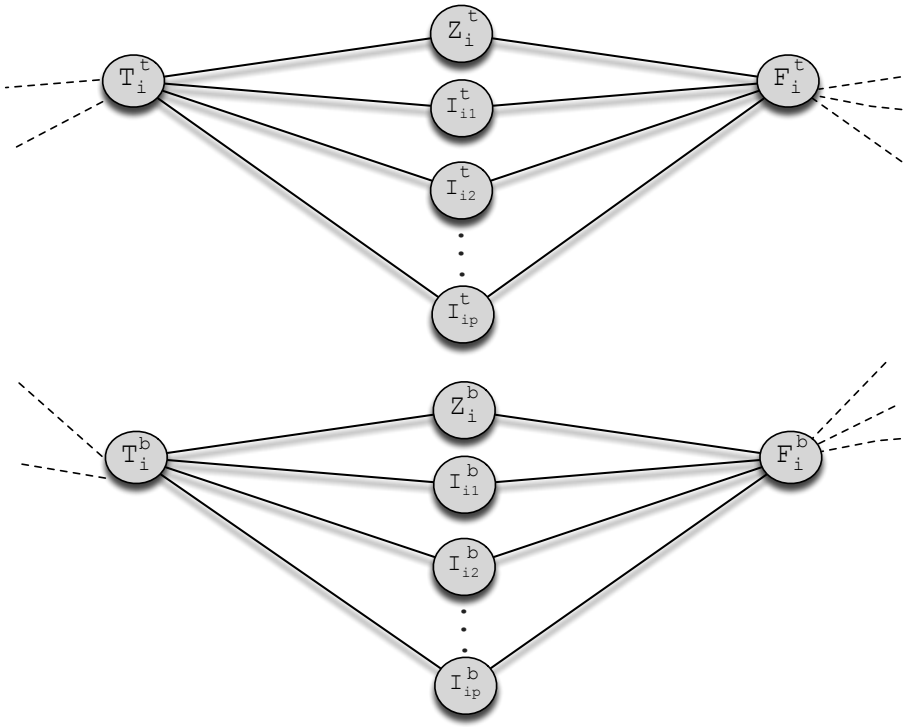
3.4 Approximation Algorithms

Because all four placement problems are NPC, we propose greedy approximation algorithms for each problem, which iteratively add a PMU in each step to the node that observes the maximum number of new nodes. We present two such algorithms, one that directly addresses MaxObserve (`greedy`) and the other MaxObserve-XV (`xvgreedy`). `greedy` and `xvgreedy` can easily be used to solve FullObserve and FullObserve-XV, respectively, by selecting the appropriate k value to ensure full observability. We prove these algorithms have polynomial complexity (i.e., they are in $\mathcal{P}$), making them feasible tools for approximating optimal PMU placement.

[7] Other modifications exist for the clause gadgets that do not involve solely injection nodes, with similar results.

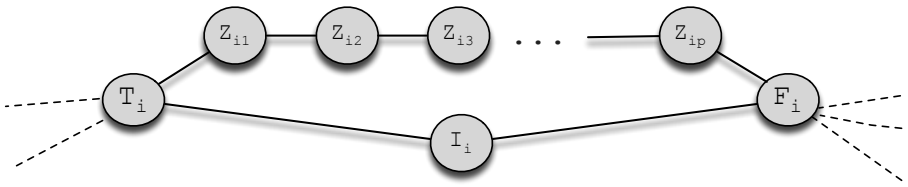


(a) Modified variable gadget used in FullObserve and MaxObserve, containing additional injection nodes: $I_{i1}, I_{i2}, \ldots I_{ip}$.

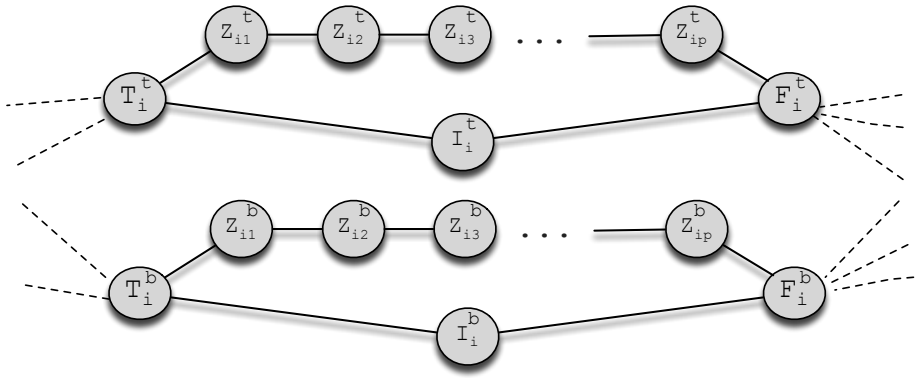


(b) Modified variable gadget used in FullObserve-XV and MaxObserve-XV. Each disconnected subgraph has additional injection nodes: nodes $I^t_{i1}, I^t_{i2}, \ldots I^t_{ip}$ are added to the upper subgraph and nodes $I^b_{i1}, I^b_{i2}, \ldots I^b_{ip}$ are included in the bottom subgraph.

Figure 3.4. Figures for variable gadget extensions to include more injection nodes described in Section 3.3.6. The dashed edges indicate connections to clause gadget nodes.



(a) Modified variable gadget used in FullObserve and MaxObserve, containing additional injection nodes: $Z_{i1}, Z_{i2}, \ldots Z_{ip}$.



(b) Modified variable gadget used in FullObserve-XV and MaxObserve-XV. Each disconnected subgraph has additional injection nodes: the upper subgraph includes nodes $Z^t_{i1}, Z^t_{i2}, \ldots Z^t_{ip}$ and nodes $Z^b_{i1}, Z^b_{i2}, \ldots Z^b_{ip}$ are added in the bottom subgraph.

Figure 3.5. Figures for variable gadget extensions to include more non-injection nodes described in Section 3.3.6. The dashed edges indicate connections to clause gadget nodes.

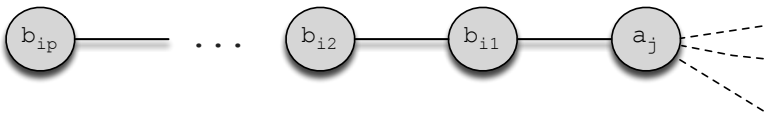


Figure 3.6. Extended clause gadget, C'_j, used in Section 3.3.6. All nodes are injection nodes.

Lastly, we explore the possibility that the PMU observability rules are submodular functions (Section 3.4.2).

3.4.1 Greedy Approximations

`greedy` **Algorithm**. We start with $\Phi = \emptyset$. At each iteration, we add a PMU to the node that results in the observation of the maximum number of new nodes. The algorithm terminates when all PMUs are placed. [8] The pseudo-code for `greedy` can be found in Appendix B.2 (Algorithm B.2.1).

Theorem 3.8. *For input graph $G = (V, E)$ and k PMUs* `greedy` *has $O(dkn^3)$ complexity, where $n = |V|$ and d is the maximum degree node in V.*

Proof. The proof can be found in Appendix B.2 (Theorem B.4). □

`xvgreedy` **Algorithm**. `xvgreedy` is almost identical to `greedy`, except that PMUs are added in pairs such that the selected pair observe the maximum number of nodes under the condition that the PMU pair satisfy one of the cross-validation rules. We provide the pseudo code for `xvgreedy` in Algorithm B.2.2.

Theorem 3.9. *For input graph $G = (V, E)$ and k PMUs* `xvgreedy` *has $O(kdn^3)$ complexity, where $n = |V|$ and d is the maximum degree node in V.*

Proof. This theorem is proved in Appendix B.2 (Theorem B.5). □

3.4.2 Observability Rules as Submodular Functions?

Submodular functions are set functions with diminishing marginal returns: the value that each subsequent element adds decreases as the size of the input set increases. More formally, let X be a ground set such that $|X| = n$. We define a set

[8] The same greedy algorithm is proposed by Aazami and Stilp [4] and is shown to $\Theta(n)$ approximation ratio under the assumption that all nodes are zero-injection.

function on X as $f : 2^X \to \mathbb{R}$. Using the definition from Dughmi [24] f is *submodular* if, for all $A, B \subseteq X$ with $A \subseteq B$, and for each $j \in X$,

$$f(A \cup \{j\}) - f(A) \quad \geq \quad f(B \cup \{j\}) - f(B) \tag{3.2}$$

It has been shown that greedy algorithms admit a $1 - 1/e$ approximation of submodular functions [63], where e is the base of the natural logarithm. For this reason, we aim to show that our observability rules are submodular.

For the PMU placement problem, consider $G = (V, E)$. For $S \subseteq V$ we define $f(S)$ as the number of observed nodes derived by placing a PMU at each $s \in S$. We prove that f is not submodular for graphs containing zero-injection nodes (Theorem 3.10) but is submodular when restricted to graphs with only injection nodes (Theorem 3.11).

Theorem 3.10. *f is not submodular for graphs, G_z, with zero-injection nodes.*

Proof. Let G_z be the graph from Figure 3.7, $A = \{a\}$, and $B = \{a, b\}$. Then,

$$\begin{aligned} f(A \cup \{c\}) - f(A) &\overset{?}{\geq} f(B \cup \{c\}) - f(B) \\ f(A \cup \{c\}) - 2 &\overset{?}{\geq} f(B \cup \{c\}) - 3 \\ 3 - 2 &\overset{?}{\geq} 8 - 3 \\ 1 &\overset{?}{\geq} 5 \end{aligned}$$

We conclude that f is not submodular for G_z. □

Note that in this example, O2 prevented us from meeting the criteria for submodular functions. For PMU placement $B \cup \{c\}$, we were able to apply O2 at e, resulting in the observation of the chain of nodes at the top of the graph. However, we were unable to apply O2 for the PMU placement $A \cup \{c\}$. This observation provides the motivation for our next Theorem (3.11).

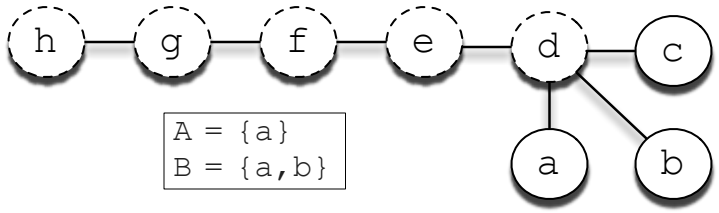


Figure 3.7. Example used in Theorem 3.10 showing a function defined using our observability rules is not submodular for graphs with zero-injection nodes. Nodes with a dashed border are zero-injection nodes and injection nodes have a solid border. For set function $f : 2^X \to \mathbb{R}$, defined as the number of observed nodes resulting from placing a PMU at each $x \in X$, we have $f(A) = f(\{a\}) = 2$ where $\{a, d\}$ are observed, while $f(B) = f(\{a, b\}) = 3$ where $\{a, b, d\}$ are observed.

Theorem 3.11. *f is a submodular function for graphs, G_I, containing only injection nodes.*

Proof. Consider a graph $G_I = (V_I, E_I)$ where each $v \in V_I$ is an injection node. Let $A \subseteq B \subseteq V_I$ and $j \in V_I$. Placing a PMU at j can at most result in the observation of $j \cup \Gamma(j)$ because we cannot apply O2 in G_I since we have assumed all nodes are injection nodes. We claim that any $x \in j \cup \Gamma(j)$ that is unobserved after placing a PMU at nodes in B is not observed with the PMU placement derived from A. x is unobserved only if x has no PMU nor if any $\Gamma(x)$ has a PMU. Since $A \subseteq B$ and we have assumed x is not observed using B, it must be the case that x is not observed under A. Since we have show that all unobserved nodes resulting from PMU placement B must be unobserved under A, we conclude that $f(A \cup \{j\}) - f(A) \geq f(B \cup \{j\}) - f(B)$ and, therefore, f is submodular for G_I. □

3.5 Simulation Study

Topologies. We evaluate our approximation algorithms using simulations over IEEE topologies as well as synthetic ones. For IEEE topologies, we use bus systems

14, 30, 57, and 118 [9]. The bus system number indicates the number of nodes in the graph (e.g., bus system 57 has 57 nodes). Synthetic graphs are then generated based on each of these topologies, and are used to quantify the performance of our greedy approximations.

Since observability is determined by the connectivity of the graph, we use the *degree distribution* of IEEE topologies as the template for generating our synthetic graphs. A synthetic topology is generated from a given IEEE graph by randomly "swapping" edges in the IEEE graph. Specifically, we select a random $v \in V$ and then pick a random $u \in \Gamma(v)$. Let u have degree d_u. Next, we select a random $w \notin \Gamma(v)$ with degree $d_w = d_u - 1$. [10] Finally, we remove edge (v, u) and add (v, w), thereby preserving the node degree distribution. We continue this swapping procedure until the original graph and generated graph share *no edges*, and then return the resulting graph.

Evaluation Methods. We are interested in evaluating how close our algorithms are to the optimal PMU placement. Thus, when computationally possible (for a given k) we use brute-force algorithms to iterate over all possible placements of k PMUs in a given graph and select the best PMU placement. When computationally infeasible, we present only the performance of the greedy algorithm without corresponding optimal solutions. In what follows, the output of the brute-force algorithm is denoted `optimal`, and when we require cross-validation it is denoted `xvoptimal`.

We present three different simulations in Section 3.5.1-3.5.3. In Section 3.5.1 we consider performance as a function of the number of PMUs, and in Section 3.5.2 we investigate the performance impact of the number of zero-injection nodes in the network. These two sections are performed over sets of synthetic graphs. We conclude

[9] http://www.ee.washington.edu/research/pstca/

[10] Here "random" means uniformly at random.

in Section 3.5.3 where we compare these results to the performance over the actual IEEE graphs.

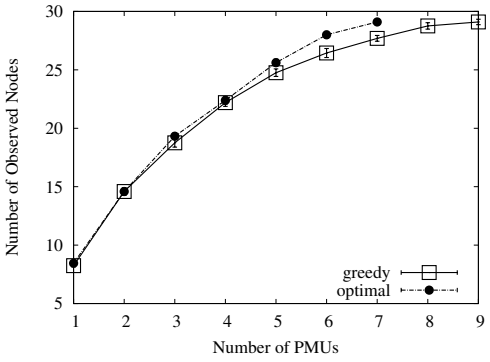
3.5.1 Simulation 1: Impact of Number of PMUs

In the first simulation scenario we vary the number of PMUs and determine the number of observed nodes in the synthetic graph. Each data point is generated as follows. For a given number of PMUs, k, we generate a graph, place k PMUs on the graph, and then determine the number of observed nodes. We continue this procedure until $[0.9(\overline{x}), 1.1(\overline{x})]$ – where $\overline{x}$ is the mean number of observed nodes using k PMUs – falls within the 90% confidence interval.

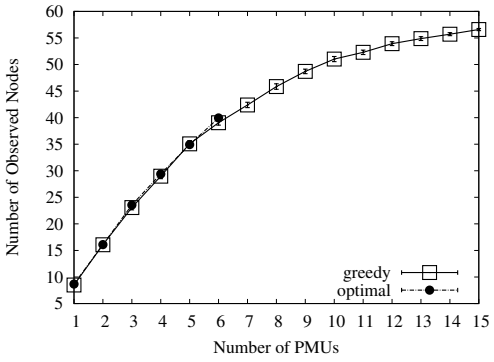
In addition to generating a topology, for each synthetic graph we determined the members of V_I, V_Z. These nodes are specified for the original graphs in the IEEE bus system database. Thus, we randomly map each node in the IEEE network to a node in the synthetic network with the same degree, and then match their membership to either V_I or V_Z.

We present here results for solving MaxObserve and MaxObserve-XV. The number of nodes observed given k, using `greedy` and `optimal`, are shown in Figure 3.8, and Figure 3.9 shows this number for `xvgreedy` and `xvoptimal`. In both sets of plots we show 90% confidence intervals. We omit results for graphs based on IEEE bus 14 because the same trends are observed.

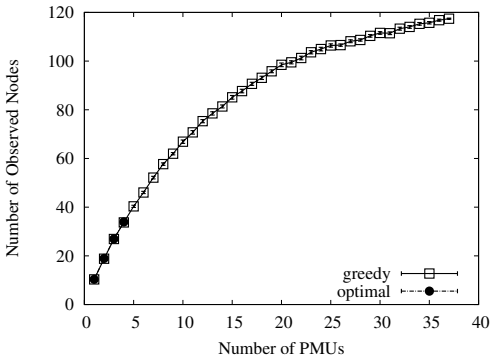
Our greedy algorithms perform well. On average, `greedy` is within 98.6% of `optimal`, is never below 94% of `optimal`, and in most cases gives the optimal result. Likewise, `xvgreedy` is never less than 94% of `xvoptimal` and on average is within 97% of `xvoptimal`. In about about half the cases `xvgreedy` gives the optimal result. These results suggest that despite the complexity of the problems, a greedy approach can return high-quality results. Note, however, that these statistics do not include performance over large topologies (i.e., IEEE graphs $57, 118$) when k is large. It is



(a) Graphs based on IEEE Bus 30



(b) Graphs based on IEEE Bus 57

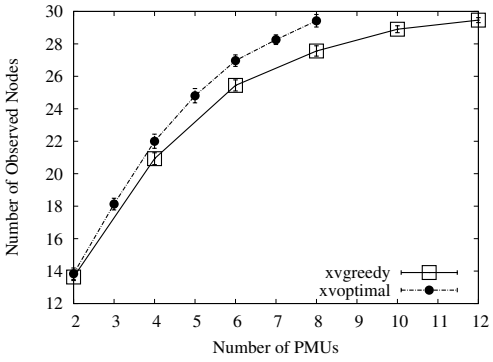


(c) Graphs based on IEEE Bus 118

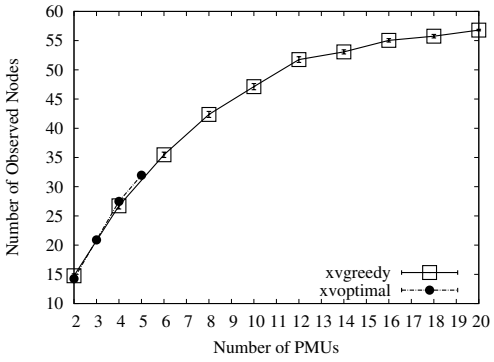
Figure 3.8. Mean number of observed nodes over synthetic graphs – using `greedy` and `optimal` – when varying number of PMUs. The 90% confidence interval is shown.

an open question whether the greedy algorithms used here would do well for larger graphs.

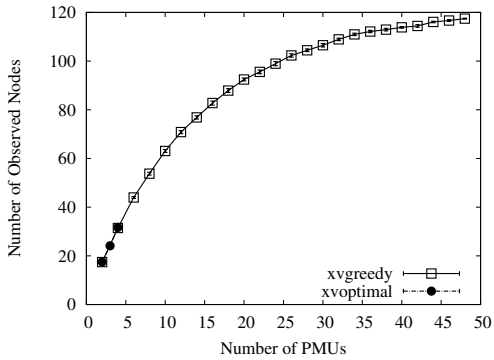
Surprisingly, when we compare our results with and without the cross-validation requirement, we find that this set of constraints does not have a significant effect on the number of observed nodes for the same k. Our experiments show that on average `xvoptimal` observed only 5% fewer nodes than `optimal`. Similarly, on average `xvgreedy` observes 5.7% fewer nodes than `greedy`. This suggests that the cost of imposing this requirement is low, with the clear gain of ensuring PMU correctness across the network via cross-validation.



(a) Graphs based on IEEE Bus 30



(b) Graphs based on IEEE Bus 57



(c) Graphs based on IEEE Bus 118

Figure 3.9. Over synthetic graphs, mean number of observed nodes – using `xvgreedy` and `xvoptimal` – when varying number of PMUs. The 90% confidence interval is shown.

3.5.2 Simulation 2: Impact of Number of Zero-Injection Nodes

Next, we examine the impact of $|V_Z|$ on algorithm performance. For each synthetic graph, we run our algorithms for increasing values of $|V_Z|$ and determine the minimum number of PMUs needed to observe all nodes in the graph (k^*). For each $z := |V_Z|$, we select z nodes uniformly at random to be zero-injection, and the rest are in V_I. Because we compute k^* here, we solve FullObserve and FullObserve-XV, rather than MaxObserve and MaxObserve-XVas in Simulation 1.

We generate each data point using a similar procedure to the one described in Section 3.5.1. For each $z = z_i$, we generate a graph and determine k^*. We then

compute $\overline{k^*}$, the mean value of k^* over all simulation runs with $|V_Z| = z_i$. We continue this procedure until $[0.9(\overline{k^*}), 1.1(\overline{k^*})]$ falls within the 90% confidence interval.

Figure 3.10(a) shows the simulation results for solving FULLOBSERVE and FULLOBSERVE-XV on synthetic graphs modeled by IEEE bus 57. Results for other topologies considered here (i.e., 14, 30 and 118) followed the same trend and are thus omitted. Due to the exponential running time of `optimal` and `xvoptimal`, we present here only results of our greedy algorithms.

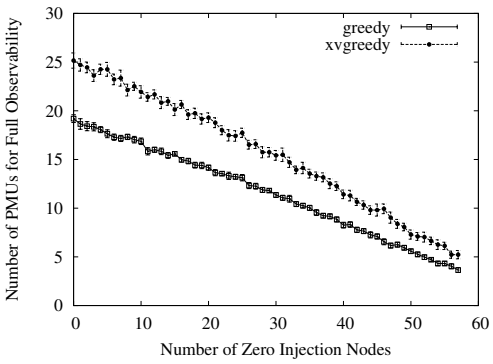
As expected, increasing the number of zero-injection nodes – for both `greedy` and `xvgreedy` – reduces the number of PMUs required for full observability. More zero-injection nodes allow O2 to be applied more frequently (Figure 3.10(b)), thereby increasing the number of observed nodes without using more PMUs. In fact, we found the relationship between $|V_Z|$ to the greedy estimate of k^* to be linear.

The gap in k^* between `greedy` and `xvgreedy` decreases as z grows. `greedy` and `xvgreedy` observe a similar number of nodes via O2 across all z values: the mean absolute difference in the number of nodes observed by O2 between the two algorithms is 1.66 nodes. Thus, as z grows the number of nodes observed by O2 accounts for an increasing proportion of all observed nodes (Figure 3.10(b)), causing the gap between `greedy` and `xvgreedy` to shrink.

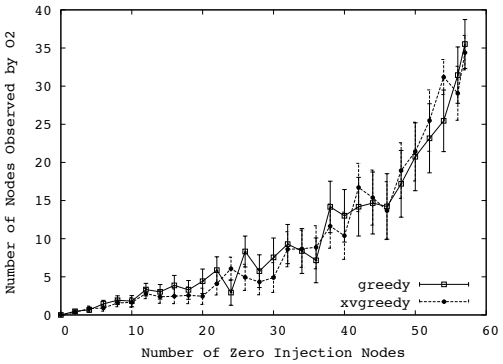
3.5.3 Simulation 3: Synthetic vs Actual IEEE Graphs

In this section, we compare our results with the performance over the original IEEE systems. We assign nodes to V_Z and V_I as specified in the IEEE database files. Our results indicate that the trends we observed over the synthetic graphs apply as well to real topologies.

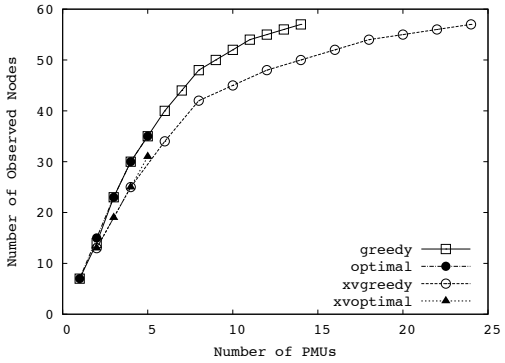
Figure 3.10(c) shows the number of observed nodes for the `greedy`, `xvgreedy`, `optimal`, and `xvoptimal` algorithms for IEEE bus system 57. `greedy` and `xvgreedy` observe nearly as many nodes as the corresponding optimal solution. In many cases,



(a) Simulation 2: Number of PMUs needed for full observability for different $|V_Z|$ values, using synthetic graphs based on IEEE Bus 57.



(b) Simulation 2: Number of nodes observed by O2 for different $|V_Z|$ values, using synthetic graphs based on IEEE Bus 57.



(c) Simulation 3: Number of observed nodes when varying number of PMUS, using IEEE Bus 57

Figure 3.10. Results for Simulation 2 and 3. In Figures (a) and (b) the 90% confidence interval is shown.

greedy yields the optimal placement. Similarly, as with the synthetic graphs, the number of PMUs required to observe all nodes decreases linearly as $|V_Z|$ increases. [11]

To compare the actual values for synthetic graphs to those over IEEE graphs, we took the mean absolute difference between the results, and normalized by the result for the synthetic graph. For example, let n_k be the mean number of observed nodes using `greedy` over all synthetic graphs with input k, and let $n_{G,k}$ be the output of

[11]The same trends were observed using IEEE bus systems 14, 30, and 118.

	greedy	xvgreedy	optimal	xvoptimal
Simulation 1	4%	4.6%	6%	7.6%
Simulation 2	9.1%	16.1%	N/A	N/A

Table 3.1. Mean absolute difference between the computed values from synthetic graphs and IEEE graphs, normalized by the result for the synthetic graph.

greedy for IEEE graph G and k. We compute $n_{d,k} = (|n_k - n_{G,k}|)/n_k$. Finally, we calculate the mean over all $n_{d,k}$. This process is done for each algorithm we evaluate. The resulting statistics can be found in Table 3.1. The small average difference between the synthetic graphs and the actual IEEE topologies suggests that the node degree distribution of the IEEE graph is an effective feature for generating similar synthetic graphs.

3.6 Related Work

FullObserve is well-studied [9, 14, 36, 60, 77]. Haynes et al. [36] and Brueni and Heath [14] both prove FullObserve is NPC. However, their proofs make the unrealistic assumption that all nodes are zero-injection. We drop this assumption and thereby generalize their NPC results for FullObserve. Additionally, we leverage the proof technique from Brueni and Heath [14] in all four of our NPC proofs, although our proofs differ considerably in their details.

MaxObserve and FullObserve are closely related to the problem of finding a dominating set of minimum size [33]. A dominating set of an undirected graph, $G = (V, E)$, is a set of nodes S such that every $v \in V$ is either in S or has a neighbor in S. Finding the minimum size dominating set is equivalent to finding the minimum number of PMUs required to observe all graph nodes (FullObserve) where nodes can only be observed using observability rule 1. Aazami and Stilp [4] extend the dominating set problem to account for observability rule 2. This new formulation,

called the power dominating set problem, is equivalent to the general FullObserve problem formulation where nodes can be observed by observability rule 1 and 2.

In the power systems literature, Xu and Abur [77, 78] use integer programming to solve FullObserve, while Baldwin et al. [9] and Mili et al. [60] use simulated annealing to solve the same problem. All of these works allow nodes to be either zero-injection or non-zero-injection. However, these papers make no mention that FullObserve is NPC, i.e., they do not characterize the fundamental complexity of the problem.

Aazami and Stilp [4] investigate approximation algorithms for FullObserve. They derive a hardness approximation threshold of $2^{\log^{1-\epsilon} n}$. Aazami and Stilp also prove that `greedy`, from Section 3.4, is a $\Theta(n)$-approximation. However, this performance ratio is derived under the assumption that all nodes are zero-injection.

Chen and Abur [18] and Vanfretti et al. [75] both study the problem of bad PMU data. Chen and Abur [18] formulate their problem differently than FullObserve-XV and MaxObserve-XV. They consider fully observed graphs and add PMUs to the system to make all existing PMU measurements non-critical (a critical measurement is one in which the removal of a PMU makes the system no longer fully observable). Vanfretti et al. [75] define the cross-validation rules used in this chapter. They also derive a lower bound on the number of PMUs needed to ensure all PMUs are cross-validated and the system is fully observable.

3.7 Conclusions

In this chapter, we formulated four PMU placement problems and proved that each one is NPC. Consequently, future work should focus on developing approximation algorithms for these problems. As a first step, we presented two simple greedy algorithms: `xvgreedy` which considers cross-validation and `greedy` which does not.

Both algorithms iteratively add PMUs to the node which observes the maximum of number of nodes.

Using simulations, we found that our greedy algorithms consistently reached close-to-optimal performance. Our simulations also showed that the number of PMUs needed to observe all graph nodes decreases linearly as the number of zero-injection nodes increase. Finally, we found that cross-validation had a limited effect on observability: for a fixed number of PMUs, `xvgreedy` and `xvoptimal` observed only 5% fewer nodes than `greedy` and `optimal`, respectively. As a result, we believe imposing the cross-validation requirement on PMU placements is advised, as the benefits they provide come at a low marginal cost.

There are several topics for future work. The success of the greedy algorithms suggests that bus systems have special topological characteristics, and we plan to investigate their properties. Additionally, we intend to implement the integer programming approach proposed by Xu and Abur [77] to solve FullObserve. This would provide valuable data points to measure the relative performance of `greedy`.

CHAPTER 4

RECOVERY FROM LINK FAILURES IN A SMART GRID COMMUNICATION NETWORK

4.1 Introduction

In this chapter we continue our study of issues related to PMU sensors. In the previous chapter, we proposed and evaluated algorithms to ensure that PMU measurements are observed in the first place and are correct. Now we consider algorithms that take these (correct) PMU measurements and disseminate them quickly and reliably to power grid operators, utility companies, and power grid managing and monitoring entities.

PMU applications have stringent, and in many cases ultra-low, *per-packet* delay and loss requirements. If these per-packet delay requirements are not met, PMU applications can miss a critical power grid event (e.g., lightning strike, power link failure), potentially leading to a cascade of incorrect decisions and corresponding actions. For example, closed-loop control applications require delays of $8 - 16$ ms per-packet [8]. If *any* packet is not received within this time window, the closed-loop control application may take a wrong control action. In the worst case, this can lead to a cascade of power grid failures similar to the August 2003 blackout in North America [7] and the recent power grid failures in India [79].

As a result of this sensitivity, the communication network that disseminates PMU data must provide hard end-to-end data delivery guarantees [8]. For this reason, the Internet's best-effort service model alone is unable to meet the stringent packet delay and loss requirements of PMU applications [12]. Instead, either a new network

architecture or enhancements to the existing Internet architecture and its protocols are needed [8, 12, 13, 39] to provide efficient, in-network forwarding and fast recovery from link and switch failures. Additionally, multicast should figure prominently in data delivery, since PMUs disseminate data to applications across many locations [8].

Software-defined networking (SDN) provides a vehicle for this type of innovation by providing programmable access to the forwarding plane of network switches and routers. New network services are defined in a programmable control plane, which SDN cleanly separates from the data plane (e.g., forwarding), and are instantiated as forwarding rules installed at network switches. The communication between the control and data planes, including the messaging to install forwarding rules, are typically managed by the OpenFlow protocol [58]. [1]

This separation of control and data planes is similar in spirit to the approach used by the Gridstat system [8] that also manages data plane actions through a separate control plane. Gridstat is a publish-subscribe system designed specifically for disseminating critical power grid data; however, because Gridstat is an overlay service built on top of existing network protocols, Gridstat alone cannot meet the delivery requirements of PMU applications. Rather, the underlying network protocols themselves must also be improved. This is the emphasis of our research here.

In this chapter, we use OpenFlow to define and implement new control plane algorithms, tailored specifically for disseminating critical power grid data, that program data plane forwarding by installing forwarding rules at network switches. We focus on algorithms for fast recovery from link failures. Informally, a link that does not meet its packet delivery requirement (either due to excessive delay or actual packet loss) is considered failed. We propose, design, and evaluate solutions to all three aspects

[1] Protocols other than OpenFlow can be used. OpenFlow is the first and most popular protocol used to interface between SDN control and data planes.

of link failure recovery: link failure detection, algorithms for pre-computing backup multicast trees, and fast backup tree installation.

We make the following contributions in this chapter:

- **Design a link-failure detection algorithm**. We design a link-failure detection and reporting mechanism, PCOUNT, that uses OpenFlow [58] to detect link failures when and where they occur, *inside* the network. In-network detection is used to reduce the time between when the loss occurs and when it is detected. In contrast, most previous work [5, 16, 30] focuses on measuring end-to-end packet loss, resulting in slower detection times.

- **Formulate a novel optimization problem for computing backup multicast trees.** Inspired by MPLS fast-reroute algorithms that quickly reroute time-critical unicast IP flows over pre-computed backup paths [69], we formulate a new problem, MULTICAST RECYCLING, that pre-computes backup multicast trees, to be used after a link failure, with the aim of minimizing the control overhead required to install the backup trees. This optimization criteria differs from those proposed in the literature [20, 27, 59, 67, 76] that use optimization criteria specified over a *single* multicast tree and typically emphasize maximizing node (link) disjointedness between the backup and primary path, while we consider conditions specified across *multiple* multicast trees.

- **Prove Multicast Recycling is at least NP-hard and propose an approximation algorithm, Bunchy, for Multicast Recycling.** BUNCHY uses an approximation to the directed Steiner Tree problem taken from the literature [17] to compute backup trees and modifies link weights to encourage backup trees to reuse existing forwarding rules installed in the network. Doing so reduces both the number of control messages the controller must send to install each backup tree and the number of forwarding rules maintained at each switch.

- **Propose Merger, an OpenFlow implementation of multicast that aims to reduce forwarding state.** MERGER uses local optimization to create a near minimal set of forwarding rules by "merging" forwarding rules in cases where multiple multicast trees have common forwarding behavior.

- **Design two algorithms – Proactive and Reactive – for fast backup tree installation.** PROACTIVE pre-installs backup tree forwarding rules and activates these rules after a link failure is detected, while, REACTIVE installs backup trees *after* a link a failure is detected. We show how MERGER can be applied to PROACTIVE and REACTIVE to reduce the amount of PROACTIVE pre-installed forwarding state and decrease REACTIVE signaling overhead.

- **Provide a prototype implementation of our algorithms, Appleseed, using POX and evaluate each algorithm using Mininet.** PCOUNT, BUNCHY, MERGER, PROACTIVE, and REACTIVE are implemented in POX [57], an open-source OpenFlow controller.

  We use emulations based on Mininet [50] to evaluate our algorithms over synthetic graphs and actual IEEE bus systems. We find that PCOUNT provides fast and accurate link loss estimates: after sampling only 75 packets, the 95% confidence interval is within 15% of the true loss probability. Additionally, we find PROACTIVE yields faster recovery than REACTIVE (REACTIVE sends up to 10 times more control messages than PROACTIVE) but at the cost of storage overhead at each switch (in our emulations, pre-installed backup trees can account for as much as 35% of the capacity a conventional OpenFlow switch [21]). Lastly, we observe that MERGER reduces control plane messaging and the amount of pre-installed forwarding state by a factor of 2 to 2.5 when compared to a standard multicast implementation, resulting in faster installation and smaller flow table sizes.

The remainder of this chapter is structured as follows. In the following section (Section 4.2), we provide necessary background on PMU application requirements and OpenFlow, as well as introduce a running example used later to describe our algorithms. Then, we outline our algorithms in Section 4.3: Section 4.3.1 details our link-failure detection algorithm called PCOUNT; in Section 4.3.2, we outline our algorithms for computing backup multicast trees; and then describe algorithms for installation backup trees in Section 4.3.3; Section 4.3.5 presents MERGER, a fast multicast implementation when applied to our backup tree installation algorithms can significantly improve performance. Next, we briefly survey relevant literature (Section 4.4). Our emulation study is presented in Section 4.5. Section 4.6 concludes this chapter with a summary.

4.2 Preliminaries

In this section, we provide the necessary background to describe our algorithms in Section 4.3. First, we describe several PMU applications and their QoS requirements, including justification for multicasting PMU measurement data (Section 4.2.1). Then, we present a motivating example referenced throughout this chapter (Section 4.2.2). Section 4.2.3 defines terms and notation. We give an overview of OpenFlow in Section 4.2.4 and Section 4.2.5 details our OpenFlow multicast implementation.

4.2.1 PMU Applications and Their QoS Requirements

In this work, we consider the design of a communication network for disseminating critical Smart Grid data, principally data associated with PMU applications. Here we detail the QoS requirements of a few of these applications, with a particular emphasis on PMU applications with the most stringent end-to-end delay requirements. Table 4.1 shows packet delay and frequency requirements of three PMU applications, each described below.

PMU Application	E2E Delay	Rate (Hz)
SIPS	$8-16$ ms	$120-720+$
Wide Area Control	$5-50$ ms	$1-240$
Anti-Islanding	$5-50$ ms	$30-720+$

Table 4.1. PMU applications and their QoS requirements [8]. The end-to-end (E2E) delay requirement is *per-packet*, as advocated by Bakken et al. [8].

Example PMU Applications and Their QoS Requirements. System Integrity Protection Scheme (SIPS) applications ensure that the entire power grid remains in a healthy state after local power grid components (e.g., relays) have taken mitigating actions to remedy local emergencies. To do so, SIPS applications require accurate and timely PMU measurements to identify system instability and to take correct mitigating actions (e.g., trip power generation) [8].

Islanding is a safety measure commonly used in power systems that separates entire sections (i.e., an island) of the power grid from the larger system in periods of voltage and frequency instability. Due to a lack of real-time situational awareness, power grid operators and oversight bodies mandate a conservative approach of disconnecting all distributed generation sources (e.g., wind and solar) from a locally islanded system. PMU measurements can provide the situational awareness to allow more narrowly targeted islands to be identified. Moreover, as distributed generation increases (as is the current trend), simultaneously disconnecting large number of generation sources may actually further destabilize the grid. PMUs are critical to future anti-islanding applications that will use PMU data to determine when distributed generation sources can safely remain online and thereby help preserve power grid stability [8].

Wide-area control is a more general category of PMU applications, referring to applications that gather PMU data from disparate sources to determine the health of the power grid and initiate control actions, both in real-time. For example, Southern

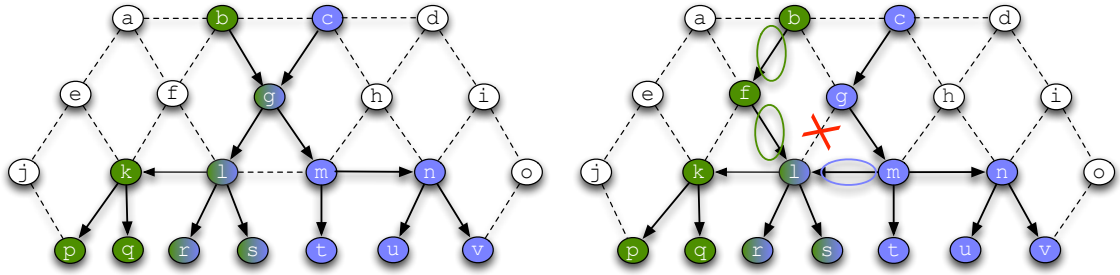
California Edison [43] measures (using PMUs) and controls voltage at remote locations to ensure voltage safety limits are met for its bulk energy transfers. Another wide-area control application uses PMUs to measure, detect, and dampen inter-area oscillations in real-time[8].

Why Multicast is Needed. As noted by Bakken et al. [8], PMU measurements are increasingly needed at multiple locations (e.g., utility companies, balancing authorities) and by many different power grid applications making multicast a natural fit. In today's electric power grid, the set of receivers that would require real-time PMU measurements is relatively small and localized (near the location of a PMU). However, as the power grid evolves to make distributed power generation (e.g., wind and solar) a larger portion of its energy portfolio, we believe PMU measurements will need to be multicasted to a larger set of receivers in order for these distributed sources of power to be safely integrated into the power grid. It is in this context that we define our PMU data dissemination problem and corresponding algorithms.

4.2.2 Motivating Example

Here we present a motivating example to highlight different aspects of the challenges addressed in this work. This example is used throughout this chapter to help describe our algorithms.

Figure 4.1 shows two source-based multicast trees used to disseminate PMU measurement data produced by a PMU at b and another at c. The multicast tree, T_b, shown in green, is rooted at b and disseminates b's PMU data to its leaf nodes (i.e., data sinks) $\{p, q, r, s\}$. The blue multicast tree, T_c, multicasts c's PMU measurements to its leaf nodes $\{r, s, t, u, v\}$. Half-green/half-blue nodes are in both T_b and T_c. Dashed and solid lines indicate network links, with links in the multicast tree marked by arrows. Before any link failures occur, the original multicast trees (Figure 4.1(a)) meets the delay requirements specified by each data sink.



(a) Original multicast trees T_b (green) and T_c (blue).

(b) Backup multicast trees – $\hat{T}_b$ and $\hat{T}_c$ for T_b and T_c, respectively – installed after (g, l) fails. New links for each backup tree are circled.

Figure 4.1. Example problem scenarios with two source-based multicast trees, one rooted at b (in green), T_b, and the other at c (in blue), T_c. Half-blue/half-green nodes are members of both multicast trees. Let f_b and f_c denote the multicast flows corresponding to T_b and T_c, respectively. The multicast trees are shown before and after link (g, l) fails.

At some point, link (g, l) fails (e.g., its loss rate exceeds a threshold or it goes completely offline). This prevents p, q, r and s from receiving any packets from either T_b or T_c until each multicast tree is repaired, leaving the delay requirement of each these sink nodes unsatisfied. Figure 4.1(b) shows two backup multicast trees – one for T_b and the other for T_c – installed after it is detected that (g, l) has failed. Notice that each backup tree contains no path using the failed link, (g, l), and has a path between its root and each of its data sinks. In the following sections we present algorithms that detect these types of link failures, compute backup multicast trees such as those shown in Figure 4.1(b), and quickly install these backup trees in order to minimize packet loss and delay.

4.2.3 Notation and Assumptions

We model the communication network as a directed graph $G = (V, E)$, where $(u, d) \in E$ denotes a directed edge from u to d and V consists of three types of nodes: ones that send PMU data (PMU nodes), nodes that receive PMU data (data sinks), and switches connecting PMU nodes and data sinks (typically via other switches).

We assume G has $m \geq 1$ source-based multicast trees to disseminate PMU data. Let $T = \{T_1, T_2, \ldots, T_m\}$ refer to the set of m source-based multicast trees in G such that each $T_i = (V_i, E_i, r, S)$ is a tree rooted at r with directed edges E_i, vertices V_i, and a directed path from r to each $s \in S$. Let $w(T_i)$ be the total weight of all T_i edges.

For convenience, denote $T_i^l = (V_i^l, E_i^l, r, S)$ as the *ith* directed tree with $l \in E_i^l$. For each link l in each directed tree i, T_i^l is a backup tree $\hat{T}_i^l = (\hat{V}_i^l, \hat{E}_i^l, r, S)$, a directed tree with root r, a directed path from r to each $s \in S$ such that $l \notin \hat{E}_i^l$. We refer to T_i^l and $\hat{T}_i^l$ as a *primary tree* and *backup tree*, respectively. In Figure 4.1(b) the two backup trees – one for T_b and the other for T_c – both route around the failed link, (g, l), and have a directed path from its root (b and c) to their data sinks ($\{p, q, r, s\}$ and $\{r, s, t, u, v\}$).

Corresponding to each primary tree, $T_i = (V_i, E_i, r, S)$, is a multicast flow $f_i = (r, S)$ with source, r, and data sinks $S = \{d_1, d_2, ...d_k\}$. Each $d_i \in S$ has an end-to-end per-packet delay requirements and loss rate requirement (specified as the maximum tolerable loss rate of each $e \in E_i$). Let F be the set of all multicast flows in G.

Lastly, we make the following simplifying assumptions:

- Before any link fails, we assume that all packets (of PMU data) are correctly delivered such that each data sink's per-packet delay and loss requirements are satisfied.
- All sinks have the same same per-packet delay and loss rate requirements.
- We consider the case where multiple links fail over the lifetime of the network but assume that only a *single link fails at-a-time*.

4.2.4 OpenFlow

Our algorithms are built using OpenFlow abstractions and features. Here we provide a brief overview of OpenFlow, with a particular emphasis on the features used by our algorithms.

OpenFlow is an open standard that cleanly separates the control and data planes, and provides a programmable (and possibly centralized) control framework [58]. All OpenFlow algorithms and protocols are managed by a (logically) centralized controller, while network switches (as their only task) forward packets according to the local forwarding rules installed by the controller at that switch.

OpenFlow exposes the flow tables of its switches, allowing the controller to add, remove, and delete flow table entries, which determine how switches forward, copy, or drop packets associated with a controller-managed flow. We will use the terms "flow table entry" and "forwarding rule" interchangeably.

OpenFlow switches follow a "match plus action" paradigm [58], in which each switch *matches* an incoming packet based on its header fields to a flow table table entry. *Actions* (e.g., forward packet, drop packet, copy packet, or modify packet header fields) are then applied to the packet as encoded in the flow table entry instructions. Switches maintains statistics for each flow table entry (e.g., packet counter, number of bytes received, time the flow was installed) that can can be queried by the controller. These statistics are key to our algorithm for detecting packet loss (Section 4.3.1).

Several of our algorithms use OpenFlow to modify packet headers to customize forwarding and other actions in parts of the network. We write identifiers in unused packet header fields to customize the set of actions applied to packets carrying specific identifiers. We refer to these identifiers as *tags*. Tags are an abstraction we use to measure packet loss rates (PCOUNT in Section 4.3.1), pre-install backup tree flow table entries (PROACTIVE in Section 4.3.3), and consolidate flow table entries that have common forwarding state (MERGER in Section 4.3.5).

Measurement studies from the literature [21, 70] have identified significant hardware limitations in OpenFlow switches. OpenFlow switches can only support a limited number of flow table entries because they rely on expensive TCAM memory to perform wildcard matching. For example, the HP5406zl switch supports approxi-

mately 1500 OpenFlow rules [21] and the Pronto 3290 switch can handle 1919 flow table entries [29]. Another major limitation is control plane bandwidth: OpenFlow switches have been found to have four orders of magnitude less control plane bandwidth than data plane forwarding bandwidth [21]. This limited bandwidth, along with their slow control/management CPUs, limits the rate in which flow table entries can be installed. Our emulation study (Section 4.5) shows the tangible effects these hardware limitations have on our algorithms (especially PCOUNT).

4.2.5 Multicast Implementation

In keeping with its role as a general framework that provides primitives for programmable networks, OpenFlow does not explicitly provide an implementation for multicast. Thus, we design our own multicast implementation called BASIC. BASIC assigns a multicast IP address to each multicast group and uses this address to setup the flow tables at the multicast tree switches. [2]

After the controller computes a multicast tree (described in Section 4.3.2), $T_i = (V_i, E_i, r, S)$, BASIC installs a flow table entry at each switch in V_i. The flow table entry matches packets using the group's multicast address (all other field are left as wildcards) and forwards a copy of each packet out the ports corresponding to the switch's outgoing links in E_i. If a switch in E_i is adjacent to a downstream host, h_j, in the multicast group, then the flow table entry rewrites the destination layer 2 and 3 addresses of the packet copy sent to h_j to h_j's layer 2 and 3 addresses. [3]

---

[2]Because multicast group membership is static for power grid applications (Section 4.2.1, we simply determine the members of each multicast group by reading their static assignment from a text file. Note that if dynamic group membership were to be required, we could replace this static policy using a protocol like IGMP.

[3]Our initial plan was to use the group table abstraction described in the OpenFlow 1.1 specification [66] to implement multicast but, unfortunately, as of the writing of this chapter, this feature is not yet supported by the POX controller [57] used to implement our algorithms and the Mininet emulator [50] used in our emulations.

4.3 Algorithms

We propose a set of algorithms, collectively referred to as APPLESEED, that make multicast trees robust to link failures. [4] APPLESEED runs at the OpenFlow controller with the goal of minimizing packet loss associated with link failures while ensuring that end-to-end delay requirements are satisfied. APPLESEED divides into three parts:

1. **Pcount algorithm: monitor and quickly detect link failures** when and where they occur inside the network (Section 4.3.1).

2. **Precompute backup trees** that are amenable to fast installation. In Section 4.3.2 we formulate a new problem, MULTICAST RECYCLING, that aims to compute backup trees that reuse primary tree edges, prove MULTICAST RECYCLING is at least NP-hard, and provide an approximation algorithm for MULTICAST RECYCLING called BUNCHY.

3. **Fast install of pre-computed backup trees** by reusing existing forwarding rules installed in the network, sharing forwarding rules among backup trees with common links, and in some cases pre-installing forwarding rules before link failures occur (Section 4.3.3). The backup trees are computed using BUNCHY from part (2). PCOUNT, from part (1), triggers the installation of a set of backup trees.

4.3.1 Link Failure Detection Using OpenFlow

We present PCOUNT, an algorithm that uses OpenFlow to detect link failures inside the network. In-network detection is used to reduce the time between when packet loss occurs and when it is detected. Fast packet loss detection is crucial to the critical PMU applications that we target in this work, as they are particularly

[4] The name APPLESEED is inspired by Johnny Appleseed, the famous American pioneer and conservationist known for planting apple nurseries and caring for its trees.

sensitive to packet loss. Most previous work [5, 16, 30] focuses on measuring end-to-end packet loss, resulting in slower detection times.

PCOUNT considers a link as failed when the rate of packet loss exceeds a threshold, given as input. For simplicity, our description of PCOUNT assumes it measures packet loss over a single link, (u, d). At the end of the section we comment on how PCOUNT easily generalizes to detect packet loss between multiple switches and non-adjacent switches.

Although we present PCOUNT as an algorithm that detects packet loss of a single network link, the use of "link" should not be interpreted literally. Rather, each "link" represents a path where the two endpoints are OpenFlow switches that can be connected with a multi-hop path containing several (non-OpenFlow) switches and routers. In practice, we expect that OpenFlow switches will coexist with non-OpenFlow routers and switches, making this broader definition more compelling. From our presentation below it will be clear that strictly defining a link as a single physical link connecting two switches is not necessary for PCOUNT to work correctly.

Pcount Algorithm Details. PCOUNT estimates packet loss over a sampling window of length w. For each w, PCOUNT estimates packet loss along (u, d) by measuring the aggregate loss rate experienced by flows $M = \{f_1, f_2, ..., f_k\}$ across link (u, d), where M is given as input, using the following steps:

1. **Install rules (downstream) to count all tagged f_i packets received at d.** PCOUNT does so by installing a new flow table entry for each f_i at d, that matches packets using the identifier (i.e., the tag) applied at u in step (2). As noted in Section 4.2.4, for each flow table entry, OpenFlow automatically updates the packet counter each time a packet matches the flow table entry.

2. **Tag (upstream) all packets from each $f_i \in M$.** Suppose u uses flow table entry e_i to match and forward flow f_i packets. First, PCOUNT generates a unique identifier (tag). Then, for each f_i, PCOUNT creates a new flow table

entry, e'_i, that is an exact copy of e_i except that e'_i embeds a the tag in the packet's `dl_vlan` field. e'_i is installed with a higher OpenFlow priority than e_i, ensuring that all flow f_i packets are tagged.

3. **After w time units, turn tagging off at** u by installing a copy of each f_i's original flow table entry, e_i, but with a higher priority than e'_i.

4. **Query u and d for packet counts** in order to compute the packet loss. Each tagging rule is queried individually at u, while a single aggregate query (matching flows based on their the `vlan_id` field) is issued at d to retrieve the packet counts of total packet count of all $f_i \in M$. [5] Before querying d, PCOUNT waits time proportional to half the average RTT from u to d, starting from the time step (3) completes, to ensure all in-transit packets are considered at d.

5. **Signal an alarm** if the estimated loss rate exceeds the input threshold.

6. **Delete tagging and counting flow table entries** created in steps (1) and (2).

Consider the example in Figure 4.1 and assume PCOUNT monitors packet loss of both multicast flows traversing (g, l); f_b for primary tree T_b (blue) and f_c for primary tree T_c (green). First, PCOUNT selects a unique `dl_vlan` value (the identifier) and installs two flow table entries at l, one for f_b and the other f_c. These flow table entries match packets based on the packet's multicast address and `dl_vlan` value. Next, a flow table entry for f_b and f_c is installed upstream at g that writes the `dl_vlan` identifier in each packet sent along the outgoing link to l. After w seconds, tagging is turned off at g and the flow statistics are read from g and l. Two individual flow

[5] We are unable to issue an aggregate query at u because OpenFlow does not support query predicates specified over flow table entry actions. In our case, it would be convenient if we could to specify an aggregate query of the form "return statistics of all flow table entries that write identifier x in the `dl_vlan` field".

statistic queries are sent to g, while a single aggregate query at l gathers packet counts of the two flow table entries installed in the first step. Lastly, the packet counts are used to compute the loss rate and, if necessary, PCOUNT raises an alarm if the measured loss rate during window w exceeds the given threshold. If not, a new PCOUNT session is initiated, repeating the above steps.

Tagging ensures that for monitored flow PCOUNT accounts for all packets dropped during w. However, in some cases PCOUNT may be configured to monitor a subset of flows traversing a monitored link because doing so can reduce the time required to compute packet loss, since $k + 1$ statistic queries are required when monitoring k flows. For example, in Figure 4.1 PCOUNT may only monitor f_b's packet loss along (g, l). As a result, only a single read statistics request needs to be sent to g rather than two statistic queries if f_c were also monitored. However, monitoring a subset of flows means that PCOUNT does not account for packet loss of unmonitored flows. In Section 4.5.1 we use emulations to explore how adjusting the number of monitored flows affects the speed and accuracy of packet loss estimates.

Although PCOUNT sends instructions simultaneously to start tagging each M flow (step 2) and, likewise, sends all k messages in parallel to stop tagging (step 3), in practice these actions are unlikely to be executed at the same time. The implication is that across each $f_i \in M$ the start and stop time of w is not perfectly synchronous. This does not affect the accuracy of PCOUNT packet loss measurements, provided that all tagged packets that will eventually reach the downstream node do so before that node is queried.

Pcount Extensions. No changes are required for PCOUNT to monitor packet loss between non-adjacent switches. Consider the case where PCOUNT measures packet loss between two non-adjacent switches a and b. The PCOUNT actions at a and b are the same as described above, while the forwarding at any switches along a path

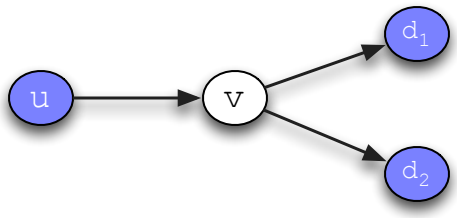


Figure 4.2. Example topology used to explain how PCOUNT can be used to monitor packet loss between multiple non-adjacent switches.

between a to b disregards (as they already do) any tag applied at a or b. In this scenario, PCOUNT measures packet loss of a path rather than that of a single link.

PCOUNT can also be used to monitor packet loss between multiple (possibly) non-adjacent switches. Consider the example topology in Figure 4.2, where PCOUNT measures packet loss between u and downstream nodes d_1 and d_2. For simplicity, we assume a single flow multicasts packets from u to d_1 and d_2. PCOUNT installs a rule to tag packets at u, leaves v is unchanged, and installs a rule at d_1 and d_2 to count packets tagged at u. Then, PCOUNT (as its only modification) queries u, d_1, and d_2 for their packet counts.

Notice that by comparing packet counts between u, d_1, and d_2, packet loss of links (u, v), (v, d_1), and (v, d_2) can all be estimated using network tomography techniques [15]. For example, if u and d_1 have the same packet counts but d_2 counts fewer packets than u, we can infer packet loss incurs along (v, d_2). This approach provides the same coverage (scope of packet loss measurements) as an alternative approach that creates three separate PCOUNT sessions between (u, v), (v, d_1), and (v, d_2), but does do so using fewer measurement points. We later find in our emulations (Section 4.5.1) that monitoring a large number of flows using a single link incurs high processing time at network switches, suggesting that the savings projected here of running a single PCOUNT session between multiple switches can provide significant savings.

4.3.2 Computing Backup Trees

APPLESEED pre-computes backup trees to install after PCOUNT detects a link failure. Here we present a new problem, MULTICAST RECYCLING, and an approximate solution to MULTICAST RECYCLING, called BUNCHY, that aim to facilitate fast recovery by computing backup trees that maximize the number of edges common between each backup tree and its primary tree. This reuse of primary tree edges speeds recovery from link failures because, in SDN, this reduces the number of new flow table entries that need to be installed in network routers in response to a link failure.

APPLESEED uses BUNCHY as a part of system initialization, where the set of backup trees are computed for each network link, l; APPLESEED computes a single backup tree for each primary tree using l. Additionally, BUNCHY is used after a set of backup trees, $\hat{T}^l$, are installed in response to a link failure. For each newly installed tree $\hat{T}^l_i \in \hat{T}^l$, APPLESEED computes a backup tree for each link in $\hat{T}^l_i$.

4.3.2.1 Multicast Recycling Problem

The goal of the MULTICAST RECYCLING problem is to compute backup trees that maximize reuse of primary tree edges. Recycling primary tree edges allows the SDN controller, when generating the forwarding rules for multicasting packets using the backup tree, to use primary tree rules already installed in the network rather than install new ones. This speeds recovery in cases where backup trees are installed *after* a link failure is detected and reduces the number of flow table entries pre-installed at switches (control state) when backup trees are installed *before* a link failure occurs. Reducing control state is especially important with OpenFlow because OpenFlow switches can only store a limited number of flow table entries (see Section 4.2.4). [6]

[6]Following from our assumption that a single link fails at-a-time, MULTICAST RECYCLING assumes that all other links besides the failed one, l, satisfy packet loss requirements.

For the primary tree $T_i^l = (V_i^l, E_i^l, r, S)$ and its backup $\hat{T}_i^l = (\hat{V}_i^l, \hat{E}_i^l, r, S)$, we define a binary variable c_v^l for all $v \in \hat{V}_i^l$. If v has exactly the same predecessors (outgoing edges) in T_i^l and $\hat{T}_i^l$, then c_v^l takes value 0. Otherwise, $c_v^l = 1$. For the T_i^l,$\hat{T}_i^l$ pair define:

$$C_i^l = \sum_{\forall v \in \hat{V}_i^l} c_v^l \tag{4.1}$$

For our purposes, C_i^l is the number of new rules (i.e., non-recycled primary tree rules) needed to install $\hat{T}_i^l$. Note that the primary tree rules not recycled by $\hat{T}_i^l$ should be deleted after l fails and $\hat{T}_i^l$ is installed, especially considering the limited size of OpenFlow switch flow tables (Section 4.2.4). As we describe in Section 4.3.4, these rules can be garbage collected in the background because stale primary tree forwarding rules do not affect how packets are (correctly) forwarded by $\hat{T}_i^l$. For this reason, Multicast Recycling aims to minimize the number of new rules needed to install a backup tree, rather than the number of primary tree rules that must be garbage collected after a link failure.

Consider the example in Figure 4.1 where (g, l) fails. The green backup tree, $\hat{T}_b$, shown in Figure 4.1(b), has $C_b = 2$ because a new forwarding rule is required at b, and f to account for the new outgoing links at each node. $\hat{T}_c$, in blue, has only one link, (m, l) not in $\hat{T}_c$'s primary tree. As a result, $C_c = 3$.

Our Multicast Recycling problem definition below references a modified version of the Steiner tree problem, called the Steiner-Arborescence problem [17]. As input, Steiner-Arborescence is given a directed graph $G = (V, A)$, a root vertex r, and a set of terminals, S. An arborescence is defined as a tree rooted at r that has directed edges spanning S. Steiner-Arborescence aims to find a minimum cost arborescence, called a Steiner arborescence or directed Steiner tree. We denote

$SA_i(G) = (V, E, r, S)$ as the Steiner arborescence computed over directed graph, G, rooted at r, and spanning S such that $r, S \in T_i$.

We formulate the MULTICAST RECYCLING problem as follows:

- Input: (G, T^l, l, α) where $G = (V, E)$ is a directed graph, $T^l = \{T_1^l, T_2^l, \ldots T_k^l\}$ where each $T_i^l \in T^l$ is a primary tree that uses l, $l \in E$, and $\alpha \geq 1$.

- Output: A backup tree for each primary tree using l. This set of backup trees, $\hat{T}^l = \{\hat{T}_1^l, \hat{T}_2^l, \ldots, \hat{T}_k^l\}$:

$$\begin{aligned} &\text{minimize} \quad \sum_{1 \leq i \leq k} C_i^l \\ &\text{subject to} \quad w(\hat{T}_i^l) \leq \alpha \cdot w(SA_i(G')),\ \forall \hat{T}_i^l \in \hat{T}^l \end{aligned} \tag{4.2}$$

  where $G' = (V', E')$ such that $E' = E - \{l\}$ and $w(\hat{T}_i^l)$ is the sum of $\hat{T}_i^l$'s link weights. [7]

The objective function maximizes the reuse of primary tree edges, while α bounds how large the backup tree can grow as consequence of minimizing C_i^l. When applied to our problem scenario this formulation reduces the number of installation rules by reusing rules already installed in the network, under the constraint that the backup tree does not become too large to meet the end-to-end latency requirements. By defining G' as a copy of G with the failed link removed from G, we are assuming that all links in G besides l are operational. For our purposes, this amounts to assuming that all non-l links have packet loss rates less than their threshold.

Notice that we have defined C_i^l in Equation 4.1 on a per-backup tree basis where for backup tree $\hat{T}_i^l$, C_i^l is a relationship defined strictly between $\hat{T}_i^l$ and its primary tree T_i^l (there are no constraints specified across any other primary or backup tree). As

[7] We assume $\hat{T}_i^l$, satisfies all per-packet delay and loss requirements if $l \notin \hat{T}_i^l$ and $w(\hat{T}_i^l) \leq \alpha \cdot w(SA_i(G'))$

a result, the globally optimal solution for MULTICAST RECYCLING (i.e., the optimal set of backup trees for a single link) can be found by computing the optimal backup for each primary tree in isolation and then taking the union of these solution We shall revisit this important property when describing our approximation algorithm for MULTICAST RECYCLING.

Theorem 4.1. MULTICAST RECYCLING *is at least NP-hard.*

Proof. The details of our proof can be found in Appendix C.1. This proof shows that MULTICAST RECYCLING is NP-hard even when considering just a single backup tree. The proof demonstrates that in some cases an optimal solution to MULTICAST RECYCLING requires a solution to STEINER-ARBORESCENCE, a problem known to be NP-hard. This proves MULTICAST RECYCLING is NP-hard when considering a single backup tree and therefore the general MULTICAST RECYCLING problem for k backup trees must at least be NP-hard. □

4.3.2.2 Bunchy Approximation Algorithm

BUNCHY is a simple approximation algorithm for MULTICAST RECYCLING that manipulates link weights to encourage each backup tree to reuse primary tree edges. For each link l, BUNCHY separately computes a backup tree for each primary tree using l and then returns the union of these computed trees.

BUNCHY leverages the $\sqrt{s}$ STEINER-ARBORESCENCE approximation, where s is the number of terminal nodes, from Charikar et al. [17]. Their approximation algorithm computes bunches, where a *bunch* is a subgraph formed by taking the shortest path from the root to an intermediate vertex, i, and the union of shortest paths from i to the terminal nodes. The algorithm produces the bunch with best *density* – density is the average cost of connecting a terminal node with the root – as its approximation. The lowest density bunch can easily be computed in polynomial time: a brute-force

approach that tries all possible nodes as the intermediate vertex yields an $O(ns^2 \log s)$ time algorithm.

Given (G, T^l, l, α), for each $T_i^l \in T^l$ BUNCHY uses the following two-step procedure to compute $\hat{T}_i^l$:

1. Make a copy of G called $G' = (V', E')$ and remove l from E'. Set the link weight of each $e \in T_i^l$ to 0 and the link weight of $e \notin T_i^l$ to 1.

2. Run the STEINER-ARBORESCENCE approximation, using the brute-force approach described above, over G' and set $\hat{T}_i^l$ to be the result. If $\hat{T}_i^l$ satisfies the Equation 4.2 constraint, return $\hat{T}_i^l$ as the solution. Otherwise, return False.

Setting the primary tree link weights to 0 in Step (1) allows the STEINER-ARBORESCENCE approximation algorithm to use any primary tree edge without penalty (i.e., adding cost to the backup tree) and so encourages reusing primary tree edges. If BUNCHY returns False in Step (2) either α must be made larger or a new multicast tree should be computed from scratch that satisfies the tree-size constraint.

In Figure 4.1, BUNCHY uses f as the the intermediate node for $\hat{T}_b$, yielding density of 2 (the cost of connecting terminals p,q,r, and s to the root is 2). BUNCHY selects f as the intermediate node by iterating over all nodes and remembering the node with the smallest density. g or l could have been used as the intermediate node because both, like f, have density of 2 (f is selected arbitrarily using a tiebreaker). The bunch for $\hat{T}_c$ is formed using m as the intermediate node with density 0.4: the cost of connecting r and s to the root is 1 and t,u, and v connect with the root at 0 cost.

4.3.3 Installing Backup Trees

We are now ready to describe the last part of APPLESEED, installing backup trees. Installing a backup tree is a two-step process. First, the flow table entries that forward packets along the backup tree are generated. Second, the controller signals the necessary switches to install the generated forwarding rules. Here we intro-

duce two such installation algorithms, PROACTIVE and REACTIVE. Both algorithms compute forwarding rules for a single backup tree at-a-time and so our description of each algorithm (with some abuse of notation) refers to a generic primary tree, $T^l = (V^l, E^l, r, S)$, and its backup tree for $l \in E^l$, $\hat{T}^l = (\hat{V}^l, \hat{E}^l, r, S)$.

Reactive Algorithm. REACTIVE first determines which nodes require a new forwarding rule. In cases where $\hat{T}^l$ and T^l use exactly the same outgoing links of a common node, u, we say $\hat{T}^l$ can "reuse" T^l's forwarding rule at u; since T^l's forwarding rule is already installed at u, no new forwarding rule (for $\hat{T}^l$) needs to be installed. Forwarding rules are only required at any $v \in \hat{V}^l \setminus V^l$ and at each $v \in V^l \cap \hat{V}^l$ with different outgoing links in $\hat{T}^l$ and T^l. We refer to this set of nodes as B^l.

Consider T_b and $\hat{T}_b$ in the Figure 4.1 example. Because T_b and $\hat{T}_b$ share the same outgoing links at l and k, $\hat{T}_b$ can reuse T_b's flow table entry at each of these nodes, whereas new forwarding rules are required at b and f.

REACTIVE then pre-computes a *basic* flow table entry for each $b \in B^l$. Like the flow table entries BASIC computes (see Section 4.2.5), a basic flow table entry, for a multicast tree T_i and $u \in V_i$, matches packets using T_i's multicast address and has instructions to forward matching packets out the correct ports at u. Lastly, when l fails, the REACTIVE signals each $b \in B^l$ to install the pre-computed basic flow rule.

Proactive Algorithm. PROACTIVE computes and installs backup tree flow table entries *before* a primary tree link, l, fails. After l fails, PROACTIVE signals the backup tree root to install a forwarding rule that activates the backup tree. We use the term "activate" to indicate that packets are multicasted using the backup tree rather than the primary tree. Note that the PROACTIVE algorithm can respond quickly to link failures, as only a single new flow table entry needs to be installed at the backup tree root.

PROACTIVE cannot, without modifications, pre-install basic flow table entries at all nodes because incorrect forwarding would result. Doing so at a node, d, common

to the primary and backup tree, where the backup and primary tree have different outgoing links, would either result in packets erroneously forwarded at d using the backup tree before a link failure occurs or incorrectly forwarding packets using the primary tree after the link failure. We say that $d \in D^l$, where D^l contains each node with one or more outgoing links in T^l and one or more outgoing links in $\hat{T}^l \setminus T^l$. Revisiting $\hat{T}_c$ from Figure 4.1 example, $g, m \in D^l$ and so installing a forwarding rule at these two switches before (g, l) fails would be problematic for the reasons just described.

To circumvent this issue, PROACTIVE assigns a unique *backup tree id*, denoted `bid`, to each backup tree. For each $d \in D^l$, the flow table entry matches and forwards packets using the `bid` value written in the `dl_src` field. When the backup tree $\hat{T}^l$ is activated, PROACTIVE writes the `bid` in the `dl_src` packet header field, indicating that these packets should be disseminated by $\hat{T}^l$ rather than T^l. In more detail, PROACTIVE preinstalls and activates $\hat{T}^l$ using the following steps, where we assume $\hat{T}^l$ has `bid=AA`:

1. At each $d \in D^l$, PROACTIVE pre-installs a flow table entry matching packets using $\hat{T}^l$'s multicast address, `dl_src = AA`, and has wildcards for all other match fields. PROACTIVE preinstalls a basic flow table entry at each $b \in B^l \setminus D^l$ that matches packets using $\hat{T}^l$'s multicast address and has wildcards for all other match fields (including `dl_src`).

2. When it is detected that l fails, PROACTIVE installs a rule at the $\hat{T}^l$ root node that writes `AA` in the `dl_src` header field of each $\hat{T}^l$ packet.

For $\hat{T}_c$ in Figure 4.1, PROACTIVE pre-installs a forwarding rule at g and m that matches packets using $\hat{T}_c$'s `bid`, `CC`, and T_c's multicast address. After (g, l) fails, PROACTIVE signals c to write `CC` in the `dl_src` header field of each $\hat{T}_c$ packet. As a

result, packets at g are correctly forwarded to m and, similarity, m correctly forwards packets to l, n, and t.

Comparing Proactive and Reactive. Since PROACTIVE must signal only a single node, as opposed to multiple nodes with REACTIVE, to install a backup tree, PROACTIVE is fast. However, PROACTIVE's speed comes at the cost of storing a potentially large number of flow table entries at the switches, especially since APPLESEED computes, for each primary tree, a backup tree for each primary tree link. REACTIVE, on the other hand, only installs backup tree flow table entries after a link failure is detected. These trade-offs are studied in Section 4.5 using emulations.

4.3.4 Garbage Collection

After a link fails, primary tree forwarding rules may become stale. APPLESEED's garbage collection routine identifies and deletes these stale flow table entries. Because garbage collection is not needed for correct data dissemination, garbage collection is run when necessary to free switch flow table space.

Garbage collection is straightforward, but more involved than simply deleting all flow table entries of each primary tree, T^l, using l. Doing so would be problematic because a backup tree may be reusing one of these flow table entries. To address this, APPLESEED maintains a dictionary, `rule_map`. For each node, v, `rule_map` records each flow table entry installed at v and the multicast trees using the flow table entry. When l fails, the garbage collection routine determines the set of stale forwarding rules for each $T_i^l \in T^l$ by consulting `rule_map`. A stale rule exists at each $v \in V_i \setminus \hat{V}_i^l$ (i.e., nodes unique to the primary tree) and each $d \in D_i^l$ (nodes where the backup tree diverges from the primary tree). Finally, each stale forwarding rule is either explicitly removed (if using a hard-state signaling protocol [42]) or APPLESEED allows the forwarding rule to timeout (if using a soft-state signaling algorithm [19]).

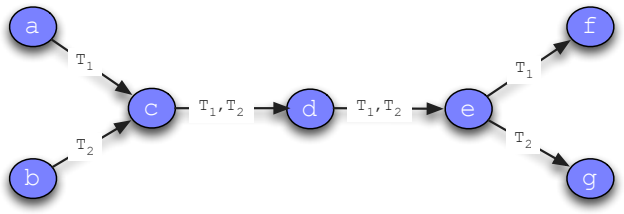


Figure 4.3. Example showing a subtree of two multicast trees, T_1 and T_2. The edges used by each multicast tree are marked.

4.3.5 Optimized Multicast Implementation

As an optimization to the BASIC multicast implementation, described in Section 4.2.4, we present the MERGER algorithm. Given a set of directed trees, MERGER produces a near-minimum set of OpenFlow forwarding rules by consolidating flow table entries at each node where multiple trees use the same set of out-links. MERGER reduces the control state (i.e., number of forwarding rules) necessary to multicast packets and, when applied to installing backup trees, can yield faster recovery since fewer control messages are needed to activate backup trees.

In the next section (4.3.5.1), we motivate the need for MERGER by demonstrating several inefficiencies in BASIC. Next, Section 4.3.5.2 presents a simplified version of MERGER that considers only primary trees. Then, we extend MERGER in Section 4.3.5.3 to account for backup trees. Section 4.3.5.4 concludes the section with a discussion of how MERGER affects garbage collection and PCOUNT, along with informal commentary on its optimality.

4.3.5.1 Motivation: Basic Algorithm Inefficiencies

The BASIC multicast implementation creates a flow table entry at each node of a multicast tree that matches incoming packets using the tree's multicast address. As a result, a switch, v, may have multiple flow table entries executing the same forwarding actions. This occurs when multicast trees share the same outgoing links

at v. [8] These inefficiencies are the motivation for developing the MERGER algorithm, which replaces the set of flow table entries BASIC would create at v with a single flow table entry. To do so, MERGER writes a common identifier, or tag, in packet headers at the node immediately upstream from v. Using this tag, MERGER creates a *single* rule at v to match and forward packets of *all* trees with the same outgoing links at v.

Consider the simple example shown in Figure 4.3 with two multicast trees T_1 and T_2. The directed links used by each tree are marked. BASIC creates a flow table entry for T_1 at a, c, d, e, and f and a flow table entry at b, c, d, e, and g for T_2. Because T_1 and T_2 both use the same outgoing links at c and d, only a single forwarding rule is needed at each node. In the next two sections we describe how MERGER finds duplicate forwarding actions and creates forwarding rules shared by multiple multicast trees.

4.3.5.2 Merger Algorithm for Primary Trees

MERGER consolidates flow table entries at each node, v, where multiple primary trees share the same outgoing links. Upstream from v, MERGER writes an identifier, or *tag*, in packet headers and uses this tag to match packets at v using a single rule shared by each of these primary trees. The tag is removed downstream from v where the trees diverge.

A tag is a globally unique Ethernet address that MERGER writes in a packet header's `dl_dst` field (i.e., the Ethernet destination address). When possible, MERGER flow table entries use tags to match and forward packets, meaning that packets are matched solely on their `dl_dst` value. When the same Ethernet address is applied to the packets of more than one multicast tree, we refer to this as a *group tag*. A *single tag* is an Ethernet address used by only one tree. We use the term *tag* to generically

[8] In cases where the tree can either be a primary tree or backup tree, we refer to the tree as a multicast tree.

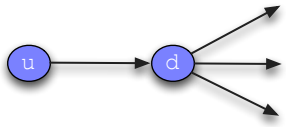


Figure 4.4. Subgraph used to describe MERGER in Section 4.3.5.2.

refer to either a group or single tag. Note that, like BASIC, MERGER has each switch adjacent to a downstream host, h_j, overwrite the destination layer 2 (including the `dl_dst` field) and 3 fields in the packet forwarded to h_j, setting these fields to h_j's layer 2 and 3 addresses. This allows MERGER to safely modify the `dl_dst` field inside the multicast tree for its tagging purposes, while ensuring successful forwarding of packets to each multicast group host.

We are now ready to describe MERGER in more detail. First, MERGER marks the edges used by each primary tree. Then, MERGER executes a breadth first search of each primary tree, T_i, starting at its root. For each link $(u, d) \in T_i$ as shown in Figure 4.4, MERGER determines the match pattern to create at d and the tagging actions to apply at u using the following steps: [9]

1. Finds the set of trees, S, using (u, d) that share the same outgoing links as T_i at d.

2. If $|S| \geq 1$, MERGER creates an action to write a group tag at u. For each $T_j \in S \cup \{T_i\}$, MERGER finds the rule at u used to forward T_j and appends an action to write a group tag to the rule's action list. Then, a single rule is created at d that matches packets using this group tag and has an initial action list forwarding packets out the appropriate ports.

[9]Because T_i is a tree, (u, d) must be its only incoming link to d. Therefore, we can determine T_i's locally optimal tagging rule at d by only considering u and d.

3. If $S = \emptyset$, MERGER looks upstream at u to determine whether to use a single tag or T_i's multicast destination address to match T_i's packets at d. When T_i is matched using either a single tag or its multicast destination address at u, MERGER creates a rule at d to match packets using a single tag and writes this single tag at u. Otherwise, MERGER creates a rule at d matching packets using T_i's multicast destination address (no action is needed at u).

In step (3), we aim to use single tags to match packets because they allow *any* backup tree to reuse T_i's rule at d by simply writing this tag at u. Whereas, if T_i's multicast address is used to match packets at d, only T_i's backup trees can reuse this forwarding rule (since T_i's multicast address is unique to its multicast group). We comment further on this design decision in the next section.

In the Figure 4.3 example, MERGER creates an action at a and b to write a group tag in the packet headers of all packets traversing (a, c) and (b, c). Then, at c and d, MERGER creates a single rule to match and forward packets based solely on this tag. With regard to the breadth-first search (BFS) described earlier, MERGER executes the following steps in its breadth-first search (BFS) of T_1 at nodes c, d, and e. At c, $S = \{T_2\}$ so MERGER finds T_1's rule at a and includes an action to write a group tag, `12`, in all packets sent out a's port to c. Then, MERGER creates a flow table entry at c that matches packets with `dl_dst = 12` and forwards packets out the port to d. The same set of actions occur when the BFS reaches d. MERGER creates a forwarding rule at e that matches packets using T_1's multicast address and forwards these packets to f.

4.3.5.3 Merger Algorithm for Backup Trees

Having discussed MERGER for the primary tree case, we are now ready to extend MERGER to generate merged forwarding rules for backup trees. MERGER aims to reuse forwarding rules of primary trees because these rules are already installed in

the network, allowing the installation algorithm (e.g., PROACTIVE or REACTIVE) to avoid installation of redundant forwarding rules. In cases where primary tree rules cannot be reused, MERGER consolidates flow table entries with other backup trees that have common forwarding behavior.

For a set of backup trees, $\hat{T}^l$, MERGER generates backup-tree forwarding rules as follows. MERGER executes two rounds of BFS, traversing all $\hat{T}^l_i \in \hat{T}^l$ in each round. In the first round, for each $\hat{T}^l_i$, MERGER finds each node where $\hat{T}^l_i$ has the same outgoing links as a primary tree. If so, $\hat{T}^l_i$ reuses the primary tree flow table entry at this node, v: MERGER writes the primary tree tag at v's parent node allowing $\hat{T}^l_i$ packets to be forwarded using the primary tree rule, and makes no changes at v.

In the second round of BFSs, MERGER consolidates flow table entries among the other backup trees for l at nodes where primary tree tag reuse was not possible. To do so, the algorithm from Section 4.3.5.2 is executed but compares $\hat{T}^l_i$'s outgoing links with the outgoing links of each $\hat{T}^l_j \neq \hat{T}^l_i$ at nodes where $\hat{T}^l_i$ and $\hat{T}^l_j$ were unable to reuse primary tree tags.

When MERGER is applied to PROACTIVE and REACTIVE (referred to as PROACTIVE+MERGER and REACTIVE+MERGER) the tag becomes the sole match criteria used by its flow table entry, with one exception. This occurs with PROACTIVE+MERGER when a `bid` is required to distinguish between a backup and primary tree, as described in Section 4.3.3. In those cases, the `bid` and `dl_dst` fields are both used as matching criteria.

4.3.5.4 Merger Discussion

We conclude our presentation of MERGER by commenting on some of the properties of the algorithm along with the implications of using MERGER on other important aspects of APPLESEED.

Benefits. In comparison with BASIC, MERGER reduces the number of forwarding rules required to install each multicast tree. As a result, MERGER is more space efficient and can yield faster backup tree installation; in Section 4.5.2 we quantitatively evaluate these gains. Recall that for each primary tree, APPLESEED pre-computes a backup tree for each of its links. In this scenario PROACTIVE+MERGER can significantly reduce the number of pre-installed rules. These savings are especially important because, as noted in Section 4.2.4, OpenFlow switches can only support a limited number of flow table entries. With REACTIVE, MERGER can yield faster recovery because fewer backup tree flow table entries translates into fewer control messages to activate backup trees.

Time Complexity. Like BASIC, MERGER's complexity is bounded by the breadth-first search (BFS) executed for each of the m multicast trees given as input. At each node, v, visited in the BFS, MERGER compares the out-links used by all other multicast trees that use v. Since there can be at most m such trees, this takes $O(m)$ time. If we let n be the number of graph nodes, each BFS takes $O(mn)$ time [10] and the total time complexity of MERGER is $O(m^2n)$.

Garbage Collection. APPLESEED's garbage collection algorithm remains unchanged from the description in Section 4.3.4. Recall that `rule_map` is a dictionary that maps the flow table entry installed at each node to the set of backup and primary trees using the flow table entry. The only difference between BASIC and MERGER garbage collection is the number of stale rules it identifies (likely fewer stale rules with MERGER) not how the stale rules are found. As with BASIC, MERGER garbage collection can find any stale forwarding rules by consulting `rule_map`.

[10] BFS has $O(|V| + |E|)$ complexity. In our case, each BFS is over a directed tree, meaning the number of edges traversed in each BFS is $O(n-1)$. Therefore, we can simplify BFS time complexity to $O(n)$.

Do No Harm. We say that MERGER is an algorithm that does "no harm" [11] because (a) MERGER never creates more flow table entries than BASIC and (b) when generating rules for backup trees, MERGER makes no modifications to flow table entries of primary trees that do not use the failed link. We informally demonstrate each of these properties below.

Regarding (a), consider an arbitrary multicast tree (primary of backup tree), T_i. MERGER creates at most one flow table entry at any $v \in T_i$. The flow table entry, e_i, either matches and forwards packets using a group tag, a single tag, or using the T_i's multicast address. Any e_i tagging actions are simply appended to e_i's action list (when MERGER visits T_i's children of v), requiring the creation of no additional flow table entries. We conclude that in the worst case, MERGER creates the same number of flow table entries as BASIC.

Now consider property (b) where we let T_j refer to a primary tree not using l. By construction, MERGER only creates flow table entries and new actions for the backup trees of link l (i.e., $\hat{T}^l$). These flow table entries have different match criteria than T_j's. If a backup tree reuses T_j's flow table entry at v, no changes are made to this flow table entry. Lastly, APPLESEED's garbage collection algorithm ensures that no T_j flow table entry is removed.

Optimality. Because MERGER makes tagging decisions locally at each node, v, based only on the multicast trees using v's outgoing links and the tags used at v's parent nodes, MERGER does not always yield the minimum set of forwarding rules. Consider again Figure 4.3 but replace T_1 with S_1 and T_2 with S_2, where S_1 and S_2 are sets of multicast trees of size k. In this scenario, MERGER writes the same group tag, denoted `12`, at a and b for each tree in S_1 and S_2. Then, at c and d MERGER installs a single rule that matches and forwards using the `12` tag. Lastly, for each

[11] This is similar in spirit to the Hippocratic Oath taken by physicians that they will "never do harm [to patients]"

$T_i \in S_1 \cup S_2$, a rule is created at e that matches packets using each T_i's multicast address.

A better solution in this example would be to use a different group tag for S_1 and S_2. In this case, let `11` be the group tag applied to each tree in S_1 at a and `22` the tag written in the packet header of each tree in S_2. These group tags can be used to create two separate rules at c, d, and e to forward S_1 packets based on `11` and S_2 packets using `22`. By using different tags for S_1 and S_2, we avoid having to create $2k$ separate rules for each tree in $S_1 \cup S_2$ at e, clearly a better solution that the one produced by MERGER.

This example suggests that an algorithm, $\mathcal{A}$, that finds the minimum number of forwarding rules for a set of multicast trees T must consider, for $S \subseteq T$ where each multicast tree $S_i \in S$ uses link (u, d), how each of S's subsets share links downstream from d. Since this requires computing the power set of S and there are an exponential number of ways S's subsets can use common links downstream from d, we conjecture that no polynomial time $\mathcal{A}$ exists.

Implications for Pcount. PCOUNT requires no changes to monitor the packet loss of flows forwarded using MERGER rules. However, we make the case here that MERGER can improve the accuracy of PCOUNT loss rate estimates and reduce the time to compute these estimates. In Section 4.5.1, we will find that our emulations bear out the qualitative argument made here.

Recall from Section 4.3.1 that with PCOUNT the number of monitored flows, k, can be tuned. Determining an appropriate value for k involves a trade-off between the accuracy of packet loss estimates and time: larger k yield more accurate packet loss estimates but at the cost of slower detection times (the time between when packet loss occurs and when it is detected). Detection times of a monitored link, (u, d), increase with larger k for two reasons. First, for each of the k flows, PCOUNT makes a copy of the flow's forwarding rule at u and d in order tag and count packets. Secondly,

PCOUNT sends k queries to u to read the state of each flow table entry generated by PCOUNT.

With MERGER, the same flow table entry, e_i, can be used by r multiple flows. In these cases, PCOUNT only needs to explicitly monitor one of these flows to measure the packet loss of all r flows. That is, PCOUNT can monitor the loss of all r flows at the cost of monitoring a single flow: the one copy of e_i PCOUNT makes at u and d ensures that packets of any of these r flows are tagged and counted and thus only a single statistic query is needed to retrieve e_i's packet count.

Consider the example in Figure 4.1(a) and suppose that PCOUNT monitors link (g, l). Two multicast flows – f_b for primary tree T_b and f_c for primary tree T_c – traverse (g, l). BASIC creates a separate forwarding rule for f_b and f_c at g while MERGER generates a single forwarding rule at g used by both f_b and f_c. As a result, with MERGER, PCOUNT can track the packet loss of f_b and f_c by querying just the single shared MERGER forwarding rule at g (rather than interact with two separate BASIC forwarding rules). These savings are quantified via emulation in Section 4.5.1.

4.4 Related Work

Related work divides into the following categories: smart grid communication networks (Section 4.4.1), algorithms for detecting packet loss (Section 4.4.2), and link failure recovery algorithms (Section 4.4.3).

4.4.1 Smart Grid Communication Networks

The Gridstat project, started in 1999, was one of the first research projects to consider smart grid communication abstractions.[12] Our work has benefited from their detailed requirements specification [8]. Gridstat proposes a publish-subscribe

[12] `http://gridstat.net/`

architecture for PMU data dissemination. By design, subscription criteria are simple in order to ensure fast forwarding of PMU data.

Gridstat is separated into a data plane and a management plane. The management plane keeps track of subscriptions, monitors the quality of service provided by the data plane, and computes paths from subscribers to publishers. To increase reliability, each Gridstat publisher sends data over multiple paths to each subscriber. Each of these paths is a part of a different (edge-disjoint) multicast tree. Meanwhile, the data plane simply forwards data according to the paths and subscription criteria maintained by the management plane.

In North America, all PMU deployments are overseen by the North American SynchroPhasor Initiative (NASPI) [13]. NASPI has proposed and started (as of December 2012) to build the communication network used to deliver PMU data, called NASPInet. The interested reader can consult [13] for more details.

Although Gridstat [8] and NASPI [13] have similarities with Appleseed, these projects have a different focus than ours. Gridstat and NASPI are overlay networks built on top of existing network protocols (e.g., IP, MPLS), while the emphasis of our work is in making network protocols more robust to handle PMU application requirements.

Hopkinson et al [39] propose a Smart Grid communication architecture that handles heterogeneous traffic: traffic with strict timing requirements (e.g., protection systems), periodic traffic with greater tolerance for delay, and aperiodic traffic. They advocate a multi-tiered data dissemination that uses a technology such as MPLS to make hard bandwidth reservations for critical applications, the use of Gridstat to handle predictable traffic with less strict delivery requirements, and finally the use of Astrolab (which uses a gossip protocol) to manage aperiodic traffic sent over the remaining available bandwidth. They advocate hard bandwidth reservations – modeled as a multi-commodity flow problem – for critical Smart Grid applications.

4.4.2 Detecting Packet Loss

Most previous work for detecting packet loss [5, 16] is based on end-to-end measurements. These approaches require too many measurements (and therefore too much time between when the loss occurs and when it is detected) to accurately measure packet loss in our problem setting. For example, the loss model proposed by Càceres et al. [16] requires approximately 2000 end-to-end probe messages for packet loss estimates to converge on the true underlying packet loss rate. In our problem, where packet loss must be detected at small time scales, these 2000 probe messages would either need to be sent at a high rate to detect packet loss at small time scales (e.g., to detect packet loss at 1 second intervals, probe messages would need to be sent at a rate 30 times higher than PMU sending rates of 60 msgs/sec) [13] or require a prohibitively large window of time if probes were sent at a rate proportional to PMU measurement rates (e.g., over 30 seconds is required to send 2000 probes at a rate of 60 msgs/sec). PCOUNT provides faster and more accurate loss estimates of individual links than these approaches based on end-to-end measurements since it directly measures actual traffic *inside* the network.

Friedl et al. [30] propose a *passive* measurement algorithm that directly measures actual network traffic to determine application-level packet loss rates. Unfortunately, their approach can only measure packet loss after a flow is expired. Since PMU application flows are long lasting (running continuously for days, weeks, and even years), this makes their algorithm unsuitable for our purposes. PCOUNT has no such restriction that packet loss can only be measured over expired flows.

A standard Internet-based approach to passive monitoring of packet loss is to query the native Management Information Base (MIB) counters stored at each router using the Simple Network Management Protocol (SNMP) [10]. This approach is well-

[13] The would likely lead to inaccurate results [10].

suited for course-grained packet loss measurements but not for the fine-grained packet loss detection required by critical PMU applications. Specifically, this approach cannot provide synchronized reads of packet counts across routers/switches, resulting inaccuracies too large for the applications we target.

Existing network protocols, such as BGP, send periodic keep-alive messages between routers to ensure network links are operational. Detecting down links is a different (but complementary) problem than the one we consider, estimating packet loss rates over small time scales.

PCOUNT's approach for ensuring consistent reads of packet counters bears strong resemblance to the idea of *per-packet consistency* introduced by Reitblatt et al. [68]. Per-packet consistency ensures that when a network of switches changes from an old policy to a new one, that each packet is guaranteed to be handled exclusively by one policy, rather than some combination of the two policies. In our case, we use per-packet consistency to ensure that when PCOUNT reads packet counters between an upstream node, u, and downstream node, d, that exactly the same set of packets are considered (excluding, of course, packets that are dropped at u or dropped along the path from u to d.)

4.4.3 Recovery from Link Failures

MPLS is commonly used to extend IP routing with traffic engineering capabilities and fast failure recovery [69]. MPLS pre-computes backup paths for link and router (node) failures and stores these paths at the node immediately upstream from the failed component. This allows for fast, localized recovery: the node detecting a link or node failure immediately reroutes packets along its pre-computed backup path. Unfortunately, MPLS cannot be directly applied to our multicast problem scenario because MPLS addresses unicast communication (a backup unicast path applied to a multicast tree may not result in a valid tree). However, PROACTIVE is in-part inspired

by MPLS fast reroute's approach of storing pre-computed backup paths inside the network.

Multicast Recycling uses different optimization criteria to compute backup trees than prior work [20, 27, 46, 51, 52, 56, 59, 67, 76]. Past approaches use local/myopic optimization criteria (i.e., constraints specified over a *single* multicast tree), while we consider global (network-wide) criteria (i.e., constraints specified across *multiple* multicast trees). [14] In addition, none of these approaches seek to reuse already installed forwarding rules or minimize control signaling, as Multicast Recycling does. Instead, backup paths or trees are computed with one of the following objective functions: maximize node (link) disjointedness with the primary path [20, 27, 56, 59], minimize bandwidth used [76], minimize backup bandwidth reservations [46, 51, 52], minimize the number of group members that become disconnected after a link failure [67], or minimize path length [73].

Because these backup paths/trees are computed using distributed algorithms, the mechanisms to install these backup trees must navigate an inherent trade-off between high overhead (e.g., message complexity, storing large number of backup paths at routers) and fast recovery (i.e., the time between when the failure is detected and when the multicast tree is repaired should be small) [20]. Algorithms that compute and install backup paths on-demand (after a component failure is detected) scale well since forwarding state for backup paths is only installed after a failure is detected. However, on-demand solutions can be have slow convergence time.

Algorithms that pre-compute and pre-install backup path/trees are fast but scale poorly as significant forwarding state must be stored and maintained at routers. Scalability is particularly challenging because previous work [20, 27, 46, 51, 52, 56,

[14] Li et al. [52] is an exception; they compute backup paths that aim to minimize the total bandwidth reserved by all backup paths.

59, 67, 76] is tailored to support Internet-based applications that typically have a large number of short-lived multicast groups and dynamic group membership.

Our algorithms for installing backup trees avoid the scalability issues addressed by prior work [20, 27, 59]. SDN allows backup trees to be pre-computed offline and stored (in the case of REACTIVE) at the controller, thus introducing no extra forwarding state at the switches. In the case of PROACTIVE, where backup trees are pre-installed in the network, managing extra forwarding state is tractable because the smart grid is many orders of magnitude smaller than the Internet and smart grid multicast group membership is mostly static [8] (for example, a utility company subscribing to a PMU data stream are likely to always want to receive updates from this PMU). As a result, the volume of pre-installed state is manageable and requires infrequent updates.

In the context of OpenFlow, Kotani et al. [47] propose an approach for fast switching between IP multicast trees, where each multicast group has two multicast trees, a primary tree and a backup tree. Each tree is assigned a unique tree ID and both trees are installed in the network, but only the primary tree is used during normal operation. After a link failure, the root node is signaled to write the backup tree ID in each packet header to force packets to be forwarded using the backup tree. PROACTIVE uses a similar backup tree ID to quickly activate a pre-installed backup tree. However, PROACTIVE takes advantage of common forwarding state between a backup tree and its primary tree to reduce the amount of pre-installed state, while Kotani et al. [47] wastefully pre-installs forwarding rules at each switch in the backup tree.

4.5 Evaluation

We implement each algorithm from Section 4.3 in the POX OpenFlow controller [57] and run emulations using the Mininet 2.0.0 virtualization environment [50]. Emulations run on a Linux machine with four 2.33GHz Intel(R) Xeon(R) CPUs and

15GB of RAM. Mininet is configured to run inside Oracle's VirtualBox [15] virtual machine and is allocated 4GB RAM and a single CPU. All generated virtual networks use Mininet's default software switch, Open vSwitch [16]. The fidelity of Mininet emulations are discussed in [35, 38, 50, 70]. Unless otherwise noted, the APPLESEED controller algorithm runs inside the VirtualBox VM.

4.5.1 Link Failure Detection Emulations

We run two sets of Mininet-based emulations to evaluate PCOUNT. First, we measure the accuracy of PCOUNT loss probability estimates and quantify how accuracy improves as more flows are monitored. Then, we consider how controller and switch processing time increases as PCOUNT monitors more flows.

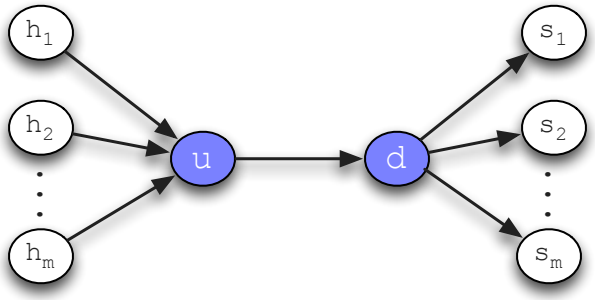
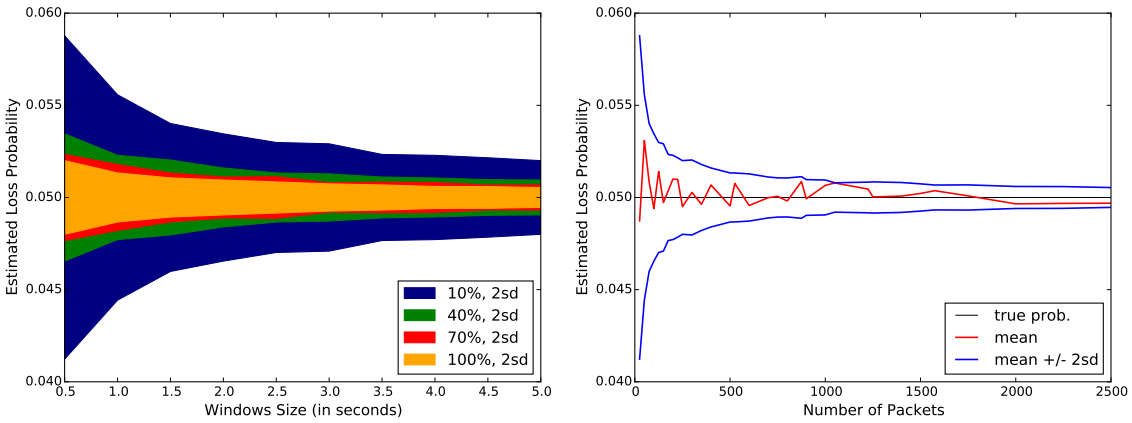


Figure 4.5. Dumbbell topology used in the PCOUNT evaluation.

Accuracy of Loss Probability Estimates. For the dumbbell topology shown in Figure 4.5, we use PCOUNT to measure the packet loss over link (u, d). We generate m multicast groups where each $h_1, h_2, ..., h_m$ multicasts packets to terminal nodes $s_1, s_2, ..., s_m$ at a constant rate of 60 packets per second, the standard sampling rate of PMUs. BASIC is used to implement multicast, resulting in m separate flow table entries at u and d. At the end of the section we comment on how our results apply

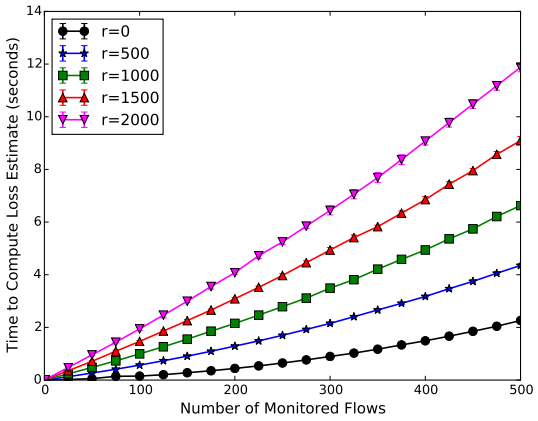
[15] https://www.virtualbox.org/

[16] http://openvswitch.org/



(a) Loss estimates as function of PCOUNT window size.

(b) Loss estimates as function of number of packets.



(c) Processing time, with 95% confidence intervals, as a function of number of monitored flows.

Figure 4.6. PCOUNT results monitoring a single link, (u, d), from Figure 4.5.

to MERGER. We let $m = \{10, 20, 30, 40, 50\}$ [17] and, using Mininet, drop packet traversing (u, d) using a Bernoulli process with loss probability $p = \{.01, .05, .10\}$.

In this emulation, we quantify how the accuracy of PCOUNT loss estimates – measured relative to the true underlying loss rate, p – as we modify the number of flows PCOUNT monitors. Recall from Section 4.3.1 that PCOUNT accounts for

[17] Emulations run prohibitively slow for $m > 50$ due to CPU overload.

every dropped packet of a flow it monitors, meaning that the only error in PCOUNT estimates results from unmonitored flows. Because the same trends hold across all m and p values, we describe here only a single representative case, where $m = 10$ and $p = .05$.

Recall from Section 4.3.1 that PCOUNT can measure packet loss over more general structures than a single physical link. Notably, in Figure 4.5, u and d may be connected with a multi-hop path containing several (non-OpenFlow) switches and routers; PCOUNT requires only that u and d are OpenFlow-enabled switches. We encourage the reader to interpret the results in this section in this broader context.

Figure 4.6(a) compares the 95% confidence intervals of PCOUNT's link loss probability estimates – centered around the true loss probability (.05) for consistency – as a function of window size $w = \{0.5, 1, ..., 5\}$ seconds. PCOUNT is configured such that each measurement window starts only after the packet loss from the previous window has been computed. Results are shown where PCOUNT monitors $k = \{10\%, 40\%, 70\%, 100\%\}$ of (u, d) flows (each monitored flow is selected randomly). The confidence intervals for each w, k pair are computed over 100 emulation runs.

PCOUNT loss rate estimates are extremely accurate: the 95% confidence interval, across all w and k, lies within 15% of the true loss probability. This is the case even when PCOUNT's estimate is based on only 30 packets (occurs when $k = 10\%$ and $w = 0.5$). Figure 4.6(b), which plots link loss probability estimates as a function of the number of packets considered during each emulation run, shows that after PCOUNT considers 75 packets, 80% of the emulations are within 5% of the true loss probability. As expected, PCOUNT accuracy increases with larger k. For each k, the standard deviation (of PCOUNT loss probability estimates) decreases as a function of the square root of w.

Processing Time. Next, we quantify how PCOUNT processing time increases when PCOUNT monitors additional flows. We measure packet loss over (u, d) from

Figure 4.5. Processing time is measured as the time between when PCOUNT sends its first statistic query and when PCOUNT computes its packet loss estimate. Recall from Section 4.3.1 that if PCOUNT monitors packet loss of k flows traversing (u, d), PCOUNT sends k statistic queries to u and one aggregate query to d.

PCOUNT is configured such that each measurement window starts only after the packet loss from the previous window has been computed. Additionally, PCOUNT window size is fixed to 2 seconds. Because Mininet multiplexes CPU resources using the default Linux scheduler, we found that running the constant rate PMU flows introduces unwanted CPU contention, adding noise to our results. For this reason, we create only a single multicast group (with source h_1 and a single sink s_1) but do not actually send any packets between the two hosts. Thus, we measure the time to send the statistic queries from the controller, process each query at the switch, and receive the query results at the controller. To further reduce CPU contention, we run PCOUNT as a remote control application, outside of the VirtualBox VM.

As computed, processing time accounts for (a) the time at the controller to generate $k+1$ statistic queries, (b) the transmission delay associated with sending the $k+1$ statistic queries from the controller to u and d, (c) the network delay in sending each statistic query from the controller to switch, (d) total time to process the statistic query at u and d, (e) the delay in sending the $k + 1$ query results from switch to controller, and (f) the latency in receiving and recording statistic query replies at the controller. We subtract (c) and (e), the network delay between controller and switches, from the measured processing times. Because the combined delay of (a), (b), and (f) accounts for less than 1% of the overall processing time, part (d), the time to process statistic queries at u and d, determines the overall processing times.

Figure 4.6(c) shows the processing time, computed as described in the previous paragraph, as a function of the number of flows PCOUNT monitors, k. [18] Each data point is the mean computed over 50 emulation runs. To measure the effect of flow table size on query processing time, we install r additional flow table entries at u and d.

We find that processing time increases roughly quadratically with k and there is a significant gap in processing time between each r (where r denotes the number of flow table entries installed at u and d that are not part of the PCOUNT tagging and counting). In practice, we expect non-empty flow tables so the $r = 0$ curve is overly optimistic. Therefore, to reasonably achieve sub-second processing time, our results show that fewer than 75 flows can be monitored.

Because the switches are completely idle during each emulation run, except for the time to process the read state queries, and the software switches used have considerably more powerful CPUs relative to hardware switches [21, 70], these results likely underestimate processing time. Nonetheless, these results underscore the high cost in monitoring and (in particular) querying a large number of flows.

Summary. The slow processing times associated with monitoring large numbers of flows and the highly accurate loss estimates for even small k strongly suggest that k should be small. Because the software switch skews the processing time results in favor of PCOUNT, we expect that even a stronger case for using small k can be made using hardware switches. We also note that the (Bernoulli) loss process favors PCOUNT because loss rates are uniform across all flows traversing (u, d) and loss events are i.i.d..

---

[18] Fake multicast groups and corresponding flow table entries are generated and installed at u and d in cases where PCOUNT monitors more than the 1 multicast group.

4.5.2 Backup Tree Installation Emulations

In this section, we emulate the failure of a single link and then measure recovery time, control plane signaling, and garbage collection overhead for PROACTIVE and REACTIVE both with and without the MERGER optimization. We aim to answer the following questions with these emulations:

- How effective is BUNCHY in reusing primary tree edges and in providing opportunities for backup-tree installation algorithms to reuse primary-tree forwarding rules?
- How much faster does PROACTIVE recover from link failure than REACTIVE?
- How many fewer control messages are needed to install backup trees under PROACTIVE versus REACTIVE?
- How much control state does PROACTIVE pre-install?
- In terms of recovery time, control plane signaling, and garbage collection, how much does MERGER improve performance relative to BASIC?

Setup. We use IEEE bus systems 14, 30, 57, and 118 [19] and synthetic graphs based on these IEEE bus systems to evaluate our algorithms. Each bus system consists of buses – electric substations, power generation centers, or aggregation points of electrical loads – and transmission lines connecting those buses. The IEEE bus systems are actual portions of the North American transmission network, where PMUs are being deployed. Synthetic graphs are generated using a procedure described in Section 3.5 of Chapter 3 that uses an IEEE bus system as a template to generate graphs with the same degree distribution as the template bus system.

[19] `http://www.ee.washington.edu/research/pstca/.`

We assume that the communication network mirrors the physical bus system topology, that an OpenFlow switch is co-located at each bus, and that two unidirectional communication links, one in each direction, connects these switches following the same configuration as the bus system's transmission lines. Additionally, we connect each switch with a leaf host using a bidirectional communication link. In this setup, the PMUs measure voltage and current phasors at the buses, then these measurements are sent by the bus's attached host to its first-hop switch, which then multicasts the PMU measurements using the network of OpenFlow switches to a set subscribing hosts (terminals).

For each bus system n, we generate synthetic topologies with n switches, n hosts, and set all link weights to 1. Then, we randomly create $m = \{1, 2, ..., \frac{n}{2}\}$ multicast groups, each with $n/3$ random terminal hosts, and use the Steiner-Arborescence approximation proposed by Charikar et al. [17] to compute the m primary trees. Bunchy, with $\alpha = 1.1$, is then used to pre-compute, for each primary tree, a backup tree for each primary tree link. Next, a random communication link, l, that is used by at least one primary tree is chosen to fail (i.e., drop enough packets to trigger a Pcount alert), triggering the installation of backup trees using either Reactive or Proactive. For each m, we generate 35 different synthetic graphs and 3 random sets of multicast groups, yielding a total of 105 emulation runs per m.

The results described in the remainder of this section are those from synthetic topologies generated using IEEE bus system 57 as a template. The trends are consistent across all other networks. Switches in the networks generated using IEEE bus system 57 have an average diameter of 11.75 and average degree 3.74. For each of the m multicast groups, we initially attempted to multicast packets at a constant rate flow of 60 packets per second from the root host but this caused CPU emulator overload. Instead, in each emulation run we only initiated the constant rate flows for the primary trees using the failed link.

4.5.2.1 Bunchy Results

The primary trees computed using the STEINER-ARBORESCENCE approximation described in Section 4.3.2.2 have an average root-to-terminal hop count of 7.54, while the BUNCHY backup trees are slightly larger with an average end-to-end (E2E) length of 8.4. Based on the E2E latency requirements reported in Section 4.2.1, the per-link delay in our emulated topologies would need to be in the range of $0.6 - 1$ms to satisfy QoS requirements using these multicast trees. [20]

On average, the BUNCHY backup trees have stretch of 1.17. Stretch is defined per multicast tree and is the ratio of path length from the root to terminal along the multicast tree to the length of the shortest unicast path in the graph to that terminal. For comparison with BUNCHY results, we compute a second backup tree for each primary tree, T_i^l, by running the STEINER-ARBORESCENCE approximation described in Section 4.3.2.2 over the original graph with l removed. We denote the set backup trees for l computed using this algorithm as B^l.

The B^l backup trees are marginally smaller than the $\hat{T}^l$ backup trees: $w(\hat{T}^l)/w(B^l) = 1.08$. Recall that $\hat{T}^l$ refers to a set of backup tree computed by BUNCHY and $w(\hat{T}^l)$ denotes, for all $\hat{T}_i^l \in \hat{T}^l$, the sum of $\hat{T}_i^l$'s link weights. This is expected because $\hat{T}^l$ is computed using a heuristic to guide BUNCHY to reuse primary tree edges, while B^l trees are an approximation of the least cost directed tree (and so are computed independently of the edges used by the primary tree).

However, $\hat{T}^l$ reuses more primary tree edges ($\hat{T}^l$ reuses 59% of primary tree edges versus 41% under B^l). Most importantly, when comparing $\hat{T}^l$ and B^l with the primary tree, $\hat{T}^l$ has more common nodes with the primary tree that have the same children

[20] The average root-to-terminal path lengths were largest for IEEE bus system 57 and the synthetic graphs based on this bus system. The multicast trees computed for bus system 118 have an average E2E path length approximately 1 hop fewer than those for bus system 57, even though IEEE bus system 118 has more than twice as many nodes as bus system 57. This is mainly due to bus system 118's higher density (than bus system 57).

in $\hat{T}^l$ and the primary tree (55% of $\hat{T}^l$ nodes) than B^l (38% of B^l nodes). This last point is important because this allows $\hat{T}^l$ to reuse more primary tree rules once $\hat{T}^l$ is installed, thus requiring fewer rules to install after a failure. In summary, these results suggest that BUNCHY computes backup trees only slightly larger than an approximation of the least cost tree with a significant gain in primary tree edge and forwarding rule reuse.

4.5.2.2 Signaling Overhead

Next, we compare the number of control messages required to install backup trees as function of the number of primary trees (m) installed in the network. Figure 4.7(a) shows the results for REACTIVE running in BASIC and MERGER mode, referred to as REACTIVE+BASIC and REACTIVE+MERGER, respectively; a lower bound for REACTIVE (REACTIVE+LB); and PROACTIVE in BASIC mode, denoted as PROACTIVE+BASIC. Note that the results for PROACTIVE are the same using BASIC and MERGER because PROACTIVE only requires that the root node of each backup tree needs to be signaled to activate the backup. We shall later see how BASIC and MERGER affect the number of forwarding rules PROACTIVE pre-installs.

Reactive+LB summary. REACTIVE+LB computes the lower bound on the number of new rules required to install backup trees $\hat{T}^l$ after l fails. Informally, REACTIVE+LB finds the number of unique sets of outgoing links used by $\hat{T}^l$ at each node, v, since at least this many rules are required to forward $\hat{T}^l$ packets at v. These values are summed across all nodes used by $\hat{T}^l$ to find a lower bound on the total number of control messages necessary to install backup trees $\hat{T}^l$.

In more detail, REACTIVE+LB first marks each edge and node used by each $\hat{T}^l_i \in \hat{T}^l$. Then, each marked node, v, is processed. The lower bound computation finds the set of outports used by all primary trees installed in the network and the outports used by $\hat{T}^l$. Any $\hat{T}^l_i \in \hat{T}^l$ with the same outports as a primary tree does not

require a new forwarding rule because $\hat{T}_i^l$ can reuse the primary tree forwarding rule. Among the remaining backup trees, REACTIVE+LB finds the number of unique sets of outports, b, used by these trees. We claim that b is equal to the minimum number of forwarding rules that must be installed at v. Consider the case where fewer than b rules are installed at v. This would imply that packets corresponding to at least one $\hat{T}_j^l \in \hat{T}^l$ would not match with a rule that forwards packets out the complete set of ports associated with $\hat{T}_j^l$. Therefore, it must be the case that least b new rules are required at v.

Finally, REACTIVE+LB sums the b values computed at each $v \in \hat{T}^l$ and returns this value as the lower bound on the number of new rules required to install $\hat{T}^l$. Note that we can reason about the number of forwarding rules required at each node separately because the lower bound at each node, v, depends only the set of outgoing links used by $\hat{T}^l$ at v.

Results. As expected, we find that PROACTIVE requires less signaling overhead than REACTIVE, including even REACTIVE+LB. PROACTIVE activates the backup trees by sending a single control message (to install a pre-computed forwarding rule) to the root switch of each of backup tree using the failed link, whereas REACTIVE must signal multiple switches to install each backup tree.

For REACTIVE, the gap between BASIC and MERGER increases as we introduce more primary trees. When $m = 1$ there are no opportunities for MERGER to consolidate forwarding rules so MERGER and BASIC require exactly the same number of control messages to install the backup tree.

As m grows, three factors contribute to an increasing gap between BASIC and MERGER. First, there are more primary tree forwarding rules (installed in the network) that MERGER can reuse. Our results show that for $m \geq 7$, 75% of MERGER savings (versus BASIC) are due to reusing primary tree forwarding rules. Second, as m increases, more graph edges are used: when $m = 28$, 90% of all network links are used

by at least one primary tree and is at least 80% for $m \geq 10$. This benefits MERGER because it increases the likelihood that at any switch a backup tree shares the same outgoing links as at least one primary tree. Finally, as m increases more primary trees are affected by a link failure causing more backup trees to be installed for each link failure. This provides additional opportunities for MERGER to consolidate flow table entries with other backup trees. However, this third factor is less significant than the previous two, as only 25% of MERGER savings are due to consolidating flows with other backup trees.

With REACTIVE, MERGER does well compared with LB. On average, MERGER requires 25% more control messages than LB, suggesting that MERGER's local optimization does not miss many opportunities for consolidating flows.

4.5.2.3 Time to Install Backup Trees

Here we compare the time required by each of algorithms to install backup trees. Specifically, we measure the time between when the link failure is detected at the controller to when *all* pre-computed backup tree forwarding rules are installed at the network switches. We refer to this time duration as t_c. t_c is a function of the controller transmission delay (i.e., the time between when the first and last precomputed control messages are sent from the controller), the controller to switch RTT, and the time to install a forwarding rule at a switch.

We find the transmission delay to be negligible: on average, transmission delay is less than 2.8% and 0.9% of the time to install a *single* flow table entry at a switch, for REACTIVE+BASIC and REACTIVE+MERGER, respectively. Even if we conservatively assume that the inter-arrival time of installation messages at each switch is equal to the total transmission delay (i.e., the time to send all pre-computed forwarding rules for $\hat{T}^l$), it follows that each switch receives all control messages before completing the installation of its first backup tree rule. Because rules are installed in parallel across

switches, the time to install all backup tree rules occurs when the switch with the most backup tree rules to install, s_x, installs its last rule. If we assume s_x has c_x rules to install, let t_d be the total transmission delay, and denote the average latency to install a single rule at an OpenFlow switch as t_i, then

$$t_c = \frac{1}{2}RTT + t_d + c_x(t_i)$$

Because Mininet's software switches lack performance fidelity in terms flow table entry installation time [70], we determine t_i values using measurement results from the literature [29], rather than measure rule installation times in Mininet. Specifically, we assume the mean installation time per rule is 7.12ms, as reported by Ferguson et al. [29] using the Pronto 3290 OpenFlow switch running the Indigo 2012.09.07 firmware.

Figure 4.7(b) shows the estimated elapsed time to install all backup trees as a function of m. We set $RTT = 0$ in this emulation. The trends for each algorithm are a function of their c_x values, the maximum number of rules any switch must install, found at each m. With PROACTIVE, c_x is always 1. The difference in total installation time between REACTIVE+BASIC and REACTIVE+MERGER is small in absolute terms (at most 25ms) because the install times depend on the amount of rule consolidation each algorithm is able to apply at a single switch, s_x, rather than the level of rule sharing possible at multiple switches (as we observed with signaling overhead results). Nonetheless, the extra milliseconds saved using MERGER can be valuable to critical PMU applications.

4.5.2.4 Switch Flow Table Size

In our emulations in Section 4.5.2.2 we found that PROACTIVE incurs less signaling overhead than REACTIVE. Here we show that these savings come at a cost: PROACTIVE's pre-installed forwarding rules can account for a significant portion of limited OpenFlow switch capacity.

Using the same setup as the other emulations in this section, we record the number of pre-installed backup tree rules at each switch during each emulation run. Figure 4.7(c) shows the mean number of pre-installed rules per switch, r, as a function of m. The confidence intervals are omitted because of high variance (during each emulation run, individual counts range from 0 to 525.) REACTIVE is not included because it does not pre-install forwarding rules. The number of pre-installed rules for PROACTIVE+LB is computed using the same algorithm described in Section 4.5.2.2 for REACTIVE+LB.

Similar to our REACTIVE signaling overhead findings, MERGER yields up to 2.5 times better performance than BASIC. MERGER and LB savings increase with m because more primary tree flows are reused and more backup tree rules are shared as m grows. As a result, the number of pre-installed forwarding rules per backup tree decreases linearly as m increases causing the rate at which r increases for PROACTIVE+MERGER to slow.

In contrast, r increases linearly with m using PROACTIVE+BASIC. For each backup tree, BASIC is only able to avoid pre-installing a forwarding rule at a switch, v, if the backup tree uses the same outports as its primary tree at v. Because this condition depends only on the relationship between a backup tree and its primary tree, the number of pre-installed rules per backup tree is constant for all m. Since larger m implies more backup trees, r increases linearly with m.

Based on maximum flow table sizes of real OpenFlow switches (ranging from approximately 1500 to 1900 flow table entries [21, 29]), the number of pre-installed rules, r, at most accounts for 19% and 6.7% of flow table capacity for PROACTIVE+BASIC and PROACTIVE+MERGER, respectively. We find that the switch with the most pre-installed forwarding rules (across all emulation runs) has at most has 525 pre-installed forwarding rules (occurs with PROACTIVE+BASIC when $m = 28$). At most, this accounts for 35% of the switch's flow table capacity.

4.5.2.5 Garbage Collection Overhead

After a link l fails, forwarding rules corresponding to primary trees using l may become stale, as discussed in Section 4.3.4. These stale rules are deleted as part of APPLESEED garbage collection. In this section, we first we describe garbage collection results for REACTIVE and then for PROACTIVE using the same emulation setup described at the start of Section 4.5.2. REACTIVE+LB and PROACTIVE+LB are both computed based on the LB algorithm described in Section 4.5.2.2.

Reactive Garbage Collection. Figure 4.7(d) shows a modest change in the number of stale rules between BASIC, MERGER, and LB. For each of these algorithms, on average 55% of primary tree rules are reused by its backup tree (and are therefore not garbage collected). [21] This implies that REACTIVE+MERGER and REACTIVE+LB are only able to reduce garbage collection over the remaining 45% of primary tree rules. Among these remaining primary tree rules, REACTIVE+MERGER and REACTIVE+LB reduce garbage collection when any backup tree for l reuses a primary tree rule. Our results show that this yields small savings in garbage collection.

Proactive Garbage Collection. Compared with REACTIVE, we find a significant decrease in stale flows with PROACTIVE because PROACTIVE installs up to two orders of magnitude more backup trees, providing more opportunities for primary tree forwarding rules to be reused. Recall that REACTIVE only installs the backup trees for l, whereas PROACTIVE pre-installs, for each primary tree, a backup tree for each primary tree link, amounting to approximately 32 backup trees per primary tree. As a result, we observe 2.5 times fewer stale rules with PROACTIVE+BASIC versus REACTIVE+BASIC and PROACTIVE+MERGER has up to an order of magnitude decrease in garbage collection versus REACTIVE+MERGER.

---

[21] Because all three algorithms use BUNCHY to compute backup trees this statistic is the same for REACTIVE+BASIC, REACTIVE+MERGER, and REACTIVE+LB.

The number of stale PROACTIVE+MERGER forwarding rules actually decreases as m grows. Recall that for a primary tree using l, any of its rules is not stale if at least one other primary or backup tree uses this flow table entry. Because the number of pre-installed backup trees is so large (approximately $32m$), for large m, nearly all primary tree rules are still used after a link failure (resulting in a decrease in stale flow table entries as m grows).

4.5.2.6 Summary

In summary, we have shown quantitatively that as more primary trees are installed in the network, the gap between MERGER and BASIC grows (for both REACTIVE and PROACTIVE) in terms of signaling overhead, total backup tree installation time, number of pre-installed forwarding rules, and garbage collection overhead. The is the case because, with larger m, there are more opportunities for MERGER to reuse primary tree rules and consolidate rules among other backup trees.

Additionally, we found PROACTIVE yields fewer control messages and faster recovery than REACTIVE – REACTIVE sends up to 10 times more control messages than PROACTIVE – but at the cost of storage overhead at each switch. PROACTIVE's pre-installed backup trees can account for as much as 35% of the capacity reserved for wild-card matching rules of a conventional OpenFlow switches [21]. However, when applying MERGER to PROACTIVE, this statistic drops to 20% and, on average, PROACTIVE+MERGER accounts for only 6.7% of flow table capacity.

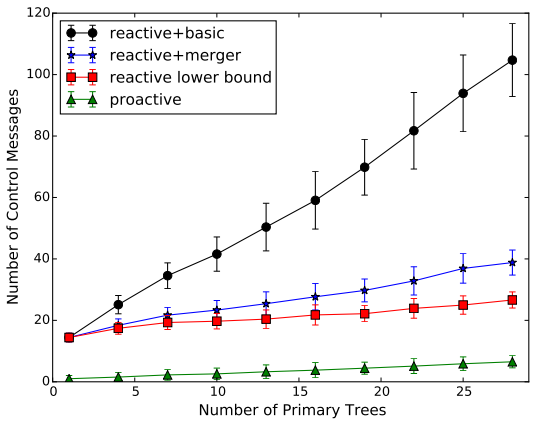
4.6 Conclusions

In this chapter we have addressed an important challenge in reliable multicasting of critical Smart Grid data. We designed, implemented, and evaluated a suite of algorithms that collectively provide fast packet loss detection and fast rerouting using pre-computed backup multicast trees. Because this required making changes to

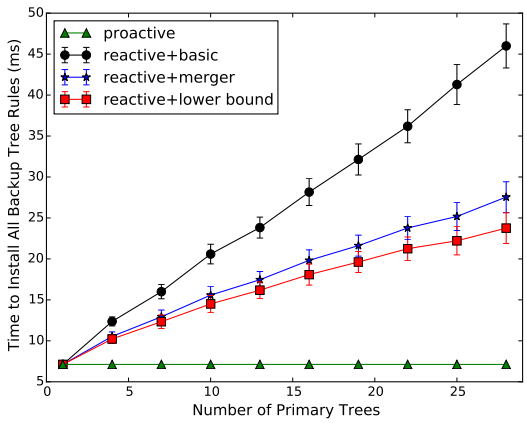
network switches, we used OpenFlow to modify switch forwarding tables to execute these algorithms in the data plane.

First, we presented PCOUNT, an algorithm that used OpenFlow primitives to accurately detect per-link packet loss inside the network rather than using slower end-to-end measurements. Next, we formulated a new problem, MULTICAST RECYCLING, that considered computing backup trees that reuse edges of already-installed multicast trees as a means to reduce control plane signaling. MULTICAST RECYCLING was proved to be at least NP-hard so we designed an approximation algorithm called BUNCHY. Lastly, we presented two algorithms, PROACTIVE and REACTIVE, that installed backup trees at OpenFlow controlled switches. As an optimization to PROACTIVE and REACTIVE, we introduced MERGER, an algorithm that consolidated forwarding rules at switches where multiple trees had common children.

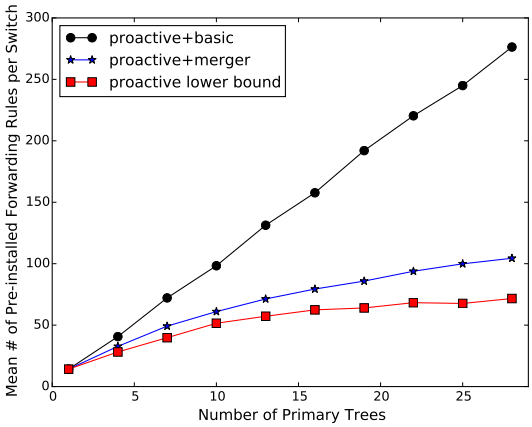
Mininet emulations were used to evaluate our algorithms over communication networks that mirrored the structure of IEEE bus systems (actual portions of the North American power grid). We found PCOUNT estimates were accurate when monitoring even a small number of flows over short time window: after sampling only 75 packets, the 95% confidence interval of PCOUNT loss estimates were within 15% of the true loss probability. By pre-installing backup trees, PROACTIVE resulted in up to a ten-fold decrease in control messages compared with REACTIVE, which had to signal multiple switches to install each backup tree. However, in scenarios with many multicast groups, PROACTIVE's pre-installed forwarding rules accounted for a significant portion of scarce OpenFlow switch table capacity (up to 35% of a standard OpenFlow switch). Fortunately, MERGER reduced the amount of pre-installed forwarding state by a factor of $2 - 2.5$, to acceptable levels.



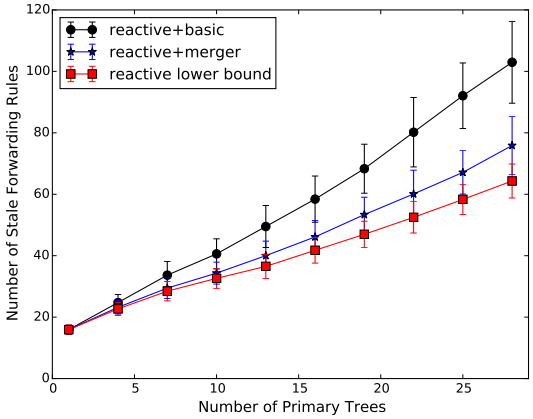
(a) Number of control messages to activate backup trees.



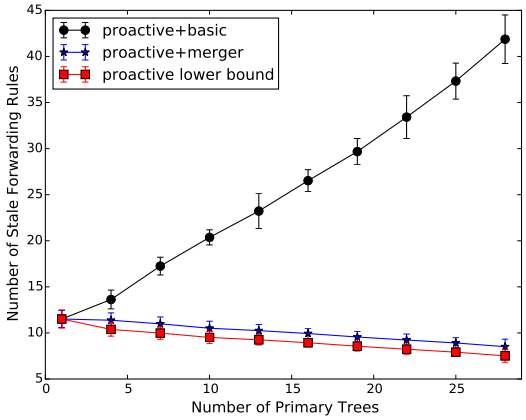
(b) Total time to install all backup tree flow table entries.



(c) Proactive: mean number of pre-installed rules per switch.



(d) Reactive: number of stale flow table entries resulting from link failure.



(e) Proactive: number of stale flow table entries resulting from link failure.

Figure 4.7. Reactive and Proactive results for a single random link failure of synthetic topologies based on IEEE bus system 57. Each data point is the mean over 105 emulation runs and the 95% confidence interval is shown in all plots expect (c).

CHAPTER 5

THESIS CONCLUSIONS AND FUTURE WORK

5.1 Thesis Summary

This thesis presented algorithms to make communication networks robust to component failures. Three separate but related problems were considered: node (i.e., switch or router) failure in traditional networks such as the Internet or wireless sensor networks, the failure of critical sensors that measure voltage and current throughout the smart grid, and link failures in a smart grid communication network.

Chapter 2 considered scenarios where a malicious node injects and spreads false routing state throughout a network of routers. We presented and evaluated three new algorithms – 2ND-BEST, PURGE, and CPR – for recovery in such scenarios. Among these algorithms, we found that CPR – a checkpoint-rollback based algorithm – yielded the lowest message overhead and convergence time over topologies with fixed link weights but at the cost of storage overhead at the routers. For topologies where link weights could change, PURGE performed best because PURGE globally invalidated false routing state, helping PURGE avoid the problems that plagued CPR and 2ND-BEST: updating large amounts of stale state (CPR) and the count-to-infinity problem (2ND-BEST).

Next, in Chapter 3 we studied PMUs – critical sensors being deployed in electric power grids worldwide that provide voltage and current measurements to power grid operators – and a set of placement problems that considered detecting PMU measurement errors. We formulated four PMU placement problems that considered two constraints: place PMUs "near" each other to allow for measurement error detection

and use the minimal number of PMUs to infer the state of the maximum number of system buses and transmission lines. Each PMU placement problem was proved to be NP-Complete. As a first step, we proposed and evaluated a simple greedy approximation algorithm to each placement problem. Using simulations based on topologies generated from real portions of the North American electric power grid, we found our greedy algorithms consistently reached close-to-optimal performance (on average within 97% of optimal). Additionally, our simulations showed that requiring PMUs to placed near each other (in order to detect measurement errors) resulted in only a small decrease in system observability (on average only 5% fewer buses were observed with this additional constraint), which made for a strong case for imposing this requirement.

In our final technical chapter, we designed algorithms that provide fast recovery from link failures in a smart grid communication network. We proposed, designed, and evaluated solutions to all three aspects of link failure recovery: link failure detection, algorithms that pre-computed backup multicast trees, and fast backup tree installation. Because these algorithms required making changes to network switches, these algorithms used OpenFlow to access and modify the forwarding plane of switches.

As an alternative to slower algorithms based on end-to-end measurements, we presented PCOUNT. PCOUNT used OpenFlow primitives to detect and report link failures inside the network. Next, a new problem was formulated, MULTICAST RECYCLING, that considered computing backup trees that reuse edges of already installed multicast trees as a means to reduce control plane signaling. MULTICAST RECYCLING was proved to be at least NP-hard so we designed an approximation algorithm for MULTICAST RECYCLING. Lastly, we presented two algorithms, PROACTIVE and REACTIVE, that installed backup trees at OpenFlow controlled switches. As an optimization to PROACTIVE and REACTIVE, we designed MERGER, an algorithm that consolidated forwarding rules at switches where multiple trees have common children.

These algorithms were evaluated with Mininet emulations using communication networks that mirrored the structure of actual portions of the North American power grid. PCOUNT packet loss estimates were accurate when monitoring even a small number of flows over short time window: after sampling only 75 packets, the 95% confidence interval of PCOUNT loss estimates were within 15% of the true loss probability. PROACTIVE had a $10x$ decrease in control messages compared with REACTIVE because PROACTIVE required only a single control message to install each backup tree since all other rules were pre-installed, whereas REACTIVE had to signal multiple switches to install each backup tree. However, PROACTIVE's pre-installed forwarding rules accounted for a significant portion of scarce OpenFlow switch table capacity, especially in cases with many multicast groups (up to 35% of flow table capacity of a standard OpenFlow switch). Fortunately, MERGER reduced the amount of pre-installed forwarding state by a factor of $2 - 2.5$, to acceptable levels.

5.2 Future Work

Our research in Chapter 2 only considered a single instance of false state where we assumed that the compromised node falsely claimed the minimum distance to all nodes. As future work, we are interested in exploring how our algorithms (i.e, 2ND-BEST, PURGE, and CPR) respond to other possible false state values. Some interesting alternatives include false state that maximizes the effect of the count-to-infinity problem and false state that contaminates a bottleneck link. We would also like to see how our distributed recovery algorithms compare with a Software Defined Networking (SDN) based approach to false state recovery. It is likely that the concerns over convergence time addressed by our distributed recovery algorithms are non-factors with an SDN approach. With SDN, recovery paths can be computed centrally at the controller (as we did when computing backup multicast trees in Chapter 4), negating the need for switches to exchange messages to compute new

paths. However, new challenges are likely to emerge with an SDN-based approach. For example, in what order should routers be signaled to install new routes such that the count-to-infinity problem is minimized?

There are several topics for future work from Chapter 3 on PMU placement. The success of the greedy PMU placement algorithms suggests that bus systems have special topological characteristics, and investigating these properties could provide interesting insight to power grid topologies. Because our brute-force optimal algorithm could only produce data points for small inputs, much could be learned by implementing the integer programming approach proposed by Xu and Abur [77] to solve FullObserve. This would provide valuable data points to measure the relative performance of `greedy`.

For the PMU placement problems from Chapter 3 (i.e., MaxObserve, FullObserve, MaxObserve-XV, FullObserve-XV), there are a number of alternative objective functions that could be used in place of the one used by each placement problem that could lead to interesting new problem formulations both from a theoretical and practical perspective. For example, we could associate a utility to observing each node and define an objective function that aims to maximize a utility function, defined using the utility associated with observing each node. This would remove our implicit assumption that observing each node yields the same utility. Another interesting objective function would be to minimize the distance (e.g., number of hops) between each unobserved node and its nearest observed node. Intuitively, this objective function seeks to ensure that no large subgraphs are completely unobserved, which could make decisions of where to put the "next" PMU easier. A third alternative objective function is to maximize the number of observed links rather than the number of observed nodes. This objective function is closely related to observing the maximum number of nodes because PMUs must be placed at nodes and, doing so, results in the observation of all links incident to the node where the PMU is placed.

From Chapter 4, several problems still remain to be solved. One problem of interest is using optimization criteria different from MULTICAST RECYCLING's objective function to compute backup trees and then evaluate PROACTIVE, REACTIVE, and MERGER performance using these backup trees. For example, backup trees may be computed with the goal of protecting against the worst-case impact of a subsequent link failure by minimizing the maximum number of multicast trees using a single link. It is unknown how effective our installation algorithms would be given these types of backup trees.

Measurements using real OpenFlow hardware switches would strengthen our PCOUNT processing time and backup tree installation time results, which both suffered from inaccuracies due to Mininet's performance fidelity issues. At the end of Section 4.3.1 we commented on how PCOUNT can be easily extended to monitor packet loss between multiple non-adjacent switches. We showed that in some cases packet loss at all links connecting switches used in the same multicast tree can be estimated using only a single PCOUNT session with measurement points at only a subset of these switches. It would be interesting to quantify the savings (in terms of switch processing time) of this approach when compared to a naive implementation that runs separate PCOUNT sessions between all adjacent switches. Our PCOUNT simulation results suggest that these savings could be significant. Lastly, the problem MERGER addresses – find the minimum number of forwarding rules for a set of multicast trees – has unknown complexity. We conjectured that this problem is NP-hard in Section 4.3.5.4.

This thesis provided some encouraging initial results of how SDN (and specifically OpenFlow) can simplify fault detection and recovery but we did so under somewhat favorable conditions. For example, in Chapter 4 we assumed that any non-OpenFlow switches or routers had no influence on our recovery algorithms (this is equivalent to assuming that all network switches support OpenFlow). In practice, it is likely that OpenFlow switches will coexist with existing network infrastructure (e.g., IP routers

and switches), which will likely complicate matters. One potential issue is that many backbone IP routers use MPLS to reroute flows in response to link failures. This would result in new paths between OpenFlow switches. In these cases, it is unclear if OpenFlow switches and the control plane need to be aware of these path changes. Also what is the best way for the OpenFlow controller to monitor the state of non-OpenFlow switches and routers? Would it be sufficient to passively monitor control messages sent among IP routers? If so, how much control state needs to be tracked and what is the cost of doing so?

Our hope is that the preliminary results in Chapter 4 will encourage other researchers to develop OpenFlow-based solutions for smart grid communication. One promising topic, not addressed in this thesis, is traffic engineering, which figures to play an important role in smart grid data dissemination. We believe OpenFlow's capabilities to directly control traffic flows makes OpenFlow well-suited to designing simple and effective traffic engineering solutions for the smart grid.

Although using OpenFlow and SDN for smart grid communication has many benefits (e.g., open access the forwarding plane of switches, clean separation of the control and data plane), using SDN introduces a potential new set of problems. For example, the controller and the path between the switch and controller become potential points of failure. [1] However, well-established solutions such as multipath routing [31] (between the controller and switches) and hot standby [72] (at the controller) can be used to mitigate these reliability issues. Another concern in using SDN is access control: due to administrative boundaries, it is unlikely that a single controller will have the ability to modify *any* switch. In practice, it likely that a number of peer controllers or possibly a hierarchy of controllers must work together to control an entire smart grid communication network. Again, this is not a new problem that

[1] This is not the case with a traditional network architecture where controller algorithms are distributed algorithms run at the network switches and routers.

must be solved and well-studied protocols can be used to coordinate communication between controllers.

We conclude this thesis by considering a broad question, does OpenFlow render all distributed computing in network control obsolete? In a word, no. We believe that there are some cases where control decisions are better made locally. For example, MPLS fast-reroute decisions are best made at the routers for increased speed because doing so avoids any signaling delay between the switch/router and controller.

APPENDIX A

PSEUDO-CODE AND ANALYSIS OF DISTANCE VECTOR RECOVERY ALGORITHMS

A.1 Recovery Algorithm Pseudo-Code

Building on the notation specified in Table 2.1 (from Section 2.2), we define some additional notation that we use in our pseudo-code specifications of 2ND-BEST, PURGE, and CPR. Let msg refer to a message sent during PURGE's diffusing computation (to globally remove false routing state). msg includes:

1. a field, src, which contains the node ID of the sending node
2. a vector, $\overrightarrow{dests}$, of all destinations that include $\overline{v}$ as an intermediary node.

Let Δ refer to the maximum clock skew for CPR.

Algorithm A.1.1: 2ND-BEST run at each $i \in adj(\overline{v})$

```
1: flag ← FALSE
2: set all path costs to v̄ to ∞
3: for each destination d do
4:    if v̄ is first-hop router in least cost path to d then
5:       c ← least cost to d using a path which does not use v̄ as first-hop router
6:       update min⃗_i and dmatrix_i with c
7:       flag ← TRUE
8:    end if
9: end for
10: if flag = TRUE then
11:    send min⃗_i to each j ∈ adj(i) where j ≠ v̄
12: end if
```

Algorithm A.1.2: PURGE's diffusing computation run at each $i \in adj(\overline{v})$

1: set all path costs to $\overline{v}$ to ∞
2: $S \leftarrow \emptyset$
3: **for each** destination d **do**
4: **if** $\overline{v}$ is first-hop router in least cost path to d **then**
5: $\overrightarrow{min}_i[d] \leftarrow \infty$
6: $S \leftarrow S \cup \{d\}$
7: **end if**
8: **end for**
9: **if** $S \neq \emptyset$ **then**
10: send S to each $j \in adj(i)$ where $j \neq \overline{v}$
11: **end if**

Algorithm A.1.3: PURGE's diffusing computation run at each $i \notin adj(\overline{v})$

1 **Input:** msg containing src, $\overrightarrow{dests}$ fields.
1: $S \leftarrow \emptyset$
2: **for each** $d \in msg.\overrightarrow{dests}$ **do**
3: **if** $msg.src$ is next-hop router in least cost path to d **then**
4: $\overrightarrow{min}_i[d] \leftarrow \infty$
5: $S \leftarrow S \cup \{d\}$
6: **end if**
7: **end for**
8: **if** $S \neq \emptyset$ **then**
9: send S to spanning tree children
10: **else**
11: send ACK to $msg.src$
12: **end if**

Algorithm A.1.4: CPR rollback

1: **if** already rolled back **then**
2: send ACK to spanning tree parent node
3: **end if**
4: $\hat{t} \leftarrow -\infty$
5: **for each** snapshot, S, **do**
6: $t'' \leftarrow S.timestamp$
7: **if** $t'' < (t' - \Delta)$ **and** $t'' > \hat{t}$ **then**
8: $\hat{t} \leftarrow t''$
9: **end if**
10: **end for**
11: rollback to snapshot taken at $\hat{t}$
12: **if** not spanning tree leaf node **then**
13: send rollback request to spanning tree children
14: **else**
15: send ACK to spanning tree parent node
16: **end if**

Algorithm A.1.5: CPR "steps after rollback" run at each $i \in adj(\overline{v})$

1: $flag \leftarrow$ FALSE
2: **for each** destination d **do**
3: **if** $\overrightarrow{min}_i[d] = \infty$ **then**
4: find least cost to d in $dmatrix_i$ and set in $\overrightarrow{min}_i$
5: $flag \leftarrow$ TRUE
6: **end if**
7: **end for**
8: **if** $flag =$ TRUE **or** adjacent link weight changed during $[t', t]$ **then**
9: send $\overrightarrow{min}_i$ to each $j \in adj(i)$ where $j \neq \overline{v}$
10: **end if**

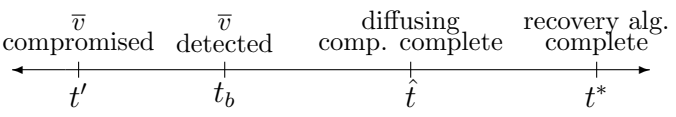


Figure A.1. Timeline with important timesteps labeled.

A.2 Correctness of Recovery Algorithms

Here we prove correctness for the 2ND-BEST, PURGE, and CPR algorithms described in Section 4.3. Our correctness proofs consider the general case where multiple nodes are compromised. We use the following notation in our proofs:

- We refer to the set of compromised nodes as $\overline{V}$.
- t_b marks the time at outside algorithm detects that all $\overline{V}$ are compromised.
- t' refers to the time the first $\overline{v} \in \overline{V}$ is compromised.
- t^* marks the time when the recovery algorithm (e.g., 2ND-BEST, PURGE, or CPR), which started executing at time t, completes.
- We use the definition of G described in Section 4.3.
- We redefine G' as follows. $G' = (V', E')$, where $V' = V - \overline{V}$, $E' = E - \{(\overline{v}, v_i) \mid \overline{v} \in \overline{V} \wedge v_i \in adj(\overline{v})\}$.

All important timesteps are shown in Figure A.1.

We make the following assumptions in our proofs. All the initial *dmatrix* values are non-negative. Furthermore, all $\overrightarrow{min}$ values periodically exchanged between neighboring nodes are non-negative. All $v \in V$ know their adjacent link weights. All link weights in G (and therefore G' as well) are non-negative and do not change. G is finite and connected. Finally, we assume reliable communication.

Definition 1. *An algorithm is correct if the following two conditions are satisfied. One, $\forall v \in V'$, v has the least cost to all destinations $v' \in V'$. Two, the least cost is computed in finite time.*

Theorem A.1. *Distance vector is correct.*

Proof. Bertsekas and Gallager [11] prove correctness for distributed Bellman-Ford for arbitrary non-negative *dmatrix* values. Their distributed Bellman-Ford algorithm is the same as the distance vector algorithm used in this thesis. □

Corollary A.2. 2ND-BEST *is correct when a single node is compromised.*

Proof. As per the preprocessing step, each $v \in adj(\overline{v})$ initiates a diffusing computation to remove $\overline{v}$ as a destination. For each diffusing computation, all nodes are guaranteed to receive a diffusing computation (by our reliable communication and finite graph assumptions). Further, each diffusing computation terminates in finite time. Thus, we conclude that each $v \in V'$ removes $\overline{v}$ as a destination in finite time.

After the diffusing computations to remove $\overline{v}$ as a destination complete, each $v \in adj(\overline{v})$ uses distance vector to determine new least cost paths to all nodes in their connected component. Because all $dmatrix_v$ are non-negative for all $v \in V'$, by Theorem A.1 we conclude 2ND-BEST is correct if no additional node(s) are compromised during $[t', t^*]$. □

Corollary A.3. 2ND-BEST *is correct when multiple nodes are compromised.*

Proof. If multiple nodes, $\overline{V}$, are simultaneously compromised the proof is the same as that for Corollary A.2, substituting $\overline{V}$ for $\overline{v}$.

Next, we prove 2ND-BEST is correct in the case where a set of nodes, $\overline{V}_2$, are compromised concurrent with a running execution of 2ND-BEST (e.g., during $[t', t^*]$), triggered by the compromise of $\overline{V}$. First we show that any least cost computation (e.g., one triggered by $\overline{V}$'s compromise) to any $v \in \overline{V}_2$ is eventually terminated.

We have already proved that the diffusing computations to remove each $v \in \overline{V}_2$ as a destination complete in finite time. Let t_d mark the time these diffusing computations complete. For all $t \geq t_d$, any running least cost computation to a destination $v \in \overline{V}_2$ is terminated by the actions specified in Section 2.3.5. Therefore, the only remaining least cost computations are to all $v \in V'$, where $V' = V - \left(\overline{V} \cup \overline{V}_2\right)$. Because all $dmatrix_i$ values are non-negative for all $i \in V'$, by Theorem A.1 we conclude 2ND-BEST is correct.

Since we have proved 2ND-BEST is correct when multiple nodes are simultaneously compromised and when nodes are compromised concurrent with any 2ND-BEST execution, we conclude that 2ND-BEST is correct when multiple nodes are compromised. □

Corollary A.4. PURGE *is correct when a single node is compromised.*

Proof. Each $v \in adj(\overline{v})$ finds every destination, a, to which v's least cost path uses $\overline{v}$ as the first-hop node. v sets its least cost to each such a to ∞, thereby invalidating its path to a. v then initiates a diffusing computation. When an arbitrary node, i, receives a diffusing computation message from j, i iterates through each a specified in the message. If i routes via j to reach a, i sets its least cost to a to ∞, therefore invalidating any path to a with j and $\overline{v}$ an intermediate nodes.

By our assumptions, each node receives a diffusing computation message for each path using $\overline{v}$ as an intermediate node. Additionally, our assumptions imply that all diffusing computation terminate in finite time. Thus, we conclude that all paths using $\overline{v}$ as an intermediary node are invalidated in finite time.

Following the preprocessing, all $v \in adj(\overline{v})$ use distance vector to determine new least cost paths. Because all $dmatrix_i$ are non-negative for all $i \in V'$, by Theorem A.1 we conclude that PURGE is correct. □

Corollary A.5. PURGE *is correct when multiple nodes are compromised.*

Proof. The same proof used for Corollary A.3 applies for PURGE. □

Corollary A.6. CPR *is correct when a single node is compromised.*

Proof. CPR sets t' to the time $\overline{v}$ was compromised. Then, CPR rolls back using diffusing computations: each diffusing computation is initiated at each $v \in adj(\overline{v})$. Each node that receives a diffusing computation message, rolls back to a snapshot with timestep less than t'. By our assumptions, all nodes receive a message and the diffusing computation terminates in finite time. Thus, we conclude that each node $v \in V'$ rolls back to a snapshot with timestamp less than t' in finite time.

CPR then runs the preprocessing algorithm described in Section 2.3.1, which removes each $\overline{v}$ as a destination in finite time (as shown in Corollary A.2). Because each node rolls back to a snapshot in which all least costs are non-negative and CPR then uses distance vector to compute new least costs, by Theorem 1 we conclude that CPR is correct if no additional nodes are compromised during $[t', t^*]$. □

Corollary A.7. CPR *is correct when multiple nodes are compromised.*

Proof. If multiple nodes, $\overline{V}$ are simultaneously compromised, CPR sets t' to the time the first $\overline{v} \in \overline{V}$ is compromised. Any nodes, $\overline{V}_2$, compromised concurrent with $\overline{V}$ (e.g., during $[t', t^*]$), trigger an additional CPR execution. The steps described in Section 2.3.5 ensure that all least cost computations (after rolling back) are to destination nodes $a \in V'$. By Theorem A.1 we conclude CPR is correct because all $dmatrix_i$ are non-negative for all $i \in V'$. □

A.3 Analysis of Recovery Algorithms

In this section we first prove specific properties of our recovery algorithms (Section A.3.1) and then find communication complexity bounds for each recovery algorithm (Section A.3.2). These results were summarized in Section 2.4. All proofs assume a synchronous model in which nodes send and receive messages at fixed epochs. In

each epoch, a node receives a message from all its neighbors and performs its local computation. In the next epoch, the node sends a message (if needed). Before we begin with the analysis, we introduce additional notation used in our proofs.

Notation. We use the definition of G and G' described in Section 4.3. For convenience, $|V| = n$ and the diameter of G' is d. Let $\delta_t(i,j)$ be the least cost between nodes i and j – used by node i – at time t (we refer to this cost as $\delta(i,j)$). $p_t(i,j)$ refers to i's actual least cost path to j at time t. $p_s(i,j)$ is the least cost path from node i to j used by i at the start of recovery and $\delta_s(i,j)$ is the cost of this path; $p_w(i,j)$ is i's least cost path to j at time $t \in [t_b, t^*]$ and $\delta_w(i,j)$ the cost of this path [1]; and $p_f(i,j)$ is i's final least cost path to j (least cost at t^*) and has cost $\delta_f(i,j)$. $\ell(i,j)$ is the minimum number of links between nodes i and j in G'. Let $\max_{i \in V}(|adj(i)|) = m$.

For each algorithm, let $\hat{t}$ mark the time all diffusing computations complete. Recall with PURGE, $\overline{v}$ is removed as a destination and $\overrightarrow{bad}$ state is invalidated in the *same* diffusing computations. Likewise, each CPR diffusing computation performs two actions: the diffusing computations remove $\overline{v}$ as a destination *and* implement the rollback. For this reason, $\hat{t}$ marks the same time across all three recovery algorithms. Let $C(i,j) = \delta_f(i,j) - \delta_{\hat{t}}(i,j)$. That is, $C(i,j)$ refers to the magnitude of change in $\delta(i,j)$ after the diffusing computations for each algorithm complete.

A.3.1 Properties of Recovery Algorithms

In this section we formally characterize how $\overrightarrow{min}$ values change during recovery. The properties established in this section will aid in understanding the simulation results presented in Section 2.5. Our proofs assume that link weights remain fixed during recovery (i.e., during $[t', t_b]$). We prove properties about $\overrightarrow{min}$ in order provide a precise characterization of recovery trends. In particular, our proofs establish that:

[1] $p_w(i,j)$ and $\delta_w(i,j)$ can change during $[t_b, t^*]$.

- The least cost between two nodes at the start of recovery is less than or equal to the least cost when recovery has completed. (Theorem A.8)
- Before recovery begins, if the least cost between two nodes is less than its cost when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary A.9)
- During 2ND-BEST and CPR recovery, if the least cost between two nodes is less than its distance when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary A.10)

The first two statements apply to any recovery algorithm because they make no claims about $\overrightarrow{min}$ values during recovery.

Theorem A.8. $\forall i, j \in V'$, $\delta_s(i,j) \leq \delta_f(i,j)$

Proof. Assume $\exists i, j \in V'$ such that $\delta_s(i,j) > \delta_f(i,j)$. The paths available at the start of recovery are a superset of those available when recovery is complete. This means $p_f(i,j)$ is available before recovery begins. Distance vector would use this path rather than $p_s(i,j)$, implying that $\delta_s(i,j) = \delta_f(i,j)$, a contradiction. □

Corollary A.9. $\forall i, j \in V'$, *if* $\delta_s(i,j) < \delta_f(i,j)$, *then* $p_s(i,j)$ *is using* $\overrightarrow{bad}$ *or* $\overrightarrow{old}$ *either directly or transitively.*

Proof. $\exists i, j \in V$ such that a path $p_s(i,j)$ with cost $\delta_s(i,j)$ is used before recovery begins where $\delta_s(i,j) < \delta_f(i,j)$ and $p_s(i,j)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. The only paths available before recovery begins, which do not exist when recovery completes, are ones using $\overrightarrow{bad}$ or $\overrightarrow{old}$. Therefore, $p_s(i,j)$ must be available after recovery completes since we have assumed that $p_s(i,j)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. Distance vector would use $p_s(i,j)$ instead of $p_f(i,j)$ because $\delta_s(i,j) < \delta_f(i,j)$. However this would imply that $\delta_s(i,j) = \delta_f(i,j)$, a contradiction. □

Corollary A.10. *For* 2ND-BEST *and* CPR. $\forall i, j \in V'$, *if* $\delta_w(i,j) < \delta_f(i,j)$ *then* $p_w(i,j)$ *must be using* $\overrightarrow{bad}$ *or* $\overrightarrow{old}$ *either directly or transitively.* [2]

Proof. We can use the same proof for Corollary A.9 if we substitute $\delta_w(i,j)$ for $\delta_s(i,j)$ and $p_w(i,j)$ for $p_s(i,j)$. □

Corollary A.10 implies that 2ND-BEST and CPR (after rolling back), count up from their initial costs – using $\overrightarrow{bad}$ or $\overrightarrow{old}$ state – until reaching the final correct least cost.

A.3.2 Communication Complexity

Next, we derive communication complexity bounds for each recovery algorithm. First, we consider graphs where link weights remain fixed (Section A.3.2.1 - A.3.2.4). Then, we derive bounds where link weights can change (Section A.3.2.5).

We make the following assumptions in our complexity analysis:

- There is only a single compromised node, $\overline{v}$.
- We assume all nodes have unit link weight of 1 and that $\overline{v}$ falsely claims a cost of 1 to each $j \in V'$ (e.g., $\forall j \in V', \delta_s(\overline{v}, j) = 1$).
- Since we assume unit link weights of 1, a link weight increase correspond to the removal of a link and a link weight decrease corresponds to the addition of a link.

A.3.2.1 Diffusing Computation Analysis

We begin our complexity analysis with a study of the diffusing computations common to all three of our recovery algorithms: 2ND-BEST, CPR, and PURGE. In our analysis, we refer to a as our generic destination node.

---

[2] Corollary A.10 does not apply to PURGE recovery because the $\delta_w(i,j) < \delta_w(i,j)$ condition is not always satisfied.

Lemma A.11. *Each diffusing computation has $O(E)$ message complexity.*

Proof. Each node in a diffusing computation sends a query to all downstream nodes and a reply to its parent node. Thus, no more than 2 messages are sent across a single edge, yielding $O(E)$ message complexity. □

Theorem A.12. *The diffusing computations for* 2ND-BEST, CPR, *and* PURGE *have $O(mE)$ communication complexity.*

Proof. For each algorithm, diffusing computations are initiated at each $i \in adj(\overline{v})$, so there can be at most m diffusing computations. From Lemma A.11, each diffusing computation has $O(E)$ communication complexity, yielding $O(mE)$ communication complexity. □

A.3.2.2 2nd-Best Analysis

Johnson [45] studies DV over topologies with bidirectional links and unit link weights of 1. Specifically, Johnson analyzes DV update activity after the failure of a single network resource, in which a resource is either a node or a link. She assumes that nodes adjacent to a failed resource detect the failure and then react according to DV: in the case of a failed node, each node sets its distance to the failed node to n and no link connected to the failed node is used in the final correct shortest paths. [3] From this point, DV behaves exactly like 2ND-BEST. [4] Therefore, by mapping our false path problem to Johnson's failed resource problem, we can use Johnson's analysis of DV to find bounds (and exact message counts) for 2ND-BEST. To do so, we modify the graph, G, that Johnson considers by adding false paths between $\overline{v}$ and all other nodes.

[3] The maximum distance to any node under Johnson's model is n, where n is the number of nodes in the graph. This is equivalent to ∞ in our case.

[4] Note that in contrast to Johnson, we assume an outside algorithm identifies the compromised node.

In Corollary A.10, we proved that with 2ND-BEST nodes using $\overline{v}$ as an intermediate node count up from an initial incorrect least costs to their final correct value. Johnson proves the same for DV. Using this pattern, Theorem A.13 derives upper and lower bounds for 2ND-BEST. Intuitively, the lower bound occurs when nodes count up by 2 (to their final correct value) and the upper bound results when nodes count up by 1.

Theorem A.13. *After $\hat{t}$,* 2ND-BEST *message complexity is bounded below by*

$$\sum_{i \in V'} \left\lceil \frac{\max_{j \in V', i \neq j} (C(i,j))}{2} \right\rceil adj(i) \tag{A.1}$$

and above by

$$\sum_{i \in V'} \max_{j \in V', i \neq j} (C(i,j))\, adj(i) \tag{A.2}$$

Proof. Theorem 2 from [45] gives a lower bound of $\sum_{i,j \in V', i \neq j} \left\lceil \frac{1}{2} C(i,j) \right\rceil adj(i)$. However, this lower bound applies to a version of DV in which each message contains update costs for only a single destination; in a single epoch, if a node finds new least costs to multiple destinations, a separate message is sent for each destination with a new least cost (and is sent to each of the node's neighbors). In contrast, 2ND-BEST handles updates to multiple destinations concurrently: in each epoch, a single message sent by node i contains new distance values for all destinations in which i has a new least cost. For this reason, the maximum $C(i,j)$ value determines the number of times a node sends a message to each neighbor node.

The upper bound (Equation A.2) is also derived from Theorem 2 in [45]. Theorem 2 gives us a upper bound of $\sum_{i,j \in V', i \neq j} C(i,j) \cdot adj(i)$. For the same reason described for the lower bound, the maximum $C(i,j)$ value determines the number of times a node sends a message to each neighbor node. □

Corollary A.14. 2ND-BEST *has* $O(mnd)$ *communication complexity.*

Proof. From Lemma A.12, 2ND-BEST's diffusing computations have $O(mE)$ communication complexity. Next, 2ND-BEST runs DV. It must be the case that $C(i,j) \leq d$ and each node can at most have m neighbors. Since $|V'| = n - 1$, DV and therefore 2ND-BEST has $O(mnd)$ communication complexity. □

Next, we restate Theorem 1 from Johnson [45] using our notation. Theorem A.15 introduces the term *allowable path*. An allowable path from node i to $\overline{v}$ is a path in the original network (G) from node i to $\overline{v}$ which does not use $\overline{v}$ as an intermediate node.

Theorem A.15. *Each incorrect route table entry assumes all possible lengths of paths of the form* $|P| + \delta_s(\overline{v}, a)$ *where* P *is an allowable path from node* i *to* $\overline{v}$ *and* $\delta_s(\overline{v}, a)$ *is the length of the false path claimed by* $\overline{v}$.

Theorem A.15 translates the problem of finding the number of update messages after false node detection into the problem of finding all possible allowable paths between each node i and $\overline{v}$. By doing so, we can find the exact number of messages required for 2ND-BEST recovery.

The next two theorems, Theorem A.16 and A.17, follow from Theorem 5 in [45] and Theorem A.15.

Theorem A.16. *If* G *contains no odd cycles, the number of update messages after* $\hat{t}$ *is described exactly by Equation A.1.*

Define $S(p)$ to be the set of nodes such that if $i \in S(p)$ there exists an allowable path of length p and $p + 1$ from i to $\overline{v}$. Let $q(\overline{v}, i)$ be the smallest positive integer p such that $i \in S(p)$ and $q(\overline{v}, i) = c$.

Theorem A.17. *If G contains an odd cycle and $c+\delta_s(\overline{v},a) < \delta_f(i,a)$, then allowable paths to $\overline{v}$ increase in length by increments of 2 until reaching the value c and then increments by 1 thereafter. Thus, the number of changes in $\delta(i,a)$, after $\hat{t}$, is:*

$$C(i,a) - \frac{1}{2}\left(c - \delta_s(i,a)\right) \tag{A.3}$$

If $c + \delta_s(\overline{v},a) \geq \delta_f(i,a)$, then update activity ceases before node i's least cost entries begin to increase by 1. Thus, in this case the number of update messages, after $\hat{t}$, is described exactly by Equation A.1.

Theorem A.15 tells us that before converging on the correct distance to a destination, a, 2ND-BEST exhaustively searches all paths from i to $\overline{v}$ and then uses $\overline{v}$'s false path to a. If G contains no odd cycle, then i counts up by 2 until reaching the final correct cost to a. Node i does so by hopping back and forth between an adjacent node j (where $j \neq \overline{v}$) k times (for some integer $k \geq 0$), then uses an allowable path from i to $\overline{v}$, and finally uses $\overline{v}$'s false path to a.

However, if G contains an odd cycle then the update behavior is slightly more complicated. Node i counts up by 2 until $\delta(i,a)$ reaches a specific value, c^*, at which point, i counts up by 1 until i converges on the final correct distance to a. In Figure A.2, $c^* = \delta(i,h)+\delta(h,\overline{v})+\delta_s(\overline{v},a) = 1+(p-1)+1 = p+1$. In the epoch after $\delta(i,a)$ is set to c^*, node i uses its path via h of length p to $\overline{v}$ (and then $\overline{v}$'s false path to a). In the following epoch, i uses its path via l of length $p+1$ to $\overline{v}$. From this point, i counts up by 1 by using allowable paths of lengths $p+2k$, for integer $k \geq 1$, (by hopping back and forth between h) to $\overline{v}$ and allowable paths of length $(p+1)+2k$ (by ping-ponging with l) to $\overline{v}$, until $\delta(i,a)$ counts up to $\delta_f(i,j)$.

A.3.2.3 CPR Analysis

The analysis for 2ND-BEST applies to CPR because after rolling back CPR, executes the steps of 2ND-BEST. In fact, because CPR performs the rollback using the

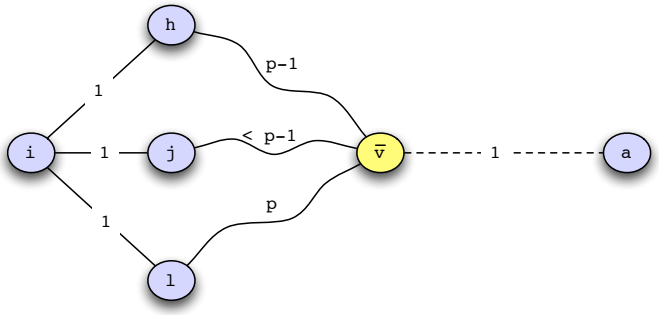


Figure A.2. The yellow node ($\overline{v}$) is the compromised node. The dotted line from $\overline{v}$ to a represents the false path.

same diffusing computations analyzed for 2ND-BEST (e.g., the diffusing computations that remove $\overline{v}$ as a destination), the results for 2ND-BEST apply to CPR with no changes.

Although Theorem A.13, Theorem A.16, and Theorem A.17 apply directly to CPR, the bounds and exact message count can defer between 2ND-BEST and CPR. In most cases, $\delta_{\hat{t}}(i,j)$ for 2ND-BEST is smaller than $\delta_{\hat{t}}(i,j)$ for CPR because CPR rolls back to a checkpoint taken before $\overline{v}$ is compromised. [5] Thus, CPR's $C(i,j)$ values are typically smaller than those for 2ND-BEST, resulting in lower message complexity for CPR.

A.3.2.4 Purge Analysis

Our PURGE analysis establishes that after the diffusing computations complete, all nodes using false routing state to reach a destination have a least cost of ∞ to this destination. From this point, these least costs remain ∞ until updates from nodes with a non-infinite cost to the destination spread through the network. Upon receiving a non-infinite least cost to the destination, nodes switch from an infinite

[5] At worst, $\delta_{\hat{t}}(i,j)$ is equivalent across 2ND-BEST and CPR. This occurs when the false least vector claimed by $\overline{v}$ matches the least cost vector used by $\overline{v}$ before being compromised (e.g., $\overrightarrow{bad}$=$\overrightarrow{old}$).

least cost to a finite one (Lemma A.18). We establish that the first finite cost to the destination is in fact the node's final correct least cost to the destination (Theorem A.20). In this way, least costs change from ∞ to their final correct value.

In the presence of a tie, we assume a node uses the least cost path that avoids $\overline{v}$. Note that if ties are broken by using the path with $\overline{v}$ as intermediate node, our proofs still apply, although with a few minor changes. Now we are ready to define two sets that are key structures in our PURGE proofs.

Definition 2. *Let $B(a,t)$ be the set of nodes that have least cost ∞ to destination node a at time t.*

Definition 3. *$F(a,t)$ is the set of nodes such that if $b \in F(a,t)$ then the following must be true:*

1. $b \notin B(a,t)$.
2. $\exists b' : b' \in adj(b) \wedge b' \notin B(a,t)$.
3. $\exists b'' : b'' \in adj(b) \wedge b'' \in B(a,t)$.

Next, in Lemma A.18 we prove that the size of $B(a,t)$ shrinks by at least one for each timestep beginning with t'' – where t'' refers to the time that the first $i \in V'$ with $\delta(i,a) = \infty$ changes $\delta(i,a)$ to a finite value – until $B(a,t)$ is empty.

Lemma A.18. *For each $t \geq t''$, $|B(a,t)| \geq |B(a,t+1)| + 1$, until $B(a,t) = \emptyset$.*

Proof. Once PURGE diffusing computations complete at $\hat{t}$, a DV computation is triggered at each $v \in adj(\overline{v})$. At this point, all least costs corresponding to paths using $\overline{v}$ as an intermediate node are set to ∞ (this is proved in Corollary A.4). As such, each $i \in B(a,\hat{t})$ sends a DV message with a least of ∞ to each neighbor, [6] unless i

[6]Recall that after $\hat{t}$, PURGE forces each node to send a least cost message to each neighbor (even if the node's least cost has not changed since $\hat{t}$).

has a neighbor node in $F(a, \hat{t})$ (note that we denote this time as t''). In this case, i selects a finite least to a (which implies $i \notin B(a, t'')$), triggering the propagation of finite least costs to a. Specifically, in each subsequent timestep t (until $B(a, t) = \emptyset$) at least one node, j, changes $\delta_t(j, a)$ from ∞ to a finite value. This is the case because unless $B(a, t) = \emptyset$, a node i that has changed $\delta_t(i, a)$ from ∞ to a finite value, has $j \in adj(i)$ with $\delta_t(j, a) = \infty$ and thus $\delta_{t+1}(j, a)$ will be finite. A finite $\delta_{t+1}(j, a)$ value implies $j \notin B(a, t + 1)$. Since $B(a, t)$ is monotonic, eventually $B(a, t) = \emptyset$. □

Our next Lemma (A.19) lists all possible values for the number of links between any $b \in F(a, \hat{t})$ and $\overline{v}$. We later use this Lemma in Theorem A.20.

Lemma A.19. *For all $b \in F(a, \hat{t})$, $\ell(b, \overline{v}) = \{\ell(b, a), \ell(b, a) - 1\}$.*

Proof. Let b be an arbitrary node in $F(a, \hat{t})$. If $\ell(b, \overline{v}) < \ell(b, a) - 1$, this would imply $b \in B(a, \hat{t})$, a contradiction (a violation of condition 1 of the $F(a, \hat{t})$ definition). On the other hand, consider the case where $\ell(b, \overline{v}) > \ell(b, a)$ and where $b' \in adj(b)$ and $b' \in B(a, \hat{t})$. Any path b' uses with $\overline{v}$ as an intermediate node has cost $\ell(b, \overline{v}) - 1 + \delta_s(\overline{v}, a) = \ell(b, \overline{v}) - 1 + 1 = \ell(b, \overline{v})$. Since we have assumed $\ell(b, \overline{v}) > \ell(b, a)$, b' would use b as a next-hop router along $p_{\hat{t}}(b', a)$. This implies $b' \notin B(a, \hat{t})$, a contradiction. □

The following theorem is the key argument in establishing PURGE's communication complexity. Theorem A.20 proves that once any $i \in V'$ changes its least cost from ∞, i changes its least cost to the final correct value.

Theorem A.20. *For $t > \hat{t}$ and an arbitrary destination $a \in V'$, each $i \in B(a, \hat{t})$ with $\delta_{\hat{t}}(i, a) = \infty$ only modifies $\delta(i, a)$ once, such that $\delta(i, a)$ changes from ∞ to $\delta_f(i, a)$.*

Proof. Consider an arbitrary $i \in V'$ such that $i \in B(a, \hat{t})$. i must use some $b \in F(a, \hat{t})$ as an intermediate node along $p_f(i, a)$. Let b^* be this node. If we show that $\delta_f(b^*, a)$ is the first least cost among all $b \in F(a, \hat{t})$ to reach i, then we have proved our claim because in Lemma A.18 we proved that i does not update its least cost to a finite value

until it receives a least cost from a $b \in F(a, \hat{t})$. [7] For the sake of contradiction, assume that for some $b' \in F(a, \hat{t})$, where $b' \neq b^*$, that $\delta_f(b', a)$ reaches i before $\delta_f(b^*, a)$. [8] This implies that:

$$\ell(b', \overline{v}) + \ell(i, b') \quad < \quad \ell(b^*, \overline{v}) + \ell(i, b^*) \tag{A.4}$$

From Lemma A.19, we know that $\ell(b', \overline{v}) = \{\ell(b', a), \ell(b', a) - 1\}$ and $\ell(b^*, \overline{v}) = \{\ell(b^*, a), \ell(b^*, a) - 1\}$ If we substitute $\ell(b', \overline{v}) = \ell(b', a)$ and $\ell(b^*, \overline{v}) = \ell(b^*, a)$ into Equation A.4, it yields:

$$\ell(b', a) + \ell(i, b') \quad < \quad \ell(b^*, a) + \ell(i, b^*) \tag{A.5}$$

However, since we have assumed that i routes via b^*, we know that:

$$\ell(b', a) + \ell(i, b') \quad > \quad \ell(b^*, a) + \ell(i, b^*) \tag{A.6}$$

Thus, between Equation A.5 and Equation A.6 we have a contradiction. Similar contradictions can be derived by substituting all other permutations of the $\ell(b', \overline{v})$ and $\ell(b^*, \overline{v})$ equalities, derived from Lemma A.19. In conclusion, we have shown by contradiction that $\delta(i, a)$ only changes a single time: $\delta(i, a)$ changes from ∞ to $\delta_f(i, a)$. □

Corollary A.21. PURGE *is loop-free at every instant of time.*

Proof. Before $\hat{t}$, only diffusing computation run. Diffusing computations are loop-free because computation proceeds along spanning trees, which are by definition acyclic.

---

[7] Note that any node i with $\delta(i, a) = \infty$ only changes $\delta(i, a)$ to a finite value. Thus, when PURGE forces nodes to send a message after $\hat{t}$ to initiate the DV computation, no $i \in B(a, \hat{t})$ receiving a least cost of ∞ updates its least cost.

[8] From Lemma A.18 we know that a finite least cost to a reaches every node in $B(a, \hat{t})$.

After $\hat{t}$, only DV computations run. From Theorem A.20 we know that each node with least cost ∞ to an arbitrary destination, changes its least cost once: from ∞ to the correct final least cost. We conclude that PURGE is loop free. □

Theorem A.22. PURGE *message complexity is* $O(mnd)$.

Proof. PURGE consists of two steps: the diffusing computations to invalidate false state and DV to compute new least cost paths invalidated by the diffusing computations. From Lemma A.12, PURGE's diffusing computations have $O(mE)$ communication complexity. The DV message complexity can be understood as follows. To start the computation, PURGE enforces that each node sends DV message (to each neighbor), even if no least costs are found. From Theorem A.20 and Lemma A.18, all $i \in B(a, \hat{t})$ only change $\delta(i, a)$ once: $\delta(i, a)$ changes from ∞ to $\delta_f(i, a)$. PURGE computations to all destinations run in parallel, meaning that all least cost updates to nodes h away are handled in the same round of update messages. For this reason, PURGE only sends messages $d + 1$ times after $\hat{t}$. Finally, since there are $n - 1$ nodes, each with a maximum of m neighbors, and each node sends messages $d + 1$ times, PURGE communication complexity if $O(mnd)$. □

A.3.2.5 Analysis that Considers Graphs with Link Weight Changes

In this section, we analyze each of our algorithms in the case where w link weight changes occur. Because we assume unit link weights of 1, a link weight decrease corresponds to the addition of a new link and a link weight increase corresponds to the removal of a link. In our analysis, we assume that all w link weight changes finish propagating before $\overline{v}$ is detected (e.g., before t_b).

The analysis for 2ND-BEST and PURGE from Section A.3.2.2 and Section A.3.2.4, respectively, does not change. This is the case because 2ND-BEST and PURGE do not roll back in time, and thus all w link weight changes are accounted for when recovery

begins at t_b. The CPR analysis from Section A.3.2.3 changes because after rolling back, all w link weight changes need to be replayed.

Let $\delta'_f(i,a)$ be node i's final least cost to a if no link weight changes occur during $[t', t_b]$. Define $C'(i,j) = \delta'_f(i,a) - \delta_i(i,j)$.

The communication complexity for a link weight increase is $O(n^2)$ [45] and $O(E)$ for a link weight decrease [44]. Let there be u link weight increases (e.g., u links are removed from G) and $w-u$ link weight decreases (e.g., $w-u$ links are added to G). At worst, the link weight changes are processed after $\overline{v}$ recovery completes. As a result, CPR communication complexity with link weight changes is bounded above by:

$$\sum_{i \in V'} \max_{j \in V', i \neq j} \left(C'(i,j)\right) adj(i) + O(un^2) + O\left((w-u)E\right) \tag{A.7}$$

A.3.2.6 Discussion

The communication complexity for 2ND-BEST, CPR, and PURGE are all $O(mnd)$ over graphs with fixed unit link weights. It is not surprising that the communication complexity is the same because all three algorithms use DV as their final step and DV asymptotically dominates the communication complexity of each recovery algorithm. Thus, the difference in message complexity between the three algorithms, found in our simulations, amounts to marginal differences in each algorithm's hidden constant in the stated message complexity bound.

We also bounded the communication overhead incurred by CPR under conditions of link weight changes. This overhead is not incurred by 2ND-BEST and PURGE because do not roll back in time, and thus all link weight changes are accounted for when recovery begins.

APPENDIX B

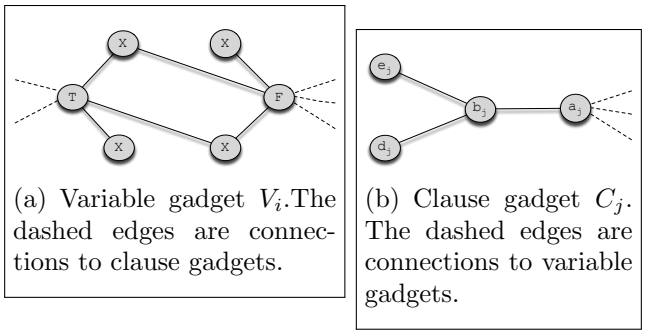
ADDITIONAL PMU PLACEMENT PROBLEM PROOFS

In Chapter 3 we proved that FullObserve (Section 3.3.2), MaxObserve (Section 3.3.3), FullObserve-XV (Section 3.3.4), and MaxObserve-XV (Section 3.3.5) are each NP-Complete when considering networks with both zero-injection and injection buses. Here we prove that each problem is also NP-Complete for graphs containing only zero-injection nodes. The proofs closely resemble those in Sections 3.3.2 - 3.3.5. Then, we provide pseudo-code and complexity proofs for the approximation algorithms described in Section 3.4. These proofs consider graphs with both zero-injection and injection buses.

B.1 NP-Completeness Proofs for PMU Placement in Zero-Injection Graphs

In the following order MaxObserve (Section B.1.1), FullObserve-XV (Section B.1.2), and MaxObserve-XV (Section B.1.3), we prove that each problem is NP-Complete for graphs containing only zero-injection nodes. Our proofs below do not explicitly mention our assumption that all nodes are zero-injection; rather, this assumption is implicit in the fact that we apply observability rule 2 whenever possible. We omit a new proof for FullObserve because Brueni and Heath [14] prove FullObserve is NP-Complete for zero-injection graphs.

Our proofs follow the same strategy outlined in Section 3.3.1: we reduce for P3SAT to show each problem is NP-Complete. Recall that our proofs from Chapter 3 relied on the definition of a bipartite graph $G(\phi) = (V(\phi), E(\phi))$ where ϕ is a 3-SAT



(a) Variable gadget V_i.The dashed edges are connections to clause gadgets.

(b) Clause gadget C_j. The dashed edges are connections to variable gadgets.

Figure B.1. Gadgets used in Theorem B.1 proof.

formula with variables $\{v_1, v_2, \ldots, v_r\}$ and clauses $\{c_1, c_2, \ldots, c_s\}$. $G(\phi)$'s vertices and edges were defined as follows:

$$\begin{aligned} V(\phi) &= \{v_i \mid 1 \leq i \leq r\} \cup \{c_j \mid 1 \leq j \leq s\} \\ E(\phi) &= \{(v_i, c_j) \mid v_i \in c_j \ \ or \ \ \overline{v_i} \in c_j\}. \end{aligned}$$

B.1.1 MaxObserve Problem for Zero-Injection Graphs

A description of MaxObserve can be found in Section 3.3.3. Our proof for the theorem below (Theorem B.1) is similar to that for Theorem 3.4.

Theorem B.1. MaxObserve *is NP-Complete when considering graphs with only zero-injection nodes.*

Proof idea: First, we construct problem-specific gadgets for variables and clauses. We then demonstrate that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. Next we show that as a result of this, the problem of observing m nodes in this graph reduces to the NP-complete problem presented in [14], which concludes our proof.

Proof. We start by arguing that MaxObserve $\in \mathcal{NP}$. First, nondeterministically select k nodes in which to place PMUs. Then we use the rules specified in Section 3.2.2 to determine the number of observed nodes.

We reduce from P3SAT, where ϕ is an arbitrary P3SAT formula, to show MaxObserve is NP-hard. Specifically, given a graph $G(\phi)$ we construct a new graph $H_1(\phi) = (V_1(\phi), E_1(\phi))$ by replacing each variable (clause) node in $G(\phi)$ with the variable (clause) gadget shown in Figure B.1(a) (B.1(b)). The edges connecting clause gadgets with variable gadgets express which variables are in each clause: for each clause gadget C_j, node a_j is attached to node T in variable gadget V_i if, in ϕ, v_i is in c_j, and to node F if $\overline{v_i}$ is in c_j. For convenience, we let $G = H_1(\phi)$.

With this construct in place, we move on to our proof. Here we consider the case of $k = r$ and $m = 6r + 2s$, and show that ϕ is satisfiable if and only if $r = |\Phi_G|$ PMUs can be placed on G such that $m \leq |\Phi_G^R| < |V|$. We will later discuss how to extend this proof for any larger value of m.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. Then, consider the placement Φ_G s.t. for each variable gadget V_i, $T_i \in \Phi_G \Leftrightarrow v_i = True$ in A_ϕ, and $F_i \in \Phi_G \Leftrightarrow v_i = False$. It has been shown in [14] that for $H(\phi)$ this placement observes all $H(\phi)$, and it can be easily verified that all nodes in $H_1(\phi)$ are observed as well except for d_j, e_j for each C_j. This amounts to $2s$ nodes, so exactly m nodes are observed by Φ_G, as required.

($\Leftarrow$) We begin by proving that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. Assume that there are $1 < t \leq r$ variable gadgets without a PMU. Then, at most t PMUs are on nodes in clause gadgets, so *at least* $\max(s - t, 0)$ clause gadgets are without PMUs. We want to show here that for $m = 6r + 2s$, $t = 0$.

To prove this, we rely on the following two simple observations:

- In any variable gadget V_i, nodes X (Figure B.1(a)) cannot be observed unless there is a PMU somewhere in V_i. Note that there are 4 such nodes per V_i.
- In any clause gadget C_j, nodes e_j and d_j cannot be observed unless there is a PMU somewhere in C_j. Note that there are 2 such nodes per C_j.

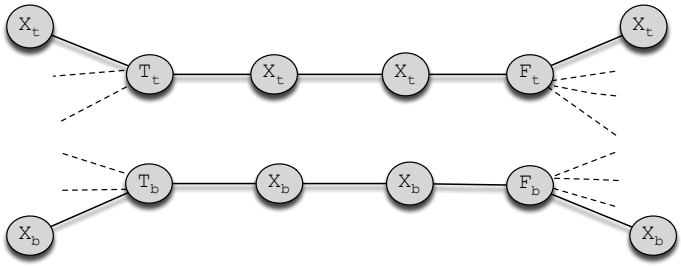


Figure B.2. Variable gadget used in Theorem B.2 proof. The dashed edges are connections to clause gadgets.

Thus, given some t, the number of unobserved nodes is *at least* $4t + \max(2(s-t), 0)$. However, since $|V| - m \leq 2s$, there are *at most* $2s$ unobserved nodes. So we get $2s \geq 4t + \max(2(s-t), 0)$. We consider two cases:

- $s \geq t$: then we get $2s \geq 2s + 2t \Rightarrow t = 0$.
- $s < t$: then we get $2s \geq 4t \Rightarrow s \geq 2t$, and since we assume here $0 \leq s < t$ this leads to a contradiction and so this case cannot occur.

Thus, we have concluded that the r PMUs must be on nodes in variable gadgets, all of which, it is important to note, were also part of the original $H(\phi)$ graph. We return to this point shortly.

We now observe that for each clause gadget C_j, such a placement of PMUs cannot observe nodes of type e_j, d_j, which amounts to a total of $2s$ unobserved nodes - the allowable bound. This means that all other nodes in G must be observed. Specifically, this is exactly all the nodes in the original $H(\phi)$ graph, and PMUs can only be placed on variable gadgets, all of which are included in $H(\phi)$ as well. Thus, the problem reduces to the problem in [14]. We use the proof in [14] to determine that all clauses in ϕ are satisfied by the truth assignment derived from Φ_G. □

B.1.2 FullObserve-XV Problem for Zero-Injection Graphs

The problem statement for FullObserve-XV can be found in Section 3.3.4. The proof for Theorem B.2, below, closely follows the structure of Theorem 3.5's proof.

Theorem B.2. FullObserve-XV *is NP-Complete when considering graphs with only zero-injection nodes.*

Proof. First, we argue that FullObserve-XV $\in \mathcal{NP}$. Given a FullObserve-XV solution, we use the polynomial time algorithm described in our proof for Theorem B.1 to determine if all nodes are observed. Then, for each PMU node we run a breadth-first search, stopping at depth 2, to check that the cross-validation rules are satisfied.

To show FullObserve-XV is NP-hard, we reduce from P3SAT. Our reduction is similar to the one used in Theorem B.1. For this problem, we use different variable and clause gadgets. The clause gadgets consist of the edge (a_j, b_j) from Figure B.1(b), which are the same as used in [14]. The new variable gadget is shown in Figure B.2. As can be seen in this figure, the variable gadgets are comprised of two disconnected subgraphs: we refer to the upper subgraph as V_{it} and the lower subgraph as V_{ib}. Clause gadgets are connected to a variable gadgets in the following manner: for each clause c_j that contains variable v_i in ϕ, the corresponding clause gadget has the edges $(a_j, T_t), (a_j, T_b)$, and for each clause c_j that contains variable $\overline{v_i}$ in ϕ, the corresponding clause gadget has the edges $(a_j, F_t), (a_j, F_b)$. We denote the resulting graph as $H_2(\phi)$, and for what follows assume $G = H_2(\phi)$.

We now show that ϕ is satisfiable if and only if $k = 2r$ PMUs can be placed on G such that G is fully observed under the condition that all PMUs are cross-validated, and that $2r$ PMUs are the minimal bound for observing the graph with cross-validation.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. For each $1 \leq i \leq r$, if $v_i = True$ in A_ϕ we place a PMU at T_b and at T_t of the variable gadget V_i. Otherwise,

we place a PMU at F_b and at F_t of this gadget. In either case, the PMU nodes in V_i must be adjacent to a clause node, making T_t (F_t) two hops away from T_b (F_b). Therefore, all PMUs are cross-validated by XV2.

Now we argue that Φ_G observes all $v \in V$:

- Consider a clause node a_j. Since ϕ is satisfied, for some index i we have $v_i \in c_j \wedge v_i \in A_\phi$ or $\overline{v_i} \in c_j \wedge \overline{v_i} \in A_\phi$. For the first case, the PMUs in V_i are placed on $\{T_b, T_t\}$ and as a result a_j is observed by applying O1 at T_b or at T_t. A similar argument applies for the second case. So, all a_j nodes are observed.

- Next, consider the nodes on the variable gadgets. When $v_i \in A_\phi$, T_t's neighbors, in V_{it}, are observed via O1. (the second case, $\overline{v_i} \in A_\phi$, follows by symmetry). The remaining V_{it} nodes are observed via O2 - note that if F_t is connected to a clause gadget we know from the previous step this clause is observed. By symmetry of V_{ib} and V_{it}, the same argument can be made for V_{ib} to show all V_{ib} nodes are observed.

- Finally, all the neighbors of a_j in variable gadgets are observed, and a_j is observed, so we can now apply O2 at each node a_j to observe the remaining b_j nodes.

This completes this direction of the theorem.

($\Leftarrow$) Suppose Φ_G observes all nodes in G under the condition that each PMU is cross-validated, and that $|\Phi_G| = 2r$. We want to show that ϕ is satisfiable by the truth assignment derived from Φ_G. We prove this by showing that (a) each variable gadget must have exactly 2 PMUs and (b) there must be a PMU at each subgraph of the variable gadget. Once (b) is shown, (c) cross-validation restrictions force the PMUs to be either on both T-nodes or both F-nodes. We conclude by showing that (d) the PMU nodes correspond to true/false assignments to variables which satisfy ϕ.

We begin by showing that each variable gadget must have 2 PMUs. Let V_i be a variable gadget with less than two PMUs. By placing PMUs on clause gadgets attached to V_i, at most we can observe T_t, T_b, F_t and F_b directly from the clause gadgets. Next, at least one of the V_i subgraphs has no PMU: without loss of generality, let this be V_{it}. We cannot apply O1 at T_t or F_t, since they have no PMU. We cannot apply O2 at these nodes since they each have two unobserved X_t nodes. Thus, all X_t nodes are unobserved in V_{it}, contrary to our assumption that the entire graph is observed. Thus we have shown that there must be at least 2 PMUs at each variable gadget. Also it is clear from this proof that, in fact, there must be at least one PMU in each subgraph of each variable gadget. Finally, since there are $2r$ PMUs and r variables, we conclude that each variable gadget has exactly two PMUs – one PMU for each variable gadget subgraph – and there are no PMUs on clause nodes.

Due to the cross-validation constraint, it is clear that a PMU on V_{it} can only be cross-validated by a PMU on V_{ib} (since all other variable-gadgets are more than 2 hops away), and specifically this would require both to be either on $\{T_t, T_b\}$ or $\{F_t, F_b\}$.

Without loss of generality, assume for an arbitrary variable gadget, V_i, we placed the PMUs at $\{T_t, T_b\}$. By applying O1 and O2, this placement can observe all nodes in the variable gadget if $\{F_t, F_b\}$ in this gadget are not adjacent to a clause node. If they are adjacent to some a_h node, each of $\{F_t, F_b\}$ can observe its adjacent leaf-X-node only via O2, and only if a_h is already observed. Since we are given a PMU placement that observes the entire graph, this implies that a_h is indeed observed and thus adjacent to some variable node with a PMU, such that O1 could be applied to view a_h. Assume without loss of generality, a_h is adjacent to PMU nodes T_b, T_t from variable gadget V_l, then the clause $c_h \in \phi$ is satisfied if v_l is true. A similar argument can be made if V_l is adjacent to PMU nodes F_t, F_b. We conclude that all clauses in ϕ are satisfied by the truth assignment derived from Φ_G. □

B.1.3 MaxObserve-XV Problem for Zero-Injection Graphs

The MaxObserve-XV problem is described in Section 3.3.5. The proof below for Theorem B.3 closely resembles the proof for Theorem 3.7.

Theorem B.3. MaxObserve-XV *is NP-Complete when considering graphs with only zero-injection nodes.*

Proof Idea: We show MaxObserve-XV is NP-hard by reducing from P3SAT. Our proof is a combination of the NP-hardness proofs for MaxObserve and FullObserve-XV. From a P3SAT formula, ϕ, we create a graph $G = (V, E)$ with the clause gadgets from MaxObserve (Figure B.1(b)) and the variable gadgets from FullObserve-XV (Figure B.2). The edges in G are identical the ones the graph created in our reduction for FullObserve-XV.

We show that any solution that observes $m = |V| - 2s$ nodes must place the PMUs exclusively on nodes in the variable gadgets. As a result, we show 2 nodes in each clause gadget – e_j and d_j for clause C_j – are not observed, yielding a total $2s$ unobserved nodes. This implies all other nodes must be observed, and thus reduces our problem to the scenario considered in Theorem B.2, which is already proven.

Proof. MaxObserve-XV is easily in $\mathcal{NP}$. We verify a MaxObserve-XV solution using the same polynomial time algorithm described in our proof for Theorem B.2.

We reduce from P3SAT to show MaxObserve-XV is NP-hard. Our reduction is a combination of the reductions used for MaxObserve and FullObserve-XV. Given a P3SAT formula, ϕ, with variables $\{v_1, v_2, \ldots, v_r\}$ and the set of clauses $\{c_1, c_2, \ldots, c_s\}$, we form a new graph, $H_3(\phi) = (V(\phi), E(\phi))$ as follows. Each clause c_j corresponds to the clause gadget from MaxObserve (Figure B.1(b)) and the variable gadgets from FullObserve-XV (Figure 3.3(c)). As in Theorem B.2, we refer to the upper subgraph of variable gadget, V_i, as V_{it} and the lower subgraph as V_{ib}. Also, we let $H_3(\phi) = G = (V, E)$.

Let $k = 2r$ and $m = 12r + 2s = |V| - 2s$. As in our NP-hardness proof for MaxObserve, m includes all nodes in G except d_j, e_j of each clause gadget. We need to show that ϕ is satisfiable if and only if $2r$ cross-validated PMUs can be placed on G such that $m \leq |\Phi_G^R| < |V|$.

($\Rightarrow$) Assume ϕ is satisfiable by truth assignment A_ϕ. For each $1 \leq i \leq r$, if $v_i = True$ in A_ϕ we place a PMU at T_b and at T_t of the variable gadget V_i. Otherwise, we place a PMU at F_b and at F_t of this gadget. In either case, the PMU nodes in V_i must be adjacent to a clause node, making T_t (F_t) two hops away from T_b (F_b). Therefore, all PMUs are cross-validated by XV2.

This placement of $2r$ PMUs, Φ_G, is exactly the same one derived from ϕ's satisfying instance in Theorem B.2. Since Φ_G only has PMUs on variable gadgets, all a_j and b_j nodes are observed by the same argument used in Theorem B.2. Thus, at least $12r + 2s$ nodes are observed in G. Because no PMU in Φ_G is placed on a clause gadget, C_j, we know that all e_j and d_j are not observed. We conclude that exactly m nodes are observed using Φ_G.

($\Leftarrow$) We begin by proving that any solution that observes m nodes must place the PMUs only on nodes in the variable gadgets. Assume that there are $1 < t \leq r$ variable gadgets without a PMU. Then, at most t PMUs are on nodes in clause gadgets, so *at least* $\max(s - t, 0)$ clause gadgets are without PMUs. We want to show here that for $m = 12r + 2s$, $t = 0$.

To prove this, we rely on the following observations:

- As shown in Theorem B.2, a variable gadget's subgraph with no PMU has at least 4 unobserved nodes.

- In any clause gadget C_j, nodes e_j and d_j cannot be observed if there is no PMU somewhere in C_j. Note that there are 2 such nodes.

Thus, given some t, the number of unobserved nodes is *at least* $4t + \max(2(s-t), 0)$. However, since $|V| - m \leq 2s$, there are *at most* $2s$ unobserved nodes. So we get $2s \geq 4t + \max(2(s-t), 0)$. We consider two cases:

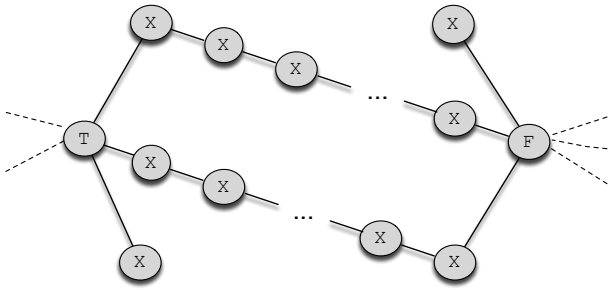
- $s \geq t$: then we get $2s \geq 2s + 2t \Rightarrow t = 0$.
- $s < t$: then we get $2s \geq 4t \Rightarrow s \geq 2t$, and since we assume here $0 \leq s < t$ this leads to a contradiction and so this case cannot occur.

Thus, we have concluded that the $2r$ PMUs must be on variable gadget. We now observe that for each clause gadget C_j, such a placement of PMUs cannot observe nodes of type e_j, d_j, which amounts to a total of $2s$ unobserved nodes - the allowable bound. This means that all other nodes in G must be observed. Specifically this is exactly all the nodes in $H_2(\phi)$ from the Theorem B.2 proof, and PMUs can only be placed on variable gadgets, all of which are included $H_2(\phi)$ from the Theorem B.2 proof. Thus, the problem reduces to the problem in Theorem B.2 and so we the Theorem B.2 proof to determine that all clauses in ϕ are satisfied by the truth assignment derived from Φ_G. □

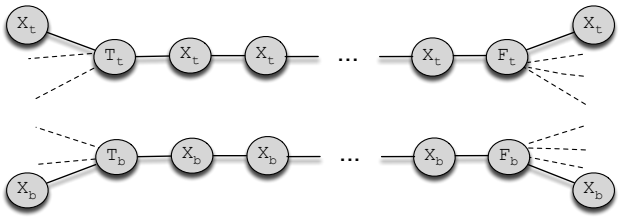
B.1.4 Extending Gadgets to Cover a Range of m and $|V|$ values

In the MaxObserve-XV and MaxObserve proofs we demonstrated NP-completeness for $m = |V| - 2s$. We show that slight adjustments to the variable and clause gadgets can yield a much wider range of m and $|V|$ values. We present the outline for new gadget constructions and leave the detailed analysis to the reader.

To increase the size of m (e.g., the number of observed nodes), we simply add more X nodes between the T and F nodes in the variable gadgets used in our proofs for MaxObserve-XV and MaxObserve. The new variable gadgets for MaxObserve and MaxObserve-XV are shown in Figure B.3(a) and Figure B.3(b), respectively. The same PMU placement described in the NP-Completeness proofs for each problem observes these newly introduced nodes.



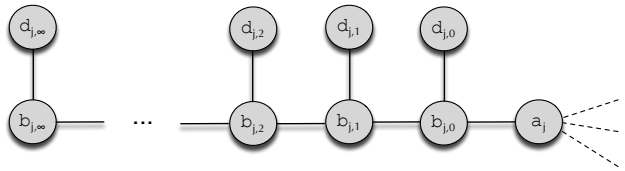
(a) Extended variable gadget used for MaxObserve.



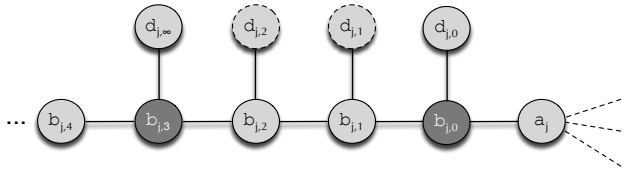
(b) Extended variable gadget used for MaxObserve-XV.

Figure B.3. Figures for variable gadget extensions described in Section B.1.4. The dashed edges indicate connections to clause gadget nodes.

In order to increase the size of $|V|$ while keeping m the same, we replace each clause gadget, C_j for $1 \leq j \leq s$, with a new clause gadget, C'_j, shown in Figure B.4(a). For MaxObserve, the optimal placement of PMUs on C'_j is to place PMUs on every fourth $b_{j,h}$ node, as shown in Figure B.4(b). As a result, the optimal placement of l PMUs on C'_j can at most observe $6l$ nodes. By adding $6l$ T nodes to each variable gadget, more nodes are always observed by placing a PMU on the variable gadget rather than at a clause gadget. We can use this to argue that PMUs are only placed on variable gadgets and then leverage the argument from Theorem B.1 to show MaxObserve is NP-Complete for any $\frac{m}{|V|}$. A similar argument can be made for MaxObserve-XV.



(a) Extended clause gadget C_j.



(b) Observed nodes in extended clause gadget C_j shown in (a). PMU nodes are dark gray, nodes observed by O2 have a dashed border, and all other nodes are observed by O1.

Figure B.4. Figures for clause gadget extensions described in Section B.1.4. The dashed edges indicate connections to variable gadget nodes.

B.2 Approximation Algorithm Complexity Proofs

In Section 3.4 we presented two greedy approximation algorithms, `greedy` and `xvgreedy`, that iteratively add a PMU in each step to the node that observes the maximum number of new nodes. Here the pseudo-code for each algorithm is specified and we prove that each algorithm has polynomial time complexity. We emphasize that these algorithms, unlike the problems discussed in the previous section, make no assumptions that nodes must be zero-injection.

The pseudo-code for `greedy` and `xvgreedy` can be found in Algorithm B.2.1 and Algorithm B.2.2, respectively.

Theorem B.4. *For input graph $G = (V, E)$ and k PMUs* `greedy` *has $O(dkn^3)$ complexity, where $n = |V|$ and d is the maximum degree node in V.*

Proof. The procedure to determine the number of nodes observed by a candidate PMU placement spans steps $6 - 18$. [1] First, we apply O1 at each PMU node (steps

[1] In this proof, step i refers to the i^{th} line in Algorithm B.2.1.

Algorithm B.2.1: `greedy` with input $G = (V, E)$ and k PMUs

```
1: Φ_G ← ∅
2: for k iterations do
3:    maxObserved ← 0
4:    for each v ∈ (V − Φ_G) do
5:       numObserved ← 0
6:       for each u ∈ (Φ_G ∪ {v}) do
7:          add PMU to u
8:          apply O1 at u and update numObserved
9:       end for
10:      repeat
11:         flag ← False
12:         for each w ∈ (V − (Φ_G ∪ {v})) do
13:            if w ∈ (V_Z ∩ Φ_G^R) and w has 1 unobserved neighbor then
14:               apply O2 at w and update numObserved
15:               flag ← True
16:            end if
17:         end for
18:      until flag = False
19:      if numObserved > maxObserved then
20:         greedyNode ← v
21:         maxObserved ← numObserved
22:      end if
23:   end for
24:   Φ_G ← Φ_G ∪ {greedyNode}
25: end for
```

$6-9$). O1 takes $O(d)$ time to be applied at a single node. Because $|\Phi_G| \leq k$, the total time to apply O1 is $O(dk)$.

Then, we iteratively apply O2 (steps $10-18$), terminating when no new nodes are observed. Like O1, applying O2 at a single node takes $O(d)$ time. In each iteration, if possible we apply O2 at each $v \in (V_Z \cap \Phi_G^R)$ (steps $13-16$). It total, the *loop* spanning steps $10-18$ repeats at most $O(n)$ times. This occurs when only a single new node is observed in each iteration. The for loop spanning steps $12-17$ repeats $O(n)$ times. We conclude that O2 evaluation for each set of candidate PMU locations takes $O(dn^2)$ time.

In order to determine the placement of each PMU, we try all possible PMU placements among nodes without a PMU. We place the PMU at the node that observes the maximum number of new nodes. This corresponds to Steps $4-23$, in which the for loop iterates $O(n)$ times. Thus the complexity of Steps $4-23$ is $O(dn^3)$.

Finally, the outer most for loop (Steps $2-25$) iterates k times: one iteration to determine the greedy placement of each PMU. We conclude that the complexity of `greedy` is $O(dkn^3)$. □

Theorem B.5. *For input graph $G = (V, E)$ and k PMUs* `xvgreedy` *has $O(kdn^3)$ complexity, where $n = |V|$ and d is the maximum degree node in V.*

Proof. The only difference between `xvgreedy` and `greedy` is that `xvgreedy` only considers pairs of cross-validated nodes. For this reason, step 4 in Algorithm B.2.2 does not appear in Algorithm B.2.1. We can find all pairs of cross-validated nodes in $O(d^2n)$ time. We do so by implementing a breadth-first search at each $v \in (V - \Phi_G)$ but stopping at a depth of 2. This takes $O(d^2)$ time for each node and since $O(n)$ searches are executed, step 4 takes $O(d^2n)$ time.

Because all other parts of Algorithm B.2.1 and Algorithm B.2.2 are nearly identical – Algorithm B.2.2 adds PMUs in pairs while Algorithm B.2.1 adds PMUs one-at-

Algorithm B.2.2: `xvgreedy` with input $G = (V, E)$ and k PMUs

$\Phi_G \leftarrow \emptyset$
for k iterations **do**
  $maxObserved \leftarrow 0$
  $C \leftarrow$ all cross-validated node pairs in $(V - \Phi_G)$
  for each $\{v_1, v_2\} \in C$ **do**
    $numObserved \leftarrow 0$
    for each $u \in (\Phi_G \cup \{v_1, v_2\})$ **do**
      add PMU to v_1 and v_2
      apply O1 at u and update $numObserved$
    end for
    repeat
      $flag \leftarrow False$
      for each $w \in (V - (\Phi_G \cup \{v_1, v_2\}))$ **do**
        if $w \in (V_Z \cap \Phi_G^R)$ and w has 1 unobserved neighbor **then**
          apply O2 at w and update $numObserved$
          $flag \leftarrow True$
        end if
      end for
    until $flag = False$
    if $numObserved > maxObserved$ **then**
      $greedyNodes \leftarrow \{v_1, v_2\}$
      $maxObserved \leftarrow numObserved$
    end if
  end for
  $\Phi_G \leftarrow \Phi_G \cup greedyNodes$
end for

a-time – we are able to directly apply the analysis from Theorem 3.8 in this proof. Therefore, we conclude the complexity of `xvgreedy` is $O(k(d^2n+dn^3)) = O(dkn^3)$. $\square$

APPENDIX C

COMPLEXITY OF MULTICAST RECYCLING PROBLEM

C.1 NP-hardness Proof for Multicast Recycling

In this section, we first define a simplified version of MULTICAST RECYCLING, 1MULTICAST RECYCLING, that considers a *single* primary tree and its backup tree for a given link. Then, we prove 1MULTICAST RECYCLING is NP-hard and use this result to bound MULTICAST RECYCLING's complexity. To enhance readability, we drop the subscript from T_i^l, $\hat{T}_i^l$, SA_i, and C_i^l when defining 1MULTICAST RECYCLING.

The 1MULTICAST RECYCLING decision problem is defined as follows:

- Instance: (G, k, T^l, l, α). $G = (V, E)$ is a directed, connected graph. k is an integer greater than 0 and $\alpha \geq 1$. $T^l = (V^l, E^l, r, D)$ is a primary tree containing nodes V^l, edges E^l such that $l \in E^l$, has root r, and spans $D = \{d_1, d_2, ..., d_m\}$.
- Question: Is there backup tree $\hat{T}^l = (\hat{V}^l, \hat{E}^l, r, D)$ such that $l \notin \hat{E}^l$ and $C^l \leq k$ under the condition that $w(\hat{T}^l) \leq \alpha \cdot w(SA(G'))$ where $G' = (V', E')$ such that $E' = E - \{l\}$; $SA(G') = (V_A, E_A, r, D)$ is a Steiner arborescence with root r and spans D; and $w(\hat{T}_i^l)$ is the sum of $\hat{T}_i^l$'s link weights?

Intuitively, we prove that 1MULTICAST RECYCLING is NP-hard by showing that in some cases an optimal solution to 1MULTICAST RECYCLING requires a solution to STEINER-ARBORESCENCE, a problem known to be NP-hard.

Theorem C.1. 1MULTICAST RECYCLING *is NP-hard.*

Proof. We reduce from the STEINER-ARBORESCENCE problem to show 1MULTICAST RECYCLING is NP-hard. Let the input to STEINER-ARBORESCENCE be a directed

graph $G = (V, E)$ with unit link weights of 1, root node $r \in V$, and a set of terminal nodes $S \subseteq V$. For integer $q > 0$, Steiner-Arborescence determines if an arborescence exists such that the sum of its link weights is less than or equal to q.

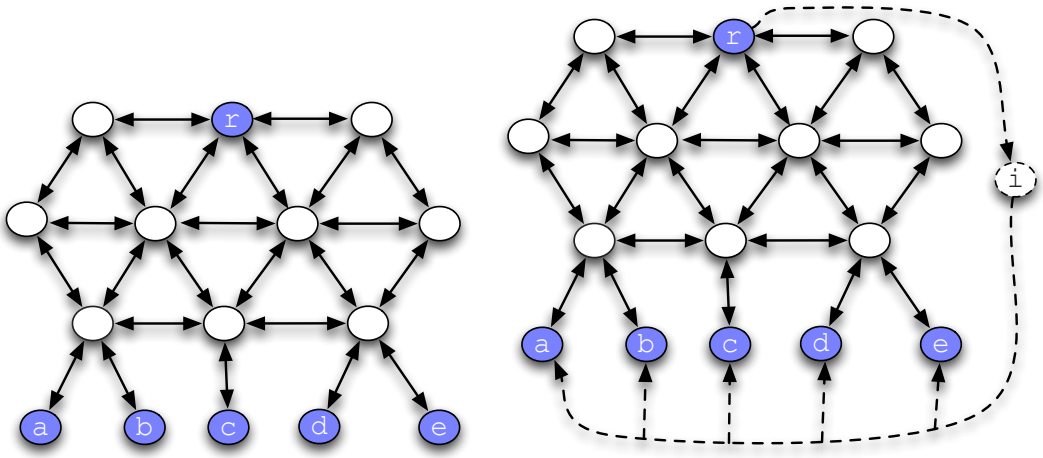
We perform our reduction as follows. We keep r as the root node and set the terminal nodes $D = S$. We make a copy of G, G^+, and add a new node, i, to G^+ with edges (r, i) and for all $d_j \in D$, (i, d_j). We let T^l be the tree rooted at r, formed by the addition of edges (r, i) and for all $d_j \in D$, (i, d_j). Finally, we set $l = (r, i)$, $k = q$, and and $\alpha = 1$. Figure C.1 shows an example of this reduction procedure.

With this construction in place, we move on to show that a Steiner arborescence rooted at r that spans S with cost less than or equal to q exists if and only if a backup tree $\hat{T}^l = (\hat{V}^l, \hat{E}^l, r, D)$ exists in G^+ such that $l \notin \hat{E}^l$ and $C^l \leq k$ under the condition that $w(\hat{T}^l) \leq \alpha \cdot w(SA(G^+))$ In the remainder of this proof, we refer to $SA(G^+)$ simply as SA.

($\Rightarrow$) Given a Steiner arborescence in G, A, rooted at r, spanning S, and with cost q, we set $\hat{T}^l = A$. By construction $\hat{T}^l$ has root r, spans D, does not use l, results in $C^l \leq k$, and trivially satisfies $w(\hat{T}^l) \leq \alpha \cdot w(SA)$.

($\Leftarrow$) In the other direction, suppose we have a solution to 1Multicast Recycling, a backup tree $\hat{T}^l = (\hat{V}^l, \hat{E}^l, r, D)$ in G^+ such that $l \notin \hat{E}^l$, $C^l \leq k$, and $w(\hat{T}^l) \leq \alpha \cdot w(SA)$. Since we have fixed l to be (r, i) and G^+ is constructed such that the only incoming edge to i is (r, i), T^l must be the tree formed by the edges (r, i) and $\forall_{d_j \in D}(i, d_j)$. Because $l \notin \hat{E}^l$, by construction, it must be the case that $\hat{T}^l$ is edge disjoint from T^l. Since we have assumed that $C^l \leq k$, it must be the case that $\hat{T}^l$ has at most $k = q$ edges. Lastly, we know that all nodes and edges in $\hat{T}^l$ exists in G, meaning that $\hat{T}^l$ forms a Steiner arborescence in G rooted at r, spanning S, and has cost q. This concludes our proof. □

Theorem C.2. Multicast Recycling *is at least NP-hard.*



(a) Graph $G = (V, E)$ where $S = \{a, b, c, d, e\}$ and r are shaded blue.

(b) Graph $G^+ = (V^+, E^+)$ resulting from our reduction in Theorem 4.1. All nodes (i) and edges (i's incoming and outgoing edges) added G by our reduction are dashed. The same dashed edges make up the primary tree in G^+, T^l.

Figure C.1. Example of reduction used in Theorem 4.1.

Proof. 1Multicast Recycling is a special case of Multicast Recycling. Since we have shown 1Multicast Recycling is NP-hard in Theorem C.1, Multicast Recycling must at least be NP-hard. □

BIBLIOGRAPHY

[1] GT-ITM. http://www.cc.gatech.edu/projects/gtitm/.

[2] Northeast blackout of 2003. http://en.wikipedia.org/wiki/Northeast_blackout_of_2003.

[3] Rocketfuel. http://www.cs.washington.edu/research/networking/rocketfuel/maps/weights/weights-dist.tar.gz.

[4] Aazami, A., and Stilp, M.D. Approximation Algorithms and Hardness for Domination with Propagation. *CoRR abs/0710.2139* (2007).

[5] Almes, G., Kalidindi, S., and Zekauskas, M. A one-way packet loss metric for ippm. Tech. rep., RFC 2680, September, 1999.

[6] Ammann, P., Jajodia, S., and Liu, Peng. Recovery from Malicious Transactions. *IEEE Trans. on Knowl. and Data Eng. 14*, 5 (2002), 1167–1185.

[7] Andersson, G, Donalek, P, Farmer, R, Hatziargyriou, N, Kamwa, I, Kundur, P, Martins, N, Paserba, J, Pourbeik, P, Sanchez-Gasca, J, et al. Causes of the 2003 major grid blackouts in north america and europe, and recommended means to improve system dynamic performance. *Power Systems, IEEE Transactions on 20*, 4 (2005), 1922–1928.

[8] Bakken, D.E., Bose, A., Hauser, C.H., Whitehead, D.E., and Zweigle, G.C. Smart generation and transmission with coherent, real-time data. *Proceedings of the IEEE 99*, 6 (2011), 928–951.

[9] Baldwin, T.L., Mili, L., Boisen, M.B., Jr., and Adapa, R. Power System Observability with Minimal Phasor Measurement Placement. *Power Systems, IEEE Transactions on 8*, 2 (May 1993), 707 –715.

[10] Barford, P., and Sommers, J. Comparing probe-and router-based packet-loss measurement. *Internet Computing, IEEE 8*, 5 (2004), 50–56.

[11] Bertsekas, D., and Gallager, R. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[12] Birman, K.P., Chen, J., Hopkinson, E.M., Thomas, R.J., Thorp, J.S., Van Renesse, R., and Vogels, W. Overcoming communications challenges in software for monitoring and controlling power systems. *Proceedings of the IEEE 93*, 5 (2005), 1028–1041.

[13] Bobba, R., Heine, E., Khurana, H., and Yardley, T. Exploring a tiered architecture for NASPInet. In *Innovative Smart Grid Technologies (ISGT), 2010* (2010), IEEE, pp. 1–8.

[14] Brueni, D. J., and Heath, L. S. The PMU Placement Problem. *SIAM Journal on Discrete Mathematics 19*, 3 (2005), 744–761.

[15] Bu, T., Duffield, N.G., Presti, F., and Towsley, D.F. Network tomography on general topologies. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 21–30.

[16] Cáceres, R., Duffield, N.G., Horowitz, J., and Towsley, D.F. Multicast-based inference of network-internal loss characteristics. *Information Theory, IEEE Transactions on 45*, 7 (1999), 2462–2480.

[17] Charikar, M., Chekuri, C., Cheung, T., Dai, Z., Goel, A., Guha, S., and Li, M. Approximation algorithms for directed steiner problems. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998), Society for Industrial and Applied Mathematics, pp. 192–200.

[18] Chen, J., and Abur, A. Placement of PMUs to Enable Bad Data Detection in State Estimation. *Power Systems, IEEE Transactions on 21*, 4 (2006), 1608 –1615.

[19] Clark, D. The design philosophy of the DARPA internet protocols. *ACM SIGCOMM Computer Communication Review 18*, 4 (1988), 106–114.

[20] Cui, J.H., Faloutsos, M., and Gerla, M. An architecture for scalable, efficient, and fast fault-tolerant multicast provisioning. *Network, IEEE 18*, 2 (2004), 26–34.

[21] Curtis, A., Mogul, J., Tourrilhes, J., Yalagandula, P., Sharma, Puneet, and Banerjee, S. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 254–265.

[22] De La Ree, J., Centeno, V., Thorp, J.S., and Phadke, A.G. Synchronized Phasor Measurement Applications in Power Systems. *Smart Grid, IEEE Transactions on 1*, 1 (2010), 20 –27.

[23] Dijkstra, E., and Scholten, C. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11 (1980).

[24] Dughmi, S. Submodular functions: Extensions, distributions, and algorithms. a survey. *CoRR abs/0912.0322* (2009).

[25] El-Arini, K., and Killourhy, K. Bayesian Detection of Router Configuration Anomalies. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data* (New York, NY, USA, 2005), ACM, pp. 221–222.

[26] Feamster, N., and Balakrishnan, H. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[27] Fei, A., Cui, J., Gerla, M., and Cavendish, D. A "dual-tree" scheme for fault-tolerant multicast. In *Communications, 2001. ICC 2001. IEEE International Conference on* (2001), vol. 3, IEEE, pp. 690–694.

[28] Feldmann, A., and Rexford, J. IP Network Configuration for Intradomain Traffic Engineering. *IEEE Network Magazine 15* (2001), 46–57.

[29] Ferguson, Andrew D., Guha, Arjun, Liang, Chen, Fonseca, Rodrigo, and Krishnamurthi, Shriram. Participatory Networking: An API for application control of SDNs. In *ACM SIGCOMM* (2013).

[30] Friedl, A., Ubik, S., Kapravelos, A., Polychronakis, M., and Markatos, E. Realistic passive packet loss measurement for high-speed networks. *Traffic Monitoring and Analysis* (2009), 1–7.

[31] Ganesan, D., Govindan, R., Shenker, S., and Estrin, D. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 4 (2001), 11–25.

[32] Garcia-Lunes-Aceves, J. J. Loop-free Routing using Diffusing Computations. *IEEE/ACM Trans. Netw. 1*, 1 (1993), 130–141.

[33] Garey, M.R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] Gyllstrom, D., Vasudevan, S., Kurose, J., and Miklau, G. Efficient recovery from false state in distributed routing algorithms. In *Networking* (2010), pp. 198–212.

[35] Handigol, N., Heller, B, Jeyakumar, V., Lantz, B., and McKeown, N. Mininet performance fidelity benchmarks. CSTR 2012-02, Stanford Univeristy, October 2012.

[36] Haynes, T. W., Hedetniemi, S. M., Hedetniemi, S. T., and Henning, M. A. Domination in Graphs Applied to Electric Power Networks. *SIAM J. Discret. Math. 15* (April 2002), 519–529.

[37] Heckmann, O., Piringer, M., Schmitt, J., and Steinmetz, R. On Realistic Network Topologies for Simulation. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research* (New York, NY, USA, 2003), ACM, pp. 28–32.

[38] Heller, B. *Reproducible Network Research with High-fidelity Emulation*. PhD thesis, Stanford University, 2013.

[39] Hopkinson, K., Roberts, G., Wang, X., and Thorp, J. Quality-of-service considerations in utility communication networks. *Power Delivery, IEEE Transactions on 24*, 3 (2009), 1465–1474.

[40] Jefferson, D. Virtual Time. *ACM Trans. Program. Lang. Syst. 7*, 3 (1985), 404–425.

[41] Jensen, C., Mark, L., and Roussopoulos, N. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Trans. on Knowl. and Data Eng. 3*, 4 (1991), 461–473.

[42] Ji, P., Ge, Z., Kurose, J., and Towsley, D. A comparison of hard-state and soft-state signaling protocols. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 251–262.

[43] Johnson, Anthony, Tucker, Robert, Tran, Thuan, Paserba, John, Sullivan, Dan, Anderson, Chris, and Whitehead, Dave. Static var compensation controlled via synchrophasors. In *proceedings of the 34th Annual Western Protective Relay Conference, Spokane, WA* (2007).

[44] Johnson, M.J. Analysis of routing table update activity after resource recovery in a distributed computer network. pp. 96–102.

[45] Johnson, M.J. Updating routing tables after resource failure in a distributed computer network. *Networks 14* (1984), 379–391.

[46] Kodialam, M., and Lakshman, T. Dynamic routing of bandwidth guaranteed multicasts with failure backup. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (2002), IEEE, pp. 259–268.

[47] Kotani, D., Suzuki, K., and Shimonishi, H. A design and implementation of openflow controller handling ip multicast with fast tree switching. In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on* (2012), IEEE, pp. 60–67.

[48] Kurose, J., and Ross, K. *Computer networking: a top-down approach featuring the internet*, second edition ed. Addison-Wesley, Reading, 2003.

[49] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[50] Lantz, B., Heller, B., and McKeown, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 19:1–19:6.

[51] Lau, W., Jha, S., and Banerjee, S. Efficient bandwidth guaranteed restoration algorithms for multicast connections. *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems* (2005), 237–243.

[52] Li, G., Wang, D., and Doverspike, R. Efficient distributed mpls p2mp fast reroute. In *Proc. of IEEE INFOCOM* (2006).

[53] Lichtenstein, D. Planar Formulae and Their Uses. *SIAM J. Comput. 11*, 2 (1982), 329–343.

[54] Liu, P., Ammann, P., and Jajodia, S. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases 8*, 1 (2000), 7–40.

[55] Lomet, D., Barga, R., Mokbel, M., and Shegalov, G. Transaction Time Support Inside a Database Engine. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering* (Washington, DC, USA, 2006), IEEE Computer Society, p. 35.

[56] Luebben, R., Li, G., Wang, D., Doverspike, R., and Fu, X. Fast rerouting for ip multicast in managed iptv networks. In *Quality of Service, 2009. IWQoS. 17th International Workshop on* (2009), IEEE, pp. 1–5.

[57] Mccauley, J. POX: A Python-based Openflow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[58] McKeown, Nick, Anderson, Tom, Balakrishnan, Hari, Parulkar, Guru M., Peterson, Larry L., Rexford, Jennifer, Shenker, Scott, and Turner, Jonathan S. Openflow: enabling innovation in campus networks. *Computer Communication Review 38*, 2 (2008), 69–74.

[59] Médard, M., Finn, S.G., Barry, R.A., and Gallager, R.G. Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. *IEEE/ACM Transactions on Networking (TON) 7*, 5 (1999), 641–652.

[60] Mili, L., Baldwin, T., and Adapa, R. Phasor Measurement Placement for Voltage Stability Analysis of Power Systems. In *Decision and Control, 1990., Proceedings of the 29th IEEE Conference on* (Dec. 1990), pp. 3033 –3038 vol.6.

[61] Mittal, V., and Vigna, G. Sensor-Based Intrusion Detection for Intra-domain Distance-vector Routing. In *CCS '02: Proceedings of the 9th ACM Conf on Comp. and Communications Security* (New York, NY, USA, 2002), ACM, pp. 127–137.

[62] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.

[63] Nemhauser, G., Wolsey, L., and Fisher, M. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming 14*, 1 (1978), 265–294.

[64] Neumnann, R. Internet routing black hole. *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems 19*, 12 (May 1997).

[65] Padmanabhan, V., and Simon, D. Secure Traceroute to Detect Faulty or Malicious Routing. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 77–82.

[66] Pfaff, B., et al. Openflow switch specification version 1.1.0 implemented (wire protocol 0x02), 2011.

[67] Pointurier, Y. Link failure recovery for mpls networks with multicasting. Master's thesis, University of Virginia, 2002.

[68] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), ACM, p. 7.

[69] Rosen, E., Viswanathan, A., Callon, R., et al. Multiprotocol label switching architecture.

[70] Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R., and Moore, A. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Passive and Active Measurement* (2012), Springer, pp. 85–95.

[71] School, K., and Westhoff, D. Context Aware Detection of Selfish Nodes in DSR based Ad-hoc Networks. In *Proc. of IEEE GLOBECOM* (2002), pp. 178–182.

[72] Schroeder, T., Goddard, S., and Ramamurthy, B. Scalable web server clustering technologies. *Network, IEEE 14*, 3 (2000), 38–45.

[73] Tian, A.J., and Shen, N. Fast reroute using alternative shortest paths. draft-tian-frr-alt-shortest-path-01.txt, July 2004.

[74] Vanfretti, L. *Phasor Measurement-Based State-Estimation of Electrical Power Systems and Linearized Analysis of Power System Network Oscillations.* PhD thesis, Rensselaer Polytechnic Institute, December 2009.

[75] Vanfretti, L., Chow, J. H., Sarawgi, S., and Fardanesh, B. (B.). A Phasor-Data-Based State Estimator Incorporating Phase Bias Correction. *Power Systems, IEEE Transactions on 26*, 1 (Feb 2011), 111–119.

[76] Wu, C.S., Lee, S.W., and Hou, Y.T. Backup vp preplanning strategies for survivable multicast atm networks. In *Communications, 1997. ICC 97 Montreal,'Towards the Knowledge Millennium'. 1997 IEEE International Conference on* (1997), vol. 1, IEEE, pp. 267–271.

[77] Xu, B., and Abur, A. Observability Analysis and Measurement Placement for Systems with PMUs. In *Proceedings of 2004 IEEE PES Conference and Exposition, vol.2* (2004), pp. 943–946.

[78] Xu, B., and Abur, A. Optimal Placement of Phasor Measurement Units for State Estimation. Tech. Rep. PSERC Publication 05-58, October 2005.

[79] Yardley, J., and Harris, G. 2nd day of power failures cripples wide swath of india, July 31, 2012. `http://www.nytimes.com/2012/08/01/world/asia/power-outages-hit-600-million-in-india.html?pagewanted=all&_r=1&`.

[80] Zhang, J., Welch, G., and Bishop, G. Observability and Estimation Uncertainty Analysis for PMU Placement Alternatives. In *North American Power Symposium (NAPS), 2010* (2010), pp. 1 –8.