MAKING NETWORKS ROBUST TO COMPONENT FAILURE

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

(Compiled on 02/20/2014 at 14:26)

Computer Science

MAKING NETWORKS ROBUST TO COMPONENT FAILURE

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 Clarke, Department Chair
Computer Science

ABSTRACT

MAKING NETWORKS ROBUST TO COMPONENT FAILURE

(COMPILED ON 02/20/2014 AT 14:26)

DANIEL P. GYLLSTROM

B.Sc., TRINITY COLLEGE

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Jim Kurose

Communication network components – routers, links connecting routers, and sensors – inevitably fail, causing service outages and a potentially unusable network. Recovering quickly from these failures is vital to both reducing short-term disruption and increasing long-term network survivability. 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 component failure. This thesis divides into three parts: (a) recovery from malicious or misconfigured nodes injecting false information into a distributed system (e.g., the Internet), (b) placing smart grid sensors to provide measurement error detection, and (c) 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 a type of sensor, 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.

Lastly, we design algorithms for fast recovery from link failures in a smart grid communication network. This is a two-part problem: (a) link detection failure and (b) algorithms for pre-computing backup multicast trees. To address (a), we design link-detection failure 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 propose a set of algorithms that precompute backup multicast trees to be used after a link failure. Each algorithm aims to minimize end-to-end packet loss and delay but each uses different optimization criteria to achieve this goal: minimizing control overhead, minimizing the maximum number of flows impacted by the "next" link failure (MIN-FLOWS), and minimizing the maximum number of sink nodes impacted by the "next" link failure (MIN-SINKS). We implement and evaluate these algorithms in Openflow.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

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 smart grid – can be made more robust to component failure.*

We propose on-demand recovery algorithms for distributed network algorithms that optimize for control message overhead and convergence time, and preplanned approaches to recovery for electric power grid applications, where reliability is key. An electric power grid consists of a set of buses - electric substations, power generation centers, or aggregation points of electrical loads - and transmission lines connecting those buses. We refer to modern and future electric power grids that automate power grid operations using sensors and wide-area communication as the *smart grid*.

0.1 Thesis Overview

0.1.1 Component Failure in Communication Networks

In this thesis, we consider three separate but related problems: 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. 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 [54].

In particular, component failure in a smart grid can be 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 [68].

0.1.2 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 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.

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

In our second thesis chapter, we study a type of sensor, 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 thesis chapter, we present the initial design for algorithms that provide recovery from link failures in a smart grid communication network. The recovery problem divides into two parts: (a) link failure detection and (b) algorithms for pre-computing backup multicast trees. To address (a), we sketch the design of a 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 centralized network management and control. For part (b), we propose initial outlines for a set of

algorithms that precompute backup multicast trees to be installed after a link failure. As future work, we plan to implement these algorithms in Openflow and evaluate them.

0.2 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 costs, 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 initial approaches for algorithms that perform preplanned recovery from link failures in a smart grid communication network. Our proposed research divides into two parts: link failure detection and algorithms for precomputing backup multicast trees. For the first part, we design algorithms that use OpenFlow to detect and report link failures when and where they occur, *inside* the network. To address the second part, we propose a set of algorithms that precompute backup multicast trees that are installed after a link failure. Each algorithm computes a backup multicast tree that aims to minimize end-to-end packet loss and delay, but each algorithm uses different optimization criteria in achieving this goal: minimizing control overhead, minimizing the maximum number of flows impacted by the "next" link failure (MIN-FLOWS), and minimizing the maximum number of sink nodes impacted by the "next" link failure (MIN-SINKS). These optimization criteria differ from those proposed in the literature.

0.3 Thesis Outline

The rest of this thesis proposal is organized as follows. We present algorithms for recovery from false routing state in distributed routing algorithms in Chapter 1. In Chapter 2 we formulate PMU placement problems that provide measurement error detection. Chapter 3 presents our initial and proposed research on efficient recovery from link failures in a smart grid communication network. We conclude by outlining planned future work in Chapter 4.

CHAPTER 1

RECOVERY FROM FALSE ROUTING STATE IN DISTRIBUTED ROUTING ALGORITHMS

1.1 Introduction

TODO Notes from Proposal Defense:

- Prashant: distributed consistent snapshots paper from 687 class, no synchronized clocks needed.

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 [54]. 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 sim-

ulation. Our analysis and simulations show that when considering topologies in which link costs 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 costs can change, PURGE outperforms CPR and 2ND-BEST. CPR performs poorly because, following rollback, it must process the valid link cost 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 [7, 44] in distributed databases. Our problem is also similar to that of rollback in optimistic parallel simulation [37]. 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 1.2, 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 1.2 we define the false-state-removal problem and state our assumptions. We present our three recovery algorithms in Section 1.3. Then, in Section 1.4, we briefly state the results of our message complexity analysis. Section 1.5 describes our simulation study. We detail related work in Section 1.6 and conclude the chapter in Section 1.7. The research described here has been published in [33].

1.2 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 $w : E \rightarrow \mathbb{N}$. [1] Each node, v, maintains the following state as part of distance vector: 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 1.3.5. We assume that the identity of the compromised node is provided by a different algorithm, and thus do not consider this problem in this paper. Examples of such algorithms include [27, 28, 51, 55, 60]. 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.

---

[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.

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 1.1. Table of abbreviations.

Table 1.1 summarizes the notation used in this chapter.

1.3 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 $V' = V - \{\overline{v}\}$, $E' = E - \{(\overline{v}, v_i) \mid v_i \in adj(\overline{v})\}$, 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.

[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.

1.3.1 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 [24] 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 1.1 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

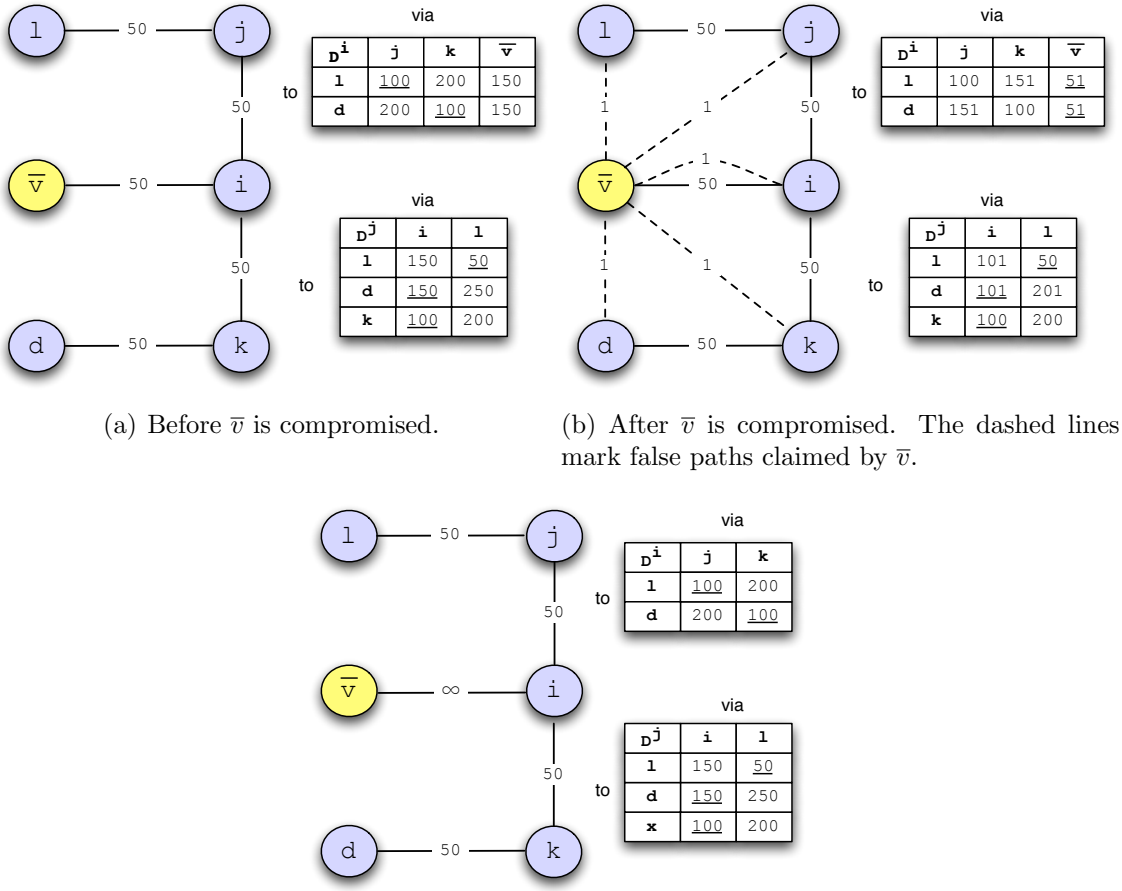
list of nodes in its connected component. Finally, i broadcasts the vector of node IDs in its connected component.

1.3.2 The 2^{nd} best Algorithm

2ND-BEST invalidates state locally and then uses distance vector to implement network-wide recovery. Following the preprocessing described in Section 1.3.1, 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 1.1. In Figure 1.1(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 1.1, 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 1.1. 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.

1.3.3 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 1.1, 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 makes no synchronization assumptions. Also, the 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.

1.3.4 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 δ. 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 compromised.

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

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

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 cost 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 [38, 45].

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 cost change event may only have propagated through part of the network, in which case the snapshot for some nodes will reflect this link cost 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.

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

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

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'.) Algorithm A.1.4 in the Appendix gives the pseudo-code for the rollback algorithm.

Step 3: Steps after rollback. After Step 2 completes, the algorithm in Section 1.3.1 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 cost 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 1.1, the global state after rolling back is nearly the same as the snapshot depicted in Figure 1.1(c): the only difference between the actual system state and that depicted in Figure 1.1(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 cost 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.

1.3.5 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 1.3.1 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 computation 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

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

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

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 costs 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 ∞.

1.4 Analysis of Algorithms

Here we summarize the results from our analysis. The detailed proofs can be found in our corresponding technical report [34]. 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 costs, we find that the communication complexity of all three algorithms is 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 costs 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 costs 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 costs can change, while CPR is not. CPR's upper bound becomes $O(mnd) + O(wn^2)$.

1.5 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 topologies (e.g., Erdös-Rényi graphs, Internet-like graphs) and across different network conditions (e.g., topologies with fixed link costs, topologies with changing link costs, a single compromised node, and multiple compromised nodes).

We build a custom simulator with a synchronous communication model: 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). 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 simulate the following scenario:

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. The effect of $\overrightarrow{bad}$ spreads for a specified number of hops (this varies by experiment). The variable k refers to the number of hops that the effect of $\overrightarrow{bad}$ has spread.

4. At time t, some node $v \in V$ notifies all $v \in adj(\overline{v})$ that $\overline{v}$ was compromised. [7]

The message and time overhead are measured in step (4) above. The pre-computation, described in Section 1.3.1, is not counted towards message and time overhead because all three recovery algorithms use this same procedure.

In order to keep this thesis proposal document brief, we only discuss a subset of our simulation results. We focus on a simplified scenario in which a single compromised node has distributed false routing state. We consider Erdös-Rényi graphs in cases where link costs remain fixed (Section 1.5.1) and link costs change (Section 1.5.2). The corresponding technical report [34], discusses results using a richer simulation model that considers more realistic network conditions: an asynchronous communication model, Internet-like graphs generated using GT-ITM [1] and Rocketfuel [4], and multiple compromised nodes. We find that the trends discussed in this report hold when using these new topologies and additional simulation scenarios.

1.5.1 Simulations using Graphs with Fixed Link Weight

Here we evaluate our recovery algorithms, in terms of message and time overhead, using Erdös-Rényi graphs with fixed link weights. In particular, we consider Erdös-Rényi graphs with parameters n and p, where n is the number of graph nodes and p is the probability that link (i, j) exists where $i, j \in V$. Link weights are selected uniformly at random between $[1, n]$.

In order to establish statistical significance, we generate several Erdös-Rényi graphs to be used in our simulations. We iterate over different values of k. For each k, we generate an Erdös-Rényi graph, $G = (V, E)$, with parameters n and p. Then we select a $v \in V$ uniformly at random and simulate the scenario described above, using v as the compromised node. In total we sample 20 unique nodes for each G. We set

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

$k=1$	$k=2$	$k=3$	$k=4-10$
554	1303	9239	12641

Table 1.2. Average number pairwise routing loops for 2ND-BEST in simulation described in Section 1.5.1.

$n = 100$, $p = \{0.05, 0.15, 0.25, 0.25\}$, and let $k = \{1, 2, ...10\}$. Each data point is an average over 600 runs (20 runs over 30 topologies).

Figure 1.2 shows the message overhead as a function of k when $p = .05$. The 90% confidence interval is included in the plot. CPR outperforms the other algorithms because CPR removes false routing state with a single diffusing computation, rather than using an iterative process as in 2ND-BEST and PURGE. 2ND-BEST performs poorly because of the count-to-infinity problem: Table 1.2 shows the large average number of pairwise routing loops, an indicator of the occurrence of count-to-infinity problem, 2ND-BEST encounters this simulation. 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. In contrast, no routing loops are found with PURGE or CPR, as expected.

However, CPR's encouraging results should be interpreted with caution. CPR requires both loosely synchronized clocks and the time that node $\overline{v}$ was compromised to be identified, assumptions not required by 2ND-BEST nor PURGE. Furthermore, this first simulation scenario is ideal for CPR because fixed link costs ensure minimal stale state (i.e., residual $\overrightarrow{old}$ state) after CPR rolls back. The next two simulations present more challenging and less favorable conditions for CPR.

1.5.2 Simulations using Graphs with Changing Link Weights

In the next two simulations we evaluate our algorithms over graphs with changing link costs. We introduce link cost changes between the time $\overline{v}$ is compromised and

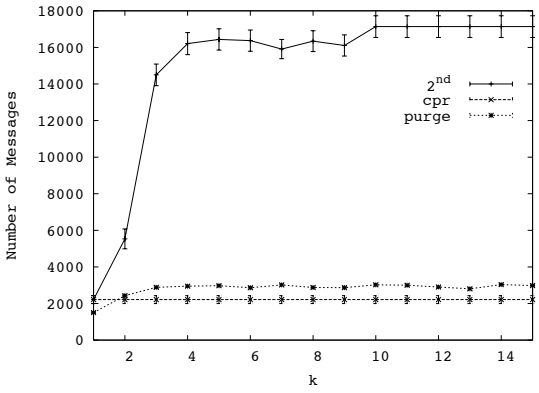


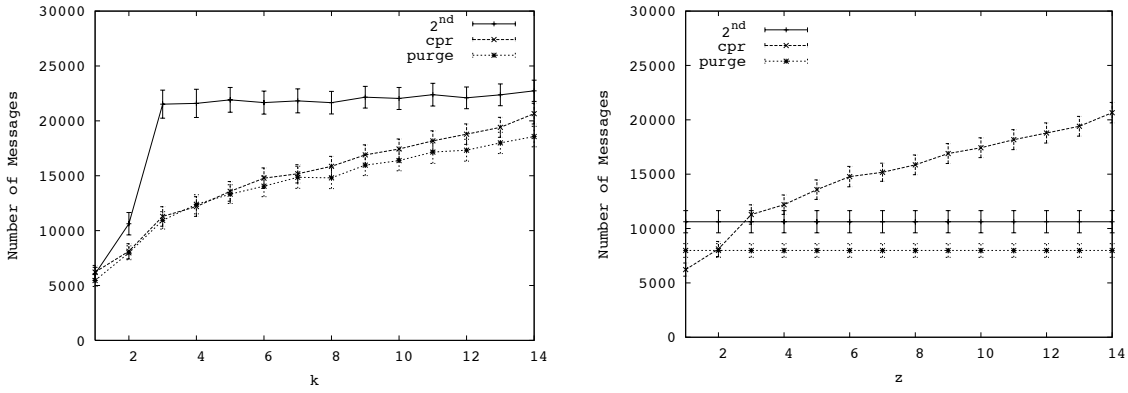
Figure 1.2. Message overhead as function of the number of hops false routing state has spread from the compromised node (k), over Erdös-Rényi graphs with fixed link weights. The Erdös-Rényi graphs are generated using $n = 100$, and $p = .05$, yielding an average diameter of 6.14.

when $\overline{v}$ is discovered (e.g. during $[t', t]$). In particular, there are λ link cost changes per timestep, where λ is deterministic. To create a link cost change event, we modify links uniformly at random (except for all $(v, \bar{v})$ links), where the new link cost is selected uniformly at random from $[1, n]$.

Effects of Link Cost Changes. Except for λ, our simulation setup is identical to that of the previous section. We let $\lambda = \{1, 4, 8\}$. In order to isolate the effects of link costs changes, we assume that CPR checkpoints at each timestep.

For the sake of brevity, we only show results for $n = 100, p = .05, \lambda = 4$ in Figure 1.3(a). [8] PURGE yields the lowest message overhead, but only slightly lower than CPR. CPR's message overhead increases with larger k because there are more link cost change events to process. After CPR rolls back, it must process all link cost changes that occurred in $[t', t]$. In contrast, 2ND-BEST and PURGE process some of the link cost change events during the interval $[t', t]$ as part of normal distance vector execution.

[8] Our simulations for different p values, yield the same trends. Refer to our technical report for more details [34].



(a) Message overhead as a function of the number of hops false routing state has spread (k) where $\lambda = 4$ link cost changes occur at each timestep.

(b) Message overhead as function of varying the checkpoint frequency, z. $z = 0$ implies checkpointing occurs at every timestep, $z = 1$ creates checkpoints every other timestamp, etc.

Figure 1.3. Section 1.5.2 plots. Both plots consider Erdös-Rényi graphs with changing link costs generated using $n = 100$ and $p = .05$. The average diameter of the generated graphs is 6.14.

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 link cost changes have a larger effect on message overhead as λ grows.

Effects of Varying 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 $[t', t]$. Otherwise, our simulation setup is the same as the one just described (under the title "Effects of Link Cost Changes").

For conciseness, we only display a single plot. Figure 1.3(b) shows the results for an Erdös-Rényi graph with link weights selected uniformly at random between $[1, n]$, $n = 100$, $p = .05$, $\lambda = 4$ 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 cost 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.

1.5.3 Summary of Simulation Results

Our results show that for graphs with fixed link costs, CPR yields the lowest message and time overhead. CPR benefits from removing false state with a single diffusing computation. However, CPR has storage overhead, requires loosely synchronized clocks, and requires the time that node $\overline{v}$ was compromised to be identified.

2ND-BEST's performance is determined by the count-to-infinity problem. PURGE avoids the count-to-infinity problem by first globally invalidating false state. Therefore in cases where the count-to-infinity problem is significant, PURGE outperforms 2ND-BEST.

When considering graphs with changing link costs, CPR's performance suffers because it must process all valid link cost changes that occurred since $\overline{v}$ was compromised. Meanwhile, 2ND-BEST and PURGE make use of computations that followed the injection of false state, that do not depend on false routing state. However, 2ND-BEST's performance degrades because of the count-to-infinity problem. PURGE eliminates the count-to-infinity problem and therefore yields the best performance over topologies with changing link costs.

Finally, we found that an additional challenge with CPR is setting the parameter that determines the checkpoint frequency. More 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.

1.6 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. [7] and Ammann et al [44] 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. [44]. Additionally, if we treat link cost change events that occur after the compromised node has been discovered as database transactions, we face a similar design decision as in [7]: do we wait until recovery is complete before applying link cost changes or do we allow the link cost changes to execute concurrently?

Database crash recovery [52] and message passing systems [27] 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 [31] uses diffusing computations to coordinate least cost updates in order to prevent routing loops. In our case, CPR and the prepossessing procedure (Section 1.3.1) 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 [37] proposes a solution to synchronize distributed systems called Time Warp. Time Warp is a form of optimistic concurrency control and, as such, occasion-

ally 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.

1.7 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 show that CPR – a checkpoint-rollback based algorithm – yields the lowest message and time overhead over topologies with fixed link costs. However, CPR has storage overhead and requires loosely synchronized clocks. In the case of topologies with changing link costs, PURGE performs best by avoiding the problems that plague 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 results in high message overhead for 2ND-BEST, while PURGE eliminates the count-to-infinity problem by globally purging false state before finding new least cost paths.

CHAPTER 2

PMU SENSOR PLACEMENT FOR MEASUREMENT ERROR DETECTION IN THE SMART GRID

2.1 Introduction

TODO Notes from Proposal Defense:

- Lixin: Approximation bounds using modularity/sub-modular functions.
- Lixin: mention in future work (may already do this) that with special topologies you may be able to find more efficient algorithms for PMU placement.
- State Aazami et al show that the approximation for greedy algorithm is $\Theta(n)$, under the assumption that all nodes are zero-injection.

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 [14].

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][23]. Fortunately, the voltage phasor at a system bus can, at times, be determined (termed *observed* in this paper) 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][15]. 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, 15, 35, 50, 66], 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 [64]. 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 formulation 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 2.2 we introduce our modeling assumptions, notation, and observability and cross-validation rules. In Section 2.3 we formulate and prove the complexity of our four PMU placement problems. Section 2.4 presents the approximation algorithms for each problem and Section 2.5 considers our simulation-based evaluation. We conclude with a review of related work (Section 2.6) and concluding remarks (Section 2.7).

2.2 Preliminaries

In this section we introduce notation and underlying assumptions (Section 2.2.1), and define our observability (Section 2.2.2) and cross-validation (Section 2.2.3) rules.

2.2.1 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 [69]. 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, 15, 18, 50, 66, 67], 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 [15], 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*.

2.2.2 Observability Rules

We use the simplified observability rules stated by Brueni and Heath [15]. For completeness, we restate the rules here:

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 2.1, 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 2.5).

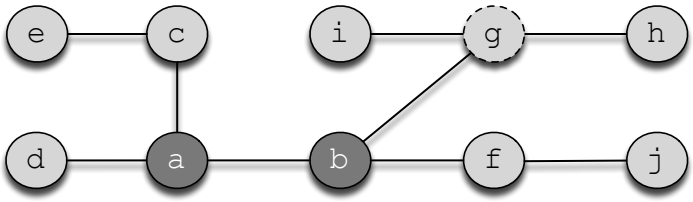


Figure 2.1. Example power system graph. PMU nodes (a, b) are indicated with darker shading. Injection nodes have solid borders while zero-injection nodes (g) have dashed borders.

2.2.3 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. [64], 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 [64]:

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 2.1, 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

[1] Vanfretti et al. [64] use the term "redundancy" instead of cross-validation.

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) [63]. 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 [63].

2.3 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 our proof strategy in this paper (Section 2.3.1). In the remainder of Section 2.3 we present and prove the NP-Completeness of four PMU placement problems, in the following order: FullObserve (Section 2.3.2), MaxObserve (Section 2.3.3), FullObserve-XV (Section 2.3.4), and MaxObserve-XV (Section 2.3.5).

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.

2.3.1 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 2.5 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, $x \in \Pi' \Leftrightarrow f(x) \in \Pi$ [32].

Next, we outline the proof strategy we use throughout this section. In Sections 2.3.2 through Section 2.3.5 we use slight variations of the approach presented by Brueni and Heath in [15] 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 [15], 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:

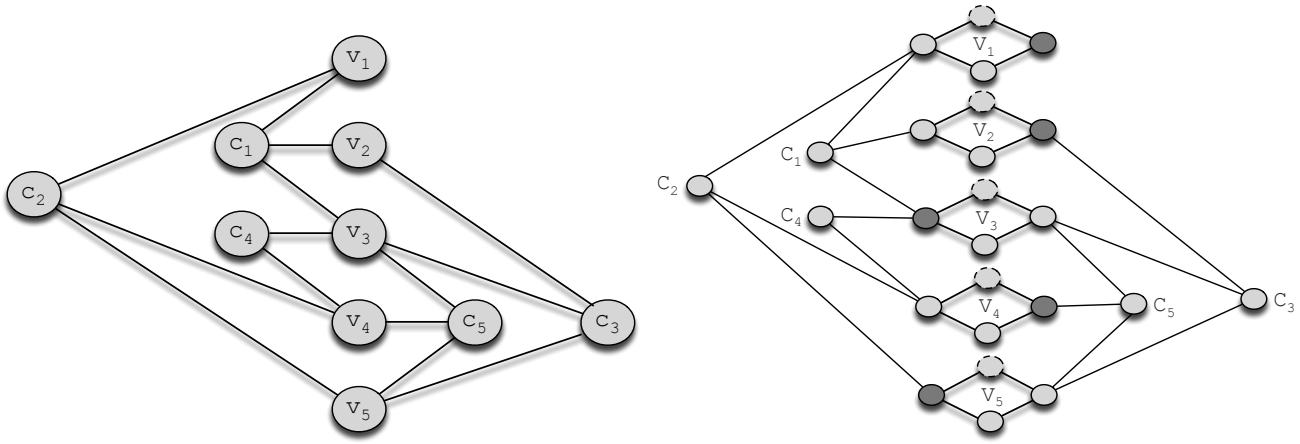
$$
\begin{aligned}
V(\phi) &= \{v_i \mid 1 \leq i \leq r\} \cup \{c_j \mid 1 \leq j \leq s\} \\
E(\phi) &= \{(v_i, c_j) \mid v_i \in c_j \ \ or \ \ \overline{v_i} \in c_j\}.
\end{aligned}
$$

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 [43]. For example, P3SAT formula

$$
\begin{aligned}
\varphi &= (\overline{v_1} \vee v_2 \vee v_3) \wedge (\overline{v_1} \vee \overline{v_4} \vee v_5) \wedge (\overline{v_2} \vee \overline{v_3} \vee \overline{v_5}) \\
&\quad \wedge (v_3 \vee \overline{v_4}) \wedge (\overline{v_3} \vee v_4 \vee \overline{v_5}) \qquad (2.1)
\end{aligned}
$$

has graph $G(\varphi)$ shown in Figure 2.2(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 [15], 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.



(a) $G(\varphi)$ formed from φ in Equation (2.1). (b) Graph formed from φ formula in Theorem 2.1 proof.

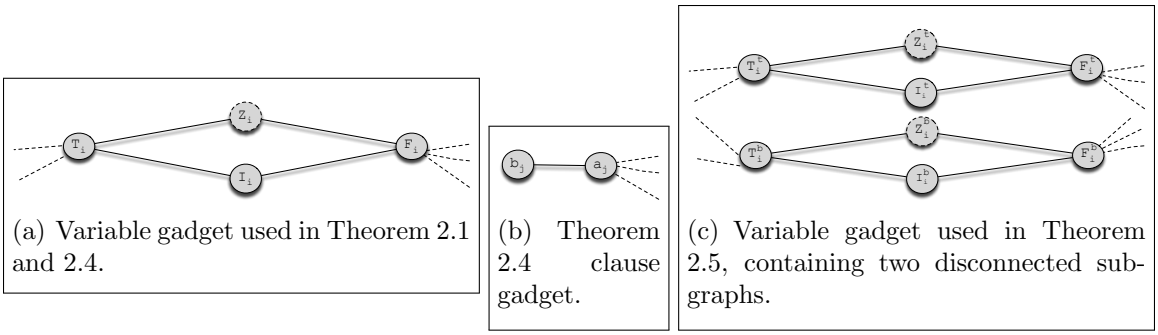
Figure 2.2. The figure in (a) shows $G(\varphi) = (V(\varphi), E(\varphi))$ using example formula, φ, from Equation (2.1). (b) shows the new graph formed by replacing each variable node in $G(\varphi)$ – as specified by the Theorem 2.1 proof – with the Figure 2.3(a) variable gadget.

Although the structure of our proofs is adapted from [15], 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 [15] 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 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.

2.3.2 The FullObserve Problem

The FullObserve problem has been addressed in the literature (e.g., the PMUP problem in [15], and the PDS problem in [35]) but only for purely zero-injection bus



(a) Variable gadget used in Theorem 2.1 and 2.4.

(b) Theorem 2.4 clause gadget.

(c) Variable gadget used in Theorem 2.5, containing two disconnected subgraphs.

Figure 2.3. Gadgets used in Theorem 2.1 - 2.7. Z_i in Figure 2.3(a), Z_i^t in Figure 2.3(c), and Z_i^b in Figure 2.3(c) are the only zero-injection nodes. The dashed edges in Figure 2.3(a) and Figure 2.3(c) are connections to clause gadgets. Likewise, the dashed edges in Figure (b) are connections to variable gadgets. In Figure 2.3(c), superscript, t, denotes nodes in the upper subgraph and superscript, b, indexes nodes in the lower subgraph.

systems. Here we consider networks with mixtures of injection and zero-injection buses, and modify the NPC proof of PMUP in [15] 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:

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 2.1. 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.

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

For our first problem, we use a single node as a clause gadget denoted a_j, and the subgraph shown in Figure 2.3(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 2.2. *Consider the gadget shown in Figure 2.3(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 2.1. 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 2.2.2, 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 2.3(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 2.3. *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 2.2(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 2.2(b). While this construction generates a graph with very specific structure, in Section 2.3.6, 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 2.2(b) we know that all nodes in variable gadgets are observed by such a placement. From Observation 2.3, 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 2.2(a) we know that for each V_i with no PMU, at least two nodes are not

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

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 2.2(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 2.3 this means that all clauses are satisfied by the semantic interpretation of our PMU placement, which concludes our proof. □

2.3.3 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 2.4. 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 2.1.

Proof. MaxObserve $\in \mathcal{NP}$ using the same argument in the proof for Theorem 2.1.

Next, we reduce from P3SAT as in the proof for Theorem 2.1, 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 2.3(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 2.3 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 2.3.6 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 2.1 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 2.2(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 2.1 demonstrated this can only be done by placing PMUs at nodes corresponding to a satisfying assignment of ϕ, and so our proof is complete. □

2.3.4 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 2.2.3.

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 2.5. 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 2.3(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 2.6. *Consider the gadget shown in Figure 2.3(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 2.2, since both V_i^t and V_i^b are identical to the gadget from Figure 2.3(a), which Lemma 2.2 refers to. □

Proof of Theorem 2.5. 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 2.1 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 2.1. 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 2.3(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 2.3 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 2.3, 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 2.6(b) we know the assignment fully observes all the variable gadgets. From Observation 2.3 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.

---

[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.

From Lemma 2.6(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 2.6(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 we assume the clause nodes are all observed and we know no PMUs are on clause nodes, from Observation 2.3 this means the PMU placement satisfies all clauses, which concludes our proof. □

2.3.5 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 2.2.3.

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 2.7. 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 2.3(b)) and the variable gadgets from FullObserve-XV (Figure 2.3(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 2.5, 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 2.5.

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 2.3(b)) and the variable gadgets from FullObserve-XV (Figure 2.3(c)). As in Theorem 2.5, 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 2.5. Since Φ_H only has PMUs on variable gadgets, all a_j nodes are observed by the same argument used in Theorem 2.5. Thus, at least $8r + s$ nodes are observed in H. Because no PMU in Φ_H is placed on a clause gadget, C_j, 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 2.5, 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.

[5]See previous note on FullObserve-XV

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 2.5 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 2.5. We use the Theorem 2.5 proof to determine that all clauses in ϕ are satisfied by the truth assignment derived from Φ_H. □

2.3.6 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 2.3.2 - Section 2.3.5 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 2.4(a) in place of the original variable gadget (Figure 2.3(a)). Our proofs for Theorem 2.1 and Theorem 2.4 can remain largely unchanged because the same PMU placement described in each NP-Completeness proof observes these newly introduced nodes. [6] For FullObserve-

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

XV and MaxObserve-XV we increase $|V_I|$ using the variable gadget shown in Figure 2.4(b). The PMU placements described in the proofs for Theorem 2.5 and Theorem 2.7 observe all newly introduced nodes in Figure 2.4(b).

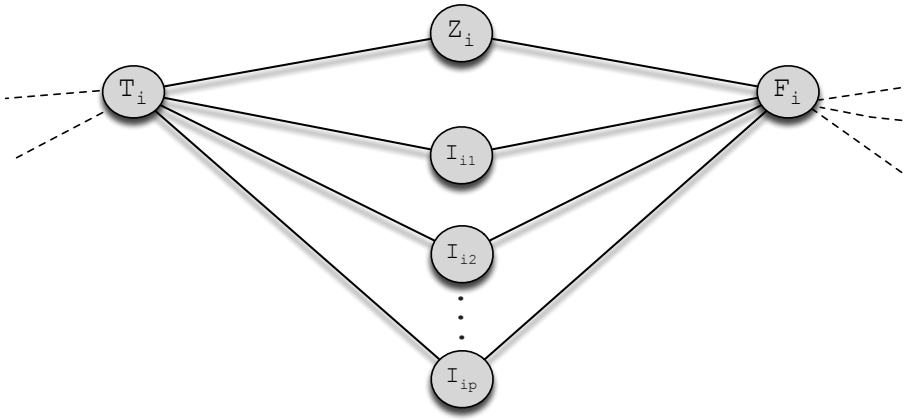
Similarly, the *number of zero-injection nodes* $|V_Z|$ can modified by changing the variable gadgets. FullObserve and MaxObserve – using the variable gadget shown in Figure 2.5(a) – and FullObserve-XV and MaxObserve-XV – using the variable gadget shown in Figure 2.5(a) – are easily extended to include more zero-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 2.6. 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 2.4 to show MaxObserve is NP-Complete for any $\frac{m}{|V|}$. A similar argument can be made for MaxObserve-XV.

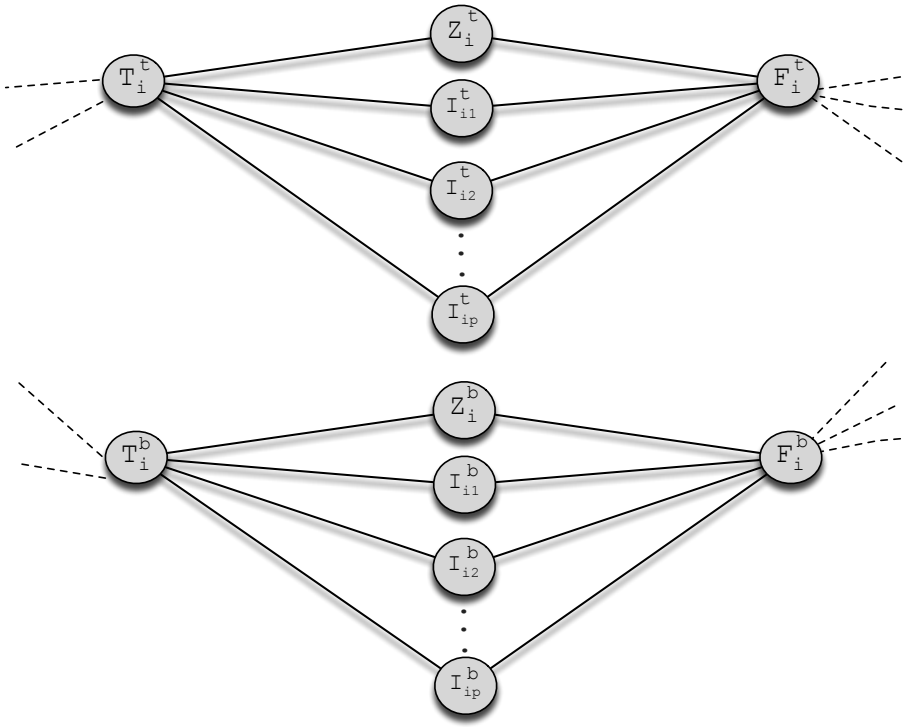
2.4 Approximation Algorithms

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

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

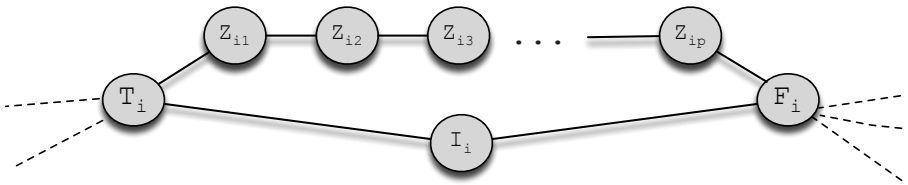


(a) Modified variable gadget used in FullObserve and MaxObserve, containing additional injection nodes: $I_{i1}, I_{i2}, \ldots I_{ip}$.

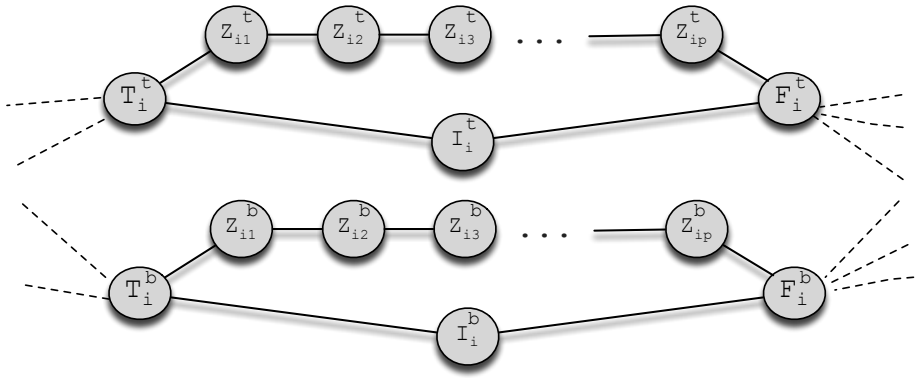


(b) Modified variable gadget used in FullObserve-XV and MaxObserve-XV. Each disconnected subgraph has additional injection nodes: nodes $I^t_{i1}, I^t_{i2}, \ldots I^t_{ip}$ are added to the upper subgraph and nodes $I^b_{i1}, I^b_{i2}, \ldots I^b_{ip}$ are included in the bottom subgraph.

Figure 2.4. Figures for variable gadget extensions to include more injection nodes described in Section 2.3.6. 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 2.5. Figures for variable gadget extensions to include more non-injection nodes described in Section 2.3.6. The dashed edges indicate connections to clause gadget nodes.

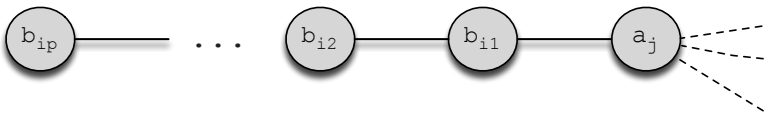


Figure 2.6. Extended clause gadget, C'_j, used in Section 2.3.6. All nodes are injection nodes.

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. Lastly, we explore the possibility that the PMU observability rules are submodular functions (Section 2.4.2).

2.4.1 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 2.8. *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 2.9. *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). □

---

[8] The same greedy algorithm is proposed by Aazami and Stilp [5] and is shown to $\Theta(n)$ approximation ratio under the assumption that all nodes are zero-injection.

2.4.2 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 function on X as $f : 2^X \rightarrow \mathbb{R}$. Using the definition from Dughmi [26] f is *submodular* if, for all $A, B \subseteq X$ with $A \subseteq B$, and for each $j \in X$,

$$f(A \cup \{j\}) - f(A) \quad \geq \quad f(B \cup \{j\}) - f(B) \tag{2.2}$$

It has been shown that greedy algorithms admit a $1 - 1/e$ approximation of submodular functions [53], 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 2.10) but is submodular when restricted to graphs with only injection nodes (Theorem 2.11).

Theorem 2.10. *f is not submodular for graphs, G_z, with zero-injection nodes.*

Proof. Let G_z be the graph from Figure 2.7, $A = \{a\}$, 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. □

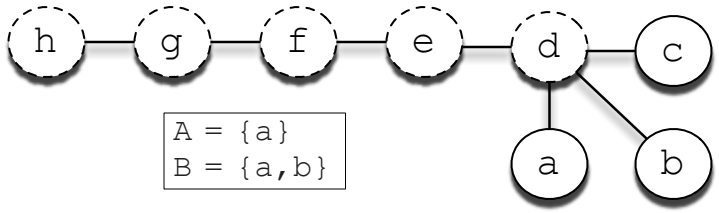


Figure 2.7. Example used in Theorem 2.10 showing a function defined using our observability rules is not submodular for graphs with zero-injection nodes. Nodes with a dashed border are zero-injection nodes and injection nodes have a solid border. For set function $f : 2^X \rightarrow \mathbb{R}$, defined 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.

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 (2.11).

Theorem 2.11. *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. □

2.5 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`.

---

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

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

We present three different simulations in Section 2.5.1-2.5.3. In Section 2.5.1 we consider performance as a function of the number of PMUs, and in Section 2.5.2 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 in Section 2.5.3 where we compare these results to the performance over the actual IEEE graphs.

2.5.1 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 2.8, and Figure 2.9 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 an open question whether the greedy algorithms used here would do well for larger graphs.

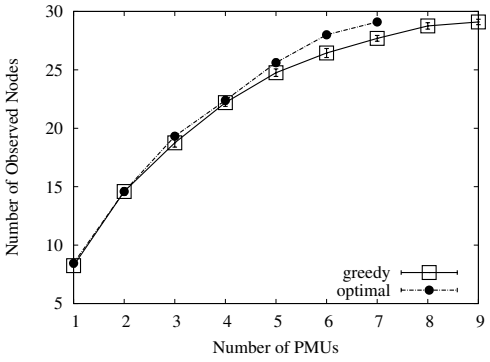
Surprisingly, when we compare our results with and without the cross-validation requirement, we find that this set of constraints does not have a significant effect on the number of observed nodes for the same k. Our experiments show that on average `xvoptimal` observed only 5% fewer nodes than `optimal`. 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.

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

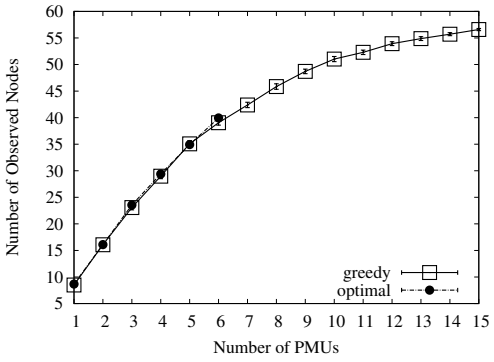
Next, we examine the impact of $|V_Z|$ on algorithm performance. For each synthetic graph, we run our algorithms for increasing values of $|V_Z|$ and determine the minimum number of PMUs needed to observe all nodes in the graph (k^*). For each $z := |V_Z|$, we select z nodes uniformly at random to be 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 2.5.1. 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.

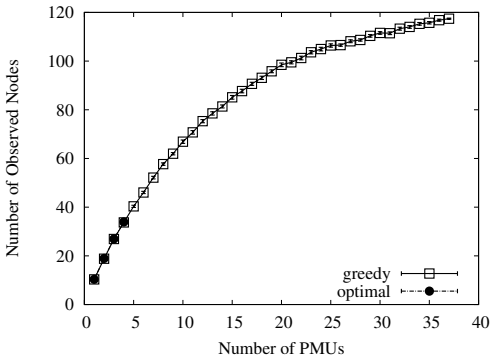
Figure 2.10(a) shows the simulation results for solving FullObserve and FullObserve-XV on synthetic graphs modeled by IEEE bus 57. Results for other topologies con-



(a) Graphs based on IEEE Bus 30



(b) Graphs based on IEEE Bus 57



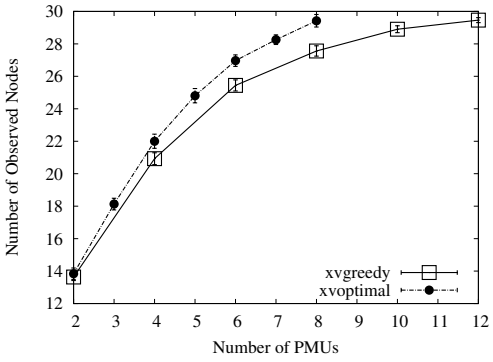
(c) Graphs based on IEEE Bus 118

Figure 2.8. Mean number of observed nodes over synthetic graphs – using `greedy` and `optimal` – when varying number of PMUs. The 90% confidence interval is shown.

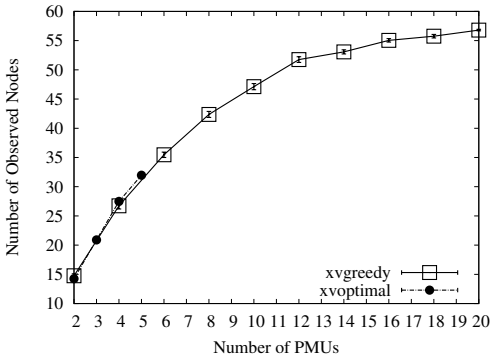
sidered here (i.e., 14, 30 and 118) followed the same trend and are thus omitted. Due to the exponential running time of `optimal` and `xvoptimal`, we present here only results of our greedy algorithms.

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 2.10(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.

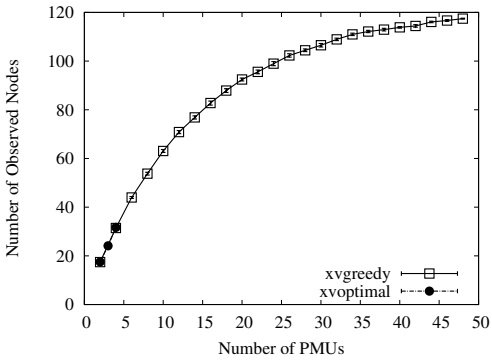
The gap in k^* between `greedy` and `xvgreedy` decreases as z grows. `greedy` and `xvgreedy` observe a similar number of nodes via O2 across all z values: the mean



(a) Graphs based on IEEE Bus 30



(b) Graphs based on IEEE Bus 57



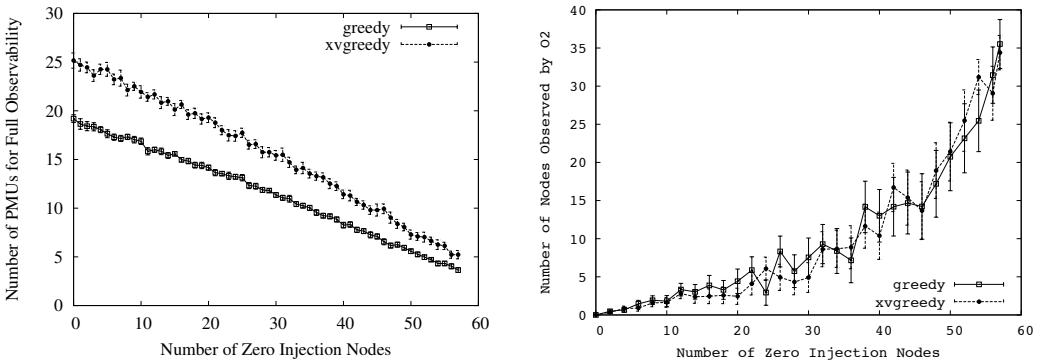
(c) Graphs based on IEEE Bus 118

Figure 2.9. Over synthetic graphs, mean number of observed nodes – using `xvgreedy` and `xvoptimal` – when varying number of PMUs. The 90% confidence interval is shown.

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 2.10(b)), causing the gap between `greedy` and `xvgreedy` to shrink.

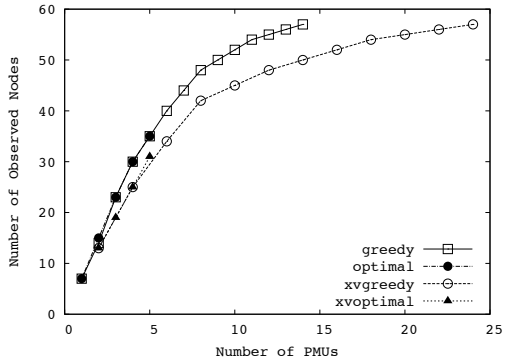
2.5.3 Simulation 3: Synthetic vs Actual IEEE Graphs

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.



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

Figure 2.10(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, 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]

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

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 `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 2.1. 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.

2.6 Related Work

FullObserve is well-studied [9, 15, 35, 50, 66]. Haynes et al. [35] and Brueni and Heath [15] 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 [15] in all four of our NPC proofs, although our proofs differ considerably in their details.

In the power systems literature, Xu and Abur [66, 67] use integer programming to solve FullObserve, while Baldwin et al. [9] and Mili et al. [50] 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 chapter papers make no mention that

FullObserve is NPC, i.e., they do not characterize the fundamental complexity of the problem.

Aazami and Stilp [5] 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 2.4, 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. [64] 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. [64] 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.

2.7 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 [66] to solve FullObserve. This would provide valuable data points to measure the relative performance of `greedy`.

CHAPTER 3

RECOVERY FROM LINK FAILURES IN A SMART GRID COMMUNICATION NETWORK

3.1 Introduction

TODO Notes from Proposal Defense:

- Motivation regarding alternative solutions: private vs public network, separate networks for control and grid itself, power link communication. need to motivate having a separate network and that generic routers are insufficient.
- 2 types of Backup Computations: (1) initialization, and (2) after each link failure
- Including writing about efficient implementation of algorithms using OpenFlow
- Mention in future work about simultaneous link failures: the input to the new version of each algorithm will now take a vector of links, instead of a single link.
- Distinguish between link failure, packet loss, packet delay + be consistent with terminology.

An electric power grid consists of a set of buses – electric substations, power generation centers, or aggregation points of electrical loads – and transmission lines

connecting those buses. The operation of the power grid can be greatly improved by high-frequency voltage and current measurements. Phasor Measurement Units (PMUs) are sensors that provide such measurements. PMUs are currently being deployed in electric power grids worldwide, providing the potential to both (a) drastically improve existing power grid operations and applications and (b) enable an entirely new set of applications, such as real-time visualization of electric power grid dynamics and the reliable integration of renewable energy resources.

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 (e.g., the August 2003 blackout in the USA [1] and the recent power grid failures in India [68]).

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 [13]. Instead, either a new network architecture or enhancements to Internet architecture and protocols are needed [8, 13, 14, 36] 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].

---

[1] `http://en.wikipedia.org/wiki/Northeast_blackout_of_2003`

In this last piece of our research, we design algorithms for fast recovery from link failures in a Smart Grid communication network. Informally, we consider a link that does not meet its packet delivery requirement (either due to excessive delay or actual packet loss) as failed. Our proposed research divides broadly into two parts:

- **Link detection failure.** Here, we design link-failure detection and reporting mechanisms that use OpenFlow [48] – an open source framework that centralizes network management and control – 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 [6, 16, 30] focuses on measuring end-to-end packet loss, resulting in slower detection times.

- **Algorithms for pre-computing backup multicast trees.** Inspired by MPLS fast-reroute algorithms that are used in practice to quickly reroute time-critical unicast IP flows over pre-computed backup paths [21, 29, 49, 57, 65], we propose a set of algorithms, each of which computes backup multicast trees that are installed after a link failure. We also implement these algorithms in OpenFlow and demonstrate their performance.

  Each algorithm computes backup multicast trees that aim to minimize end-to-end packet loss and delay, but each algorithm uses different optimization criteria in achieving this goal: minimizing control overhead (Min-Control), minimizing the maximum number of flows impacted by the "next" link failure (Min-Flows), and minimizing the maximum number of sink nodes impacted by the "next" link failure (Min-Sinks). These optimization criteria differ from those proposed in the literature. For example, most previous work [21, 29, 49, 57, 65] uses optimization criteria specified over a *single* multicast tree, while we must consider criteria specified across *multiple* multicast trees. Finally,

because the smart grid network is many orders of magnitudes smaller than the Internet [2] and multicast group membership is mostly static in the Smart Grid, we can for the most part avoid the scalability issues of Internet-based solutions [21, 29, 49, 57, 65].

The remainder of this chapter is structured as follows. In the following section (Section 3.2), we provide necessary background on PMU application requirements and OpenFlow. Then, we briefly survey relevant literature (Section 3.3). We outline proposed research in Section 3.4: section 3.4.3 details our research thus far on link-failure detection in OpenFlow, and in section 3.4.5, we outline our algorithms for computing backup multicast trees. Our treatment here is necessarily brief, but we indicate work completed thus far as well as proposed future work. Section 3.6 concludes this chapter with a summary of our proposed research and timeline for future work.

3.2 Background

3.2.1 PMU Applications and Their QoS Requirements

The QoS requirements of several PMU applications planned to be deployed on power grids worldwide are presented in Table 3.1, based on [8, 20]. We refer the reader to the actual documents for a description of each PMU application. The end-to-end (E2E) delay requirement is at the *per-packet* level, as advocated by Bakken et al. [8].

NASPI defines five service classes (A-E) for Smart Grid traffic, each designating qualitative requirements for latency, availability, accuracy, time alignment, message rate, and path redundancy [8]. At one end of the spectrum, service class A ap-

[2]For example, it is estimated that fewer than 10^4 routers/switches are needed for a smart grid network spanning the *entire* USA, whereas there are about 10^8 routers in the Internet [8].

PMU Application	E2E Delay	Rate (Hz)	NASPI Class
♡ Oscillation Detection	$0.25 - 3$ secs	$10 - 50$	N/A
♡ Frequency Instability	$0.25 - 0.5$ secs	$1 - 50$	N/A
♡ Voltage Instability	$1 - 2$ secs	$1 - 50$	N/A
♡ Line Temp. Monitoring	5 minutes	1	N/A
△ Closed-Loop Control	$8 - 16$ ms	$120 - 720+$	A
△ Direct State Measurement	$5 - 1000+$ ms	$1 - 720+$	B
△ Operator Displays	1000+ ms	$1 - 120$	D
△ Distributed Wide Area Control	$1 - 240$ ms	$1 - 240$	B
△ System Protection	$5 - 50$ ms	$120 - 720+$	A
△ Anti-Islanding	$5 - 50$ ms	$30 - 720+$	A
△ Post Event Analysis	1000+ ms	< 1	E

Table 3.1. PMU applications and their QoS requirements. The ♡ refers to reference [20] and △ to [8].

plications have the most stringent requirements, while service Class E designates applications with the least demanding requirements.

In this work, we focus on PMU applications with the most stringent E2E delay requirements, such as closed-loop control and system protection. In particular, we create a binary classification of data plane traffic: traffic belonging to critical PMU applications and all other traffic.

TODO *state than nothing is published regarding tolerance to packet loss.*

3.2.2 OpenFlow

OpenFlow is an open source framework that cleanly separates the control and data planes, and provides a programmable (and possibly centralized) control framework [48]. All OpenFlow algorithms and protocols are managed by a (logically) centralized controller, while network switches/routers (as their only task) forward packets according to the flow tables installed by the controller. By allowing a more centralized network control and management framework, the OpenFlow architecture avoids the high storage, computation, and management overhead that plague many distributed

network approaches. Our multicast tree repair algorithms benefit from these OpenFlow features (Section 3.4.5).

OpenFlow exposes the flow tables of its switches, allowing the controller to add, remove, and delete flow entries, which determine how switches forward, copy, or drop packets associated with a controller-managed flow. Phrased differently, OpenFlow switches follow a "match and action" paradigm [48], in which each switch *matches* an incoming packet to a flow table table entry and then takes some *action* (e.g., forwards, drops, or copies the packet). Each switch also maintains per-flow statistics (e.g., packet counter, number of bytes received, time the flow was installed) that can can be queried by the controller. In summary, OpenFlow provides a flexible framework for *in-network* packet loss detection as demonstrated by our detection algorithms (Section 3.4.3).

OpenFlow switches can support a limited number of flow entries because they rely on expensive TCAM memory to perform wildcard matching. For example, the HP5406zl switch supports approximately 1500 OpenFlow rules [22] and the NEC PF5820 switch can handle about 750 flow entries [3].

OpenFlow is similar in spirit to past work in Active Networking [58], as both aim to create programmable networks, but is implemented differently. Active Networking puts the smarts *inside* the network: customized smart routers are used to interpret and execute commands (that may modify network state) specified in code-carrying packets. In contrast, OpenFlow moves the network intelligence (i.e., control logic) *outside* of the network and into the controller, while switches become dumb forwarders of data as they simply follow the instructions dictated by the controller.

3.3 Related Work

3.3.1 Smart Grid Communication Architectures

The Gridstat project [3], started in 1999, was one of the first research projects to consider smart grid communication. Our work has benefited from their detailed requirements specification [8].

Gridstat proposes a publish-subscribe architecture for PMU data dissemination. By design, subscription criteria are simple to enable fast forwarding of PMU data (and as a measure towards meeting the low latency requirements of PMU applications). Gridstat separates their system 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.

Although Gridstat has similarities with our work, their project lacks details. For example, no protocol is provided defining communication between the management and data plane. Additionally, there is no explicit indication if the multicast trees are source-based.

In North America, all PMU deployments are overseen by the North American SynchroPhasor Initiative (NASPI) [14]. 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 [14] for more details.

Hopkinson et al [36] propose a Smart Grid communication architecture that handles heterogeneous traffic: traffic with strict timing requirements (e.g., protection

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

systems), periodic traffic with greater tolerance for delay, and aperiodic traffic. They advocate a multi-tier data dissemination architecture: use a technology such as MPLS to make hard bandwidth reservations for critical applications, use Gridstat to handle predictable traffic with less strict delivery requirements, and finally use 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.

3.3.2 Detecting Packet Loss

Most previous work [6, 16, 30] focuses on measuring and detecting packet loss on an end-to-end packet basis. Because PMU applications have small per-packet delay requirements (Section 3.2.1), the time delay between when the loss occurs and when it is detected needs to be small. For this reason, we will investigate detecting lossy links *inside* the network. Additionally, most previous work takes an *active measurement* approach towards detecting lossy links in which probe messages are injected to estimate packet loss. Injecting packets can potentially skew measurements – especially since accurate packet loss estimates require a high sampling probing rate – leading to inaccurate results [10].

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. This makes their algorithm unsuitable for our purposes because PMU application flows are long lasting (running continuously for days, weeks, and even years). For this reason we propose a new algorithm, Failed-Link, that provides in-network packet loss detection for long running active flows (Section 3.4.3).

We note that existing Internet ??? routing ??? algorithms (e.g., OSPF, ISIS, BGP) perform in-network detection of link failure, but not of individual packet loss.

They do so by having routers exchange "keep-alive" or "hello" messages and detect a link failure when these messages or their acknowledgments are lost.

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 Simple Network Management Protocol (SNMP) [10]. This approach is well 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.

3.3.3 Multicast Tree Recovery

TODO (B) *mention the multicast recovery approaches that don't use MPLS?*

Approaches to multicast fault recovery can be broadly divided into two groups: on-demand and preplanned. In keeping with their best-effort philosophy, most Internet-based algorithms compute recovery paths on-demand [21]. Because the Smart Grid is a critical system and its applications have strict delivery requirements, we focus our literature survey instead on preplanned approaches to failure recovery.

To date, most preplanned approaches [21, 29, 49, 57, 65] are implemented (or suggest an implementation) using virtual circuit packet switching, namely, MPLS. Citations [47, 61] are exceptions, as they both consider preplanned recovery for link failures affecting basic IP multicast traffic. For convenience, in the remainder of this section we assume MPLS is the virtual circuit packet switching technology used, realizing that other such technologies could be used in its place.

Cui et [21] define four categories for preplanned multicast path recovery: (a) link protection, (b) path protection, (c) dual-tree protection, and (d) redundant tree protection. With link protection, a backup path is precomputed for each link, connecting the link's end-nodes [57, 65]. For each destination, a path protection algorithm computes a vertex-disjoint path with the original multicast tree path between the source

and destination [65]. The dual-tree approach precomputes a backup tree for each multicast tree. The backup (dual) tree is not required to be node- or link-disjoint with the primary tree but this is desirable [29, 61]. Lastly, a redundant tree is node (link) disjoint from the primary tree, thereby ensuring that any destination remains reachable, either by the primary or redundant tree, if any vertex (edge) is eliminated [49]. This approach requires link and node disjointedness in the network topology.

However, not all previous work fits nicely into this taxonomy. Li et al. [42] and Kodialam et al. [39] compute backup paths for each link in the primary tree connecting the upstream node to each of its downstream nodes in the original primary tree. Our approach (and likewise the work by Tam et al. [61]) does not fall into any of these categories, as we propose a backup tree be computed for each link in each multicast tree.

Optimization Criteria. Our recovery algorithms use different optimization criteria from previous work considered in this document [21, 29, 39, 41, 42, 47, 49, 57, 65]. With one exception [42] (discussed below) 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). In addition, none of these approaches explicitly optimize for the criteria we consider: minimizing control overhead, minimizing the maximum number of flows impacted by the "next" link failure (Min-Flows), and minimizing the maximum number of sink nodes impacted by the "next" link failure (Min-Sinks). Instead, previous work computes backup paths or trees that optimize one of the following criteria: maximize node (link) disjointedness with the primary path [21, 29, 47, 49], minimize bandwidth usage [65], minimize backup bandwidth reservations [39, 41, 42], minimize the number of group members which become disconnected (using either the primary or backup path) after a link failure [57], or minimize path length [62].

In contrast to other related work, Li et al. [42] optimize for criteria spanning several multicast trees: minimizing the *total* bandwidth reserved by backup paths. Their problem formulation assumes that each source and destination flow is associated with a bandwidth reservation. Under their scheme, backup paths are computed with the goal of minimizing both the restoration time and the total backup bandwidth reserved over *all* backup paths. In our research, we also plan to compute backup trees by optimizing for network-wide constraints but we consider criteria other than backup bandwidth reservation.

Implementation Challenges. Each of the protection schemes presented in this section are implemented as distributed algorithms. As a result, each algorithm must navigate an inherent trade-off between high overhead and fast recovery (i.e., the time between when the failure is detected and when the multicast tree is repaired should be small) [21]: preplanned, localized recovery (e.g., fast reroute) is fast but not scalable while on-demand recovery scales well but can be slow due to slow convergence time. Here, overhead refers to (i) per-router state that needs to be stored and managed and (ii) message complexity associated with the distributed computation. Because preplanned MPLS-based approaches are tailored to support Internet-based applications – typically having a large number of multicast groups and dynamic group membership – scalability is key. Some of the preplanned approaches (dual-tree and redundant trees) focus more on scalability [21, 29, 49], while others (link and path protection schemes) optimize for fast recovery [57, 65].

For two reasons, our algorithms avoid these issues of scalability. First, we use a centralized control architecture (OpenFlow) rather than a distributed one. Using OpenFlow's centralized architecture, we precompute and store all backup paths (offline) at the controller. Thus, we avoid the storage and maintenance issues that distributed approaches must address [21, 29, 57, 65]. In other words, OpenFlow allows

our algorithms to bypass the inherent trade-off of high overhead and fast recovery, allowing (for the first time) both fast *and* scalable recovery algorithms. [4]

Second, the Smart Grid operating environment is very different from the Internet-based applications discussed in the literature. Specifically, the Smart Grid is many orders of magnitude smaller that the Internet and Smart Grid multicast group membership is mostly static [8] (a utility company subscribing to a PMU data stream are likely to always want to receive updates from this PMU).

In the context of OpenFlow, Kotani et al. [40] propose a clever approach for fast switching between IP multicast trees. For reliability purposes, they propose that each multicast group have two multicast trees, a primary tree and a backup tree. [5] 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. To do so, the root node writes the primary tree ID in each packet header [6] , where it is matched and processed by each switch along the primary MT.

After a link failure, the root node switches to use the backup tree by writing the backup tree ID in each packet header, where it is matched and forwarded by a flow entry at each switch along the backup MT. Under their approach for failure recovery, only the root node needs to be signaled by the controller to activate a backup tree. This enables fast switching between the primary and backup MTs. We plan

---

[4] In the case where the controller is unable to manage router and backup path state, we can simply provision more servers to store precomputed paths. Such a situation is unlikely, considering: the encouraging scalability results reported using a centralized Ethane controller [17] (Ethane is a precursor to OpenFlow that also separates the control and data planes), the Smart Grid is many orders of magnitude smaller that the Internet, and multicast group membership is mostly static in the Smart Grid [8].

[5] The approach in [40] for computing MTs and backup MTs is basic, as this is not the emphasis of the paper. The primary and backup tree are both computed using Dijkstra's algorithm [25]. However, the backup MT is computed over a modified version of the original network, where the link weight of each link in the primary tree is set to infinity, thereby producing link-disjoint trees when possible.

[6] Specifically, the tree ID is written in the destination address field of each packet.

to explore this approach as an alternative to the one described in this document: installing backup MTs *after* the link failure is detected.

TODO A: *something about how the local repairs may eventually lead to a poor overall tree? Localized for speed, may not be relevant with a centralized controller, more so a concern for distributed recovery algorithms.*

TODO A: *ultimately, each of these algorithms requires changes to routers/switches themselves ==¿ need OpenFlow.*

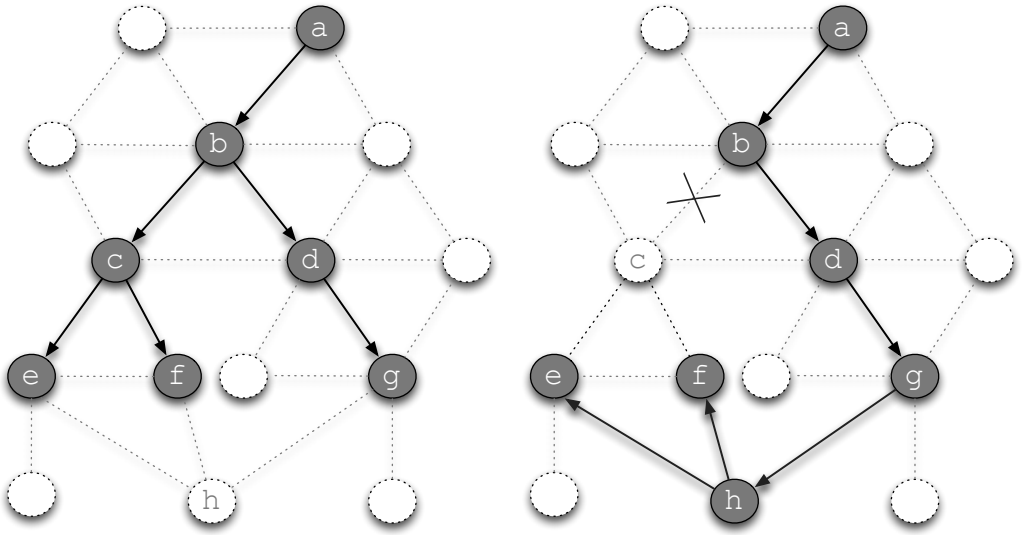
3.4 Proposed Research

In this section, we present an example problem scenario (Section 3.4.1), which we reference to explain: our link failure detection algorithm (Section 3.4.3), steps to uninstall trees that become disconnected after a link failure (Section 3.4.4), steps to install backup multicast trees (Section 3.4.4), and our algorithms for computing backup multicast trees (Section 3.4.5).

3.4.1 Preliminaries

3.4.1.1 Example Scenario

Figure 3.1 depicts a scenario where a single link, (b, c), in a multicast tree fails. Figure 3.1(a) shows a multicast tree rooted at a with leaf nodes (i.e., data sinks) $\{e, f, g\}$. a sends PMU data at a fixed rate and each data sink specifies a per-packet delay requirement. The multicast tree in Figure 3.1(a) uses link (b, c), which we assume fails between the time the snapshots in Figure 3.1(a) and Figure 3.1(b) are taken. When (b, c) fails, it prevents e and f from receiving any packets until the multicast tree is repaired, leaving e and f's per-packet delay requirements unsatisfied. Figure 3.1(b) shows a backup multicast tree installed after (b, c) fails. Notice that the backup tree does not contain any paths using the failed link, (b, c), and has a path between the root (a) and each data sink $(\{e, f, g\})$. In the coming sections we



(a) Before link (b, c) fails. (b) After link (b, c) fails.

Figure 3.1. Example used in Section 3.4.2. The shaded nodes are members of the source-based multicast tree rooted at a. The lightly shaded nodes are not a part of the multicast tree.

present algorithms that allow multicast trees, such as the one shown in Figure 3.1(a), to recover from link failures by installing backup multicast trees, similar to the one in Figure 3.1(a).

3.4.1.2 General Problem Scenario and Basic Notation

Before presenting our algorithms, we first provide a more general problem scenario than the one in Figure 3.1 and introduce some basic notation. We consider a network of nodes modeled as an undirected graph $G = (V, E)$. There are three types of nodes: nodes 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 $m > 1$ source-based multicast trees to disseminate PMU data. Let $T = \{T_1, T_2, \ldots T_m\}$, such that each $T_i = (V_i, E_i) \in T$ is a source-based multicast tree (MT). We assume G only contains MTs in T.

Each PMU node is the source of its own MT and each data sink has a per-packet delay requirement, specified as the maximum tolerable per-packet delay. *Packet delay* between a sender, s, and receiver, r, is the time it takes s to send a packet to r. We consider any packet received beyond the maximum delay threshold as lost. Note that a data sink's per-packet delay requirement is an end-to-end requirement. Before any link fails, we assume that all PMU data is correctly delivered such that each data sink's per-packet delay requirement is satisfied.

Data sent from a single sender and single data sink is called a *unicast flow*. A unicast flow is uniquely defined by a four-tuple of source address, source port, destination address, and destination port. We assume that data from a single PMU maps to a single port at the source and, likewise, a unique port at the destination.

Because the network supports multicast communication, *multicast flows* (as opposed to unicast flows) are used inside the network. Informally, a multicast flow contains multiple unicast flows and only sends a single packet across a link. We use notation $s, \{d_1, d_2, ...d_k\}$ to refer to a multicast flow containing k unicast flows with source, s, and data sinks $d_1, d_2, ...d_k$. The unicast flows are between s and each $d_1, d_2, ...d_k$. Each multicast flow, $f = s, \{d_1, d_2, ...d_k\}$, has k end-to-end per-packet delay requirements, one for each of f's data sinks. Let F be the set of all multicast flows in G.

We define multicast flows inductively using $T_i \in T$. Starting at the source/root, s, T_i has a multicast flow for each of s's outgoing links $(s, x) \in T_i$. For link (s, x) between s and its one-hop neighbor x, the multicast flow contains all T_i unicast flows that traverse (s, x). For example, the Figure 3.1(a) multicast tree has a single multicast flow $(a, \{e, f, g\})$ at a. Only a single multicast flow is needed because a has only one outgoing link in its multicast tree.

At internal T_i nodes, additional multicast flows are instantiated when T_i branches. Internal node $v \in T_i$, instantiates a new multicast flow for each of v's outgoing links

(except for the link connecting v with its parent node in T_i). For outgoing link $(v, u) \in T_i$, v's newly instantiated multicast flow represents T_i's unicast flows that traverse (v, u). In Figure 3.1(a), the $a, \{e, f, g\}$ flow splits when the tree branches at b into multicast flows $a, \{e, f\}$ and $a, \{g\}$. Likewise, multicast flow $a, \{e, f\}$ splits into multicast flows $a, \{e\}$ and $a, \{g\}$ at c.

In keeping with its role as a general framework providing necessary services for programmable networks, OpenFlow does not explicitly provide an implementation for multicast and multicast flows. Our initial plan was to use the group table abstraction described in the OpenFlow 1.1 specification [56] to implement multicast but, unfortunately, as of the writing of this paper, this feature is not yet supported by the POX controller [7] used to implement our algorithms and the Mininet [8] emulator used in our simulations. Instead, we assign a multicast IP address to each multicast group and use this abstraction to setup the flow tables at the multicast tree switches. Because multicast group membership is static in our power grid application (Section 3.2.1, 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.

As with any unicast flow, each switch matches a multicast packet using the `(src_ip,dst_ip)` tuple. If the multicast tree branches, the switch copies the packet to be sent out along each of the switch's outgoing links in the multicast tree. This how the concept of "splitting" a flow (introduced in the previous paragraph) is implemented. If a switch is adjacent to a downstream host in the multicast group, the switch rewrites the destination layer 2 and 3 addresses to those of its adjacent downstream hosts (these fields were previously populated with the multicast addresses).

---

[7] `https://github.com/noxrepo/pox`

[8] `http://mininet.org/`

Another way to implement multicast in OpenFlow is to leverage existing IP multicast protocols as detailed by Kotani et al. [40]. In this approach, the controller assigns a unique group ID to each multicast tree and creates a group table entry, that uses the group ID, at each switch along the multicast tree. Meanwhile, the sender and its first-hop switch use IGMP to set up and manage the controller-generated group IDs. Finally, the sender embeds the group ID in each multicast packet's destination field, allowing for each switch in the multicast tree to identify and forward multicast packets appropriately.

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*. We call the current lossy or faulty link, ℓ. When ℓ fails, all sink nodes corresponding to any multicast flow that traverses ℓ no longer receive packets from the source. As a result, the per-packet delay requirements of each these data sinks is not met. We refer to these data sinks, multicast flows, and the MT associated with each such flow as *directly affected*. In Figure 3.1, $a, \{e, f\}$ is directly affected by (b, c) failing, along with data sinks e and f.

3.4.2 Overview of Our Recovery Solutions and Section Outline

We propose an algorithm, APPLESEED, that is run at the OpenFlow controller to make multicast trees robust to link failures by monitoring and detecting failed links, precomputing backup multicast trees, and installing backup multicast trees after a link failure. [9] As input, APPLESEED is given an undirected graph containing OpenFlow switches; the set of all multicast trees (T); the set of all active multicast flows (F); the length of each sampling window, w, used to monitor links and specified in units of time; and, for each multicast flow, a packet loss condition for each link the flow traverses. For now, we restrict packet loss conditions to be threshold-based that

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

indicate the maximum number of packets that can be lost over w time units. The output of APPLESEED is a new set of precomputed backup multicast trees installed in the network and a set of uninstalled multicast trees. All $T_i \in T$ that use the failed link are uninstalled and we call each such T_i a *failed multicast tree*.

We define a *backup multicast tree* for link ℓ and $T_i \in T$ as a multicast tree that has a path between the source, $s \in T_i$, and each data sink $d \in T_i$ that: (a) connects s and d but does not traverse ℓ and (b) satisfies d's per-packet delay requirements. We refer to any multicast tree that satisfies these conditions for ℓ a *backup multicast tree for* ℓ or backup MT for short. In Figure 3.1(b), notice that the installed backup tree has paths $a \rightarrow b \rightarrow d \rightarrow g \rightarrow h \rightarrow \{e, f\}$ connecting a with $\{e, f, g\}$ after (b, c) fails.

APPLESEED divides into three parts:

1. Monitor to detect link failure (e.g., (b, c) in Figure 3.1).

2. Uninstall all trees using the failed link (i.e., failed trees) and install a precomputed backup multicast tree for each uninstalled tree. For each data sink that was disconnected from the root because of the link failure, the backup tree should use a path that routes around the failed link. Note that the newly installed tree will likely require changes to several switches upstream from ℓ, in addition to those at the upstream and downstream ends of link ℓ. Recall in the Figure 3.1 example, data sinks $\{e, f\}$ are reconnected with a in the installed backup tree.

3. Part (2) triggers the computation of a new backup tree for each (backup) tree installed in (2).

APPLESEED uses an OpenFlow-based subroutine (FAILED-LINK) for part (1) and is presented in Section 3.4.3. For part (2), we briefly describe APPLESEED's steps to uninstall and install multicast trees in Section 3.4.4. In Section 3.4.5, we address part (3) by proposing a set of algorithms that compute backup multicast trees.

3.4.3 Link Failure Detection using OpenFlow

In this section, we propose a simple algorithm (FAILED-LINK), used by APPLESEED, that monitors links *inside* the network to detect any packet loss. To help explain FAILED-LINK, we use the example scenario from Section 3.4.1 and refer to a generic multicast tree with an upstream node, u, and downstream node, d.

FAILED-LINK is run at the OpenFlow controller and provides accurate packet loss measurements that are the basis for identifying lossy links. Informally, a lossy link is one that fails to meet the packet loss conditions specified by the controller. We refer to such a link as *failed*. Although APPLESEED is ultimately concerned with meeting the per-packet *delay* requirements of PMU applications, we use packet loss (as opposed to delay) as an indicator for a failed link because OpenFlow provides no native support for timers.

FAILED-LINK has the same input as APPLESEED, specified in Section 3.4.2. The output of FAILED-LINK is any link that has lost packets not meeting the packet loss condition of any multicast flow traversing the link. In the remainder of this document, we assume all flows are multicast and just use *flow* to refer to a multicast flow, unless otherwise specified.

Recall from Section 3.2.2 that each OpenFlow switch maintains a flow table, where each entry contains a match rule (i.e., an expression defined over the packet header fields used to match incoming packets) and action (e.g., "send packet out port 2"). For each packet that arrives at an OpenFlow switch, it is first matched to a flow entry, e, based on the packet's header fields; then e's packet counter is incremented; and, lastly, e's action is executed on the packet. [10] FAILED-LINK uses these packet counter values to compute per-flow packet loss between switches over w time units.

[10]Not all switches are necessarily OpenFlow-enabled. In fact, we anticipate that in practice many switches will not support OpenFlow. FAILED-LINK still works such scenarios, as long as the packet counts are taken at OpenFlow switches. For ease of presentation, this section assumes all switches are OpenFlow-enabled.

FAILED-LINK uses the subroutine, PCOUNT, to measure the packet loss between an upstream node (u) and one or more downstream nodes. For simplicity, we assume only a single downstream node, d. PCOUNT does so on a per-flow basis over a specified sampling window, w, where w is the length of time packets are counted at u and d. For each window of length w, PCOUNT computes packet loss for a flow f, that traverses u and d, using the following steps:

1. **At u, tags and counts all f packets**. We assume, before any changes are made, u uses flow entry e to match and forward f packets. PCOUNT creates a new flow entry, e', that is an exact copy of e, except that e' embeds a unique identifier (i.e., the tag) in the packet's VLAN Id field. Let this identifying number be 1111. e' is installed with a higher priority than e. In OpenFlow, each flow entry has a corresponding priority specified upon its installation. Incoming packets are matched against flow entries in priority order, with the first matching entry being used. Thus, setting a higher priority for e' than e, ensures that u writes 1111 in the VLAN Id field of all f packets when e' is installed.

2. **Counts all tagged f packets received at d.** PCOUNT does so by installing a new flow entry at d, e'', that matches packets with VLAN Id equal to 1111.

3. **After w time units, turns tagging off at u.** To do so, PCOUNT simply switches the priority of e' and e at u. [11]

4. **Queries u and d for packet counts.** Specifically, the controller uses the OpenFlow protocol to query u for e''s packet count value and d's packet count value for e''. To ensure that all in-transit packets are considered, PCOUNT waits "long enough" for in-transit packets to reach d, before reading d's packet

[11] Unfortunately, OpenFlow does not allow a flow's priority to be modified. As a workaround, we install a copy of e called e_c. We ensure that e_c is given a higher priority than e'. Finally, we delete flows e.

counter (e.g., time proportional to the average per-packet delay between u and d).

5. **Garbage collection.** As a cleanup step delete e' at u and e'' at d.

6. **Computes packet loss.** The controller computes packet loss by simply subtracting e''s packet count from e'''s.

In practice, PCOUNT executes step (2) before step (1) to ensure that u and d consider the same set of packets.

PCOUNT introduced minimal overhead. At u and at each downstream counting switch, a copy of the flow entry corresponding to f is required. However, these copies only persist during the duration of each PCOUNT interval.

In the Figure 3.1 example, the controller uses FAILED-LINK to measure the packet loss for the $a, \{e, f\}$ flow. We assume for link (b, c) and the $a, \{e, f\}$ flow, FAILED-LINK is given a maximum packet loss threshold of 10 packets over w time units. For each sampling window of w time units, PCOUNT instructs b to tag and count all packets corresponding to $a, \{e, f\}$. At the same time, c is instructed by PCOUNT to count the $a, \{e, f\}$ packets tagged by b. Then, the controller uses the OpenFlow protocol to query the packet counter values for $a, \{e, f\}$ at b and c. When (b, c) fails, the packet counter at c for $a, \{e, f\}$ no longer increments, causing a violation of $a, \{e, f\}$'s packet loss threshold for (b, c).

Self Note: describe how pcount can reduce the number of necessary measurement points

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. [59]. Per-packet consistency ensures that when a network of switches change 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 u and d's packet counters, 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.

FAILED-LINK is fast because it detects link failures inside the network, rather than on an end-to-end basis. We plan to quantify how much faster FAILED-LINK is than end-to-end link failure detection techniques. In addition, FAILED-LINK allows for link failures to be localized, whereas end-to-end techniques may not provide the necessary insight to identify and isolate the faulty link.

Self Note: *e matches using tuple* $(src, dst, VLAN)$

3.4.3.1 Pcount Evaluation

TODO *move this section, possibly the E2E discussion to the related work section.*

Detection using end-to-end measurements. An alternative approach to link failure detection is to use end-to-end probes to infer packet loss rates of individual links. Càceres et al. [16] propose a maximum likelihood estimator for loss rates of internal links based on losses observed by multicast receivers. Their model uses the inherent correlation of packet loss across multicast receivers to improve accuracy of their packet loss estimates. Although impressive accuracy results are reported, the authors' simulations show that about 2000 end-to-end probe messages are required for packet loss estimates to converge on the true underlying packet loss rate [16]. In our problem setting, packet loss needs to be detected over small windows of time and, unfortunately, 2000 messages would correspond to too larger a window of time. For this reason, we deem solutions based on end-to-end measurements a poor match for our problem domain.

7/20/13 writing from grant about plans for evaluation. To date, we have implemented PCOUNT in the POX OpenFlow controller using the Mininet emulator and plan to further evaluate PCOUNT by using our POX-based implementation to

work with real OpenFlow switches. To provide a point of reference, we plan to implement and measure a simple SNMP-based approach for detecting link-level packet loss. We will quantify the error rate of PCOUNT and the SNMP-based approach as a function of the sampling window size. We will also quantify the detection time for PCOUNT and the SNMP-based approach as a function of window size. We believe our measurement study will show PCOUNT provides accurate and fast packet loss detection across all sampling window sizes, making it a superior approach for ultra-reliable data dissemination in smart grid networks

3.4.4 Uninstalling Failed Trees and Installing Backup Trees

After FAILED-LINK detects ℓ's failure, APPLESEED uninstalls all MTs that contain ℓ (i.e., failed trees) and installs a precomputed backup multicast tree for each of these uninstalled tree. Installing backup trees for ℓ, denoted T_ℓ, triggers the computation of new backup trees. A backup MT is needed for each $T_j \in T_\ell$ and for each of T_j's links. Here we only explain how MTs are uninstalled and installed and describe some of side effects of doing so. We postpone explaining how backup trees are computed until Section 3.4.5.

Uninstalling or installing a multicast tree, T_i, is simple with OpenFlow. The controller sends an instruction to each switch in T_i to remove (add) the flow entry corresponding to T_i. Starting at the root and ending at the leaf nodes, flow entries are removed top-down to prevent T_i traffic from being sent while T_i is being uninstalled. In contrast, flow entries are installed bottom-up, starting at the leaf nodes and finishing at the root. This ensures that when the root node begins to disseminate data using T_i that all downstream T_i nodes are equipped with the necessary flow entries to correctly forward T_i traffic.

Increased traffic caused by newly installed backup trees may introduce packet loss to multicast flows that do not traverse ℓ but do use a path shared by at least one switch

in a backup MT. We refer to these flows as *indirectly affected*. Additionally, we refer to any packet or data sink corresponding to an indirectly affected flow as indirectly affected. In our Figure 3.1 example, $a, \{g\}$ is indirectly affected when the backup MT is installed because the backup MT uses paths $a \rightarrow b \rightarrow d \rightarrow g \rightarrow h \rightarrow \{e, f\}$.

3.4.5 Computing Backup Multicast Trees

Here we present a set of algorithms that compute backup MTs. APPLESEED uses these algorithms in two scenarios. First, as a part of system initialization where a set of backup MTs are computed for each network link, l; APPLESEED computes a single backup MT for each MT that would be directly affected by l's failure.

Second, APPLESEED triggers the execution of backup tree computations after backup trees, T_ℓ, are installed in response to the most recent link failure, ℓ. APPLESEED reruns the entire computation (as opposed to only computing backup MTs for the newly installed T_ℓ trees): for each network link, l, APPLESEED computes a backup MT for each MT that would be affected if l failed. APPLESEED recomputes *all* backup MTs because installing the T_ℓ MTs changes how flows are distributed and processed inside the network and, as a result, may adversely affect flows processed by MTs other than those in T_ℓ. In other words, backup MTs computed before ℓ fails may become stale when the T_ℓ MTs are installed since the algorithms used to compute them considered an old network state in their optimization. *We hypothesize that recomputing* all *backup MTs each time a single link fails yields a more survivable data dissemination network than only recomputing backup MTs for newly installed backup MTs installed in response to the most recent link failure.*

APPLESEED uses one of the following backup tree computation algorithms: MIN-FLOWS, MIN-SINKS, or MIN-CONTROL. Next, we present initial sketches of each of these algorithms.

3.4.5.1 Min-Flows Algorithm

Before outlining Min-Flows, we provide a brief refresher on multicast flows, as defined in Section 3.4.1.2. There we stated that a multicast flow contains multiple unicast flows and only sends a single packet across a link. We used $s, \{d_1, d_2, ...d_k\}$ to refer to a multicast flow containing k unicast flows with source, s, and data sinks $d_1, d_2, ...d_k$, such that the unicast flows are between s and each $d_1, d_2, ...d_k$. Finally, we specified that each multicast flow, $f = s, \{d_1, d_2, ...d_k\}$, has k end-to-end per-packet delay requirements, one for each of f's data sinks.

Min-Flows Algorithm

- Input: (G, T, F, l), such that l is a link in G and each $f \in F$ specifies the maximum per-packet delay it can tolerate.
- Output: A backup multicast tree for each MT using l. This set of backup MTs, T_l, are computed such that if all MTs in T_l were installed:
    1. Across all network links, the maximum number of flows traversing a single link is minimized.
    2. The end-to-end per-packet delay requirement of all multicast flows continues to be met.

Min-Flows protects against the worst case by minimizing the maximum number of flows affected by the next link failure. Intuitively, Min-Flows aims to avoid the scenario in which many flows all traverse the same link, ℓ, as a result of installing backup MTs for the current link failure. If this were to occur, many flows would be directly affected when ℓ fails.

Min-Flows Example. To provide intuition for Min-Flows, consider an augmented version of the example from Figure 3.1, in which the Figure 3.1 graph is a subgraph of a larger graph, $G_1 = (V_1, E_1)$. We assume the following initial system state: no E_1 links have failed; 10 multicast flows traverse each E_1 link; 9 MTs, in

addition to the MT rooted at a, use link (b, c); and all end-to-end delay requirements are met. We assume that (b, c) is the the first link to fail and that (b, d) fails next, but only after the precomputed backup MTs for (b, c) have been installed.

We compare the backup MTs MIN-FLOWS computes for link (b, c) with those computed by a simple Steiner-tree-based approach, STRAW-MAN-1:

- MIN-FLOWS: Let MIN-FLOWS compute backup MTs for (b, c) such that one of the 10 MTs uses (b, d), while the other 9 backup MTs do not use (b, d).
- STRAW-MAN-1: STRAW-MAN-1 computes backup MTs such that each MT is a Steiner tree approximation over $G_1' = (V_1, (E_1 \setminus \{(b, c)\}))$ with same root node and leaf nodes as in the original MT. In this example, we assume that all of STRAW-MAN-1's backup MTs for (b, c) use (b, d).

Now, consider the following sequence of events: first link (b, c) fails, then the backup MTs for (b, c) are installed, and, lastly, link (b, d) fails. Using MIN-FLOWS's backup MTs for (b, c), only 11 flows are affected when (b, d) fails. However, 20 flows are directly impacted using STRAW-MAN-1's backup MTs for (b, c) because all 10 of these MTs use (b, d), making 20 total MTs (and flows) that use (b, d) (recall that we assumed the initial state of the system was such that 10 multicast flows traverse each link in G_1).

(4/1/13) May want to consider minimizing the variance in the number of flows traversing a single link.

MIN-FLOWS is similar to the multicast packing problem [19] that aims to compute a multicast tree for each multicast group such that the maximum link sharing is minimized, while keeping the size of each multicast tree close to optimum.

Longer version of the same writing – In order to bound the size of each multicast tree, which might grow prohibitively large if only minimizing link sharing is

considered, their optimization requires that each multicast tree must be within some delta of the multicast group's corresponding Steiner tree.

No formal proof exists showing that the multicast packing problem is intractable but informal commentary suggesting that this is the case can be found in the literature. Chen et al. [19] state that the multicast packing problem is more difficult than computing the Steiner tree for each multicast group separately, which is known to be an NP-hard problem, because the min-max nature of the objective function. The authors refer the reader to a report [12] describing a closely related mixed-integer multicommodity flow problem, suggesting that this problem might be used to formally establish that the multicast packing problem is intractable.

Lu and Zhang [46] consider a problem similar to the multicast packing problem called the multicast congestion problem. The goal of the multicast congestion problem is to compute a multicast tree for each multicast group such that the maximum edge congestion – also defined as the number of multicast trees using the edge – is minimized. Unlike the multicast packing problem, no constraints are placed on the size of the multicast tree. The authors informally state that the multicast congestion problem is NP-hard because it is a generalization of the problem of finding edge-disjoint shortest paths for source destination pairs, which is known to be NP-hard.

Chen et al. [19] propose a simple, yet effective heuristic for multicast tree packing:

1. As a preprocessing step, a Steiner tree is computed for each multicast group independently. Let OPT^i denote the Steiner tree for multicast group i.

2. Then the congestion of each edge, defined as the number of multicast tree using the edge, is computed.

3. Iteratively, consider each multicast tree, T_i, using the most congested edge, ℓ, with congestion equal to Z. Try to rebuild T_i such that the new multicast tree, T_i', yields a congestion level less than Z and the size of T_i' does not exceed a

threshold, α. Specifically, the ratio of T_i' to OPT^i cannot exceed $\alpha \geq 1$. The rebuild function works as follows:

(a) Create an auxiliary graph, G', containing the links from G that have congestion level less than $Z - 2$.

(b) Consider the cut obtained from removing the most congested link, e, from G'. If the cut contains a link e', with congestion level at most $Z - 2$, then replace link $e \in T_i'$ with e'.

4. If the congestion levels have been updated, go to step (2).

This heuristic algorithm can be easily used to solve MIN-FLOWS, requiring only three simple changes. First, we remove ℓ from the input graph. Let $\overline{T_\ell}$ represent all multicast trees that do not use ℓ. Second, for each link in $\overline{T_\ell}$ we update the congestion count to reflect the number of times the link is used a tree in $\overline{T_\ell}$. Finally, the algorithm should only consider multicast groups corresponding to each T_ℓ, leaving all other multicast trees (i.e., $\overline{T_\ell}$) untouched.

Chen et al. [19] derive lower bounds for the multicast packing problem based on dividing all network nodes into two sets and measuring the ratio of multicast traffic passing along links connecting the two sets. Tighter lower bounds are found by finding a multicut that divides the maximum number of multicast groups.

3.4.5.2 Min-Sinks Algorithm

Next, we present the MIN-SINKS algorithm. MIN-SINKS is closely related to but different from MIN-FLOWS. Both algorithms seek to minimize the disruption of future link failures but optimize different criteria towards achieving this goal.

Before specifying the input and output of MIN-SINKS, we remind the reader of previously specified notation: F is the set of all network flows and $f = s, \{d_1, d_2, ...d_k\}$

denotes a multicast flow with k data sinks, $d_1, d_2, ...d_k$. As new notation, we define s_l for each link, l:

$$s_l = \sum_{\forall f \in F: f \text{ traverses } l} |k| \tag{3.1}$$

MIN-SINKS ALGORITHM

- Input: (G, T, F, l), such that l is a link in G and each $f \in F$ specifies the maximum per-packet delay it can tolerate.
- Output: A backup multicast tree for each MT using l. This set of backup MTs, T_l, are computed such that if all MTs in T_l were installed:
  1. Across all network links, the maximum s_l value is minimized. [12]
  2. The end-to-end per-packet delay requirement of all multicast flows continues to be met.

Similar to MIN-FLOWS, MIN-SINKS minimizes the worst case impact of the "next" link failure. However with MIN-SINKS, impact is measured in terms of number of affected downstream sink nodes rather than considering the number of affected flows (as with MIN-FLOWS).

Min-Sinks Example. Consider again the MIN-FLOWS example scenario (Section 3.4.5.1), where we assumed that 10 multicast flows traverse each link in G_1 before any link fails. Now we also assume the initial state of the system is such that the installed MTs ensure that for each link, l, multicast flows traverse l so that there are 20 downstream sink nodes from l (i.e., $s_l = 20$).

When (b, c) fails, MIN-SINKS's backup MTs ensure that each link has 22 downstream sink nodes. An alternative approach to MIN-SINKS, STRAW-MAN-2, com-

[12] That is, $\min_{\forall l \in E}(\max(s_l))$.

putes backup MTs for (b, c) such that (b, d) handles multicast flows leading to 40 downstream sink nodes for (b, d). If (b, d) fails after the backup MTs for (b, c) are installed, only 22 sink nodes are directly impacted using MIN-SINKS, while 40 data sinks are impacted with STRAW-MAN-2's backup MTs installed.

3.4.5.3 Min-Control Algorithm

Lastly, we outline the MIN-CONTROL algorithm. In effort to provide fast and scalable recovery, MIN-CONTROL minimizes the control overhead required to install backup MTs. We define the control overhead as the number of new flow entries that need to be installed to activate the backup MTs. Consider a multicast tree, T_ℓ and it's backup T'_ℓ. A new flow entry must be installed at each upstream node, u, using an adjacent link, $l \in T'_\ell \setminus T_\ell$. Because the direction of the link matters in a multicast tree are directed, a flow entry needs to be installed at u for each adjacent link $l \in T'_\ell \setminus T_\ell$. For example, if switch $u \in T'_\ell$ has adjacent links $l_1, l_2 \in T'_\ell \setminus T_\ell$ we count the control overhead as number of as two rather than one. [13]

Intuitively, minimizing the control overhead yields fast recovery when backup MTs are installed after a link failure is detected, since few control messages need to be sent from the controller to switches. In the case where backup MTs are preinstalled (i.e., installed before a link failure occurs), MIN-CONTROL minimizes the amount of control state preinstalled. This is important because OpenFlow switches can only store a limited number of flow entries (see Section 3.2.2).

MIN-CONTROL ALGORITHM

- Input: (G, T, F, l), such that l is a link in G and each $f \in F$ specifies the maximum per-packet delay it can tolerate.

[13] OpenFlow requires a separate message be sent for each new flow entry to be installed.

- Output: A backup multicast tree for each MT using l. This set of backup MTs, T_l, are computed such that

  1. The *total* control overhead required to install all MTs in T_l is minimized.
  2. The end-to-end per-packet delay requirement of all multicast flows continues to be met if all MTs in T_l were installed.

3.4.6 System Initialization

OUT OF SCOPE: WILL ONLY COMMENT ON POSSIBLE APPROACHES!

3/21/13 NOTES:

- Need to compute a set of source-based MTs for source nodes $s_1, s_2, \ldots, s_m$ and sets of sink nodes $D_1, D_2, \ldots, D_m$ where $D_i = \{d_1, d_2, \ldots, d_k\}$ and s_j corresponds to D_j.
- Modified Min-Flows (or Primary-Min-Flow) has the goal to compute m primary MTs that minimizes the maximum number of flows traversing a single $l \in G$. The backup MTs are then computed.
- This modified, global Min-Flows (described in the previous bullet) can be the "global" recompute option in our simulations to be used after a link failure occurs. Actually not sure if this makes any sense, because (1) we cannot recompute all primary MTs and (2) we compute backup MTs on a per-link and per-MT basis.

A question not addressed is how are the initial set of multicast trees and their backup MTs computed? Here are two potential approaches to this problem:

- Option 1: First compute all primary MTs, and then compute backup MTs. Under this approach, the backup MTs can be computed using any of the backup MT algorithms from Section 3.4.5. However, we have no proposed an algorithm to compute the initial set of primary MTs. We can modify Min-Flows and Min-Sinks slightly to provide a similar problem formulation. In particular, instead of computing a (backup) MT for each MT using l, the modified problem formulation specifies that an MT is computed for each pair of source and set of sinks.

  Likely prefer this approach because will compute "good" primary MTs first and since primary MTs are the majority of the operation, intuitively it makes sense to focus on optimizing for good primary MTs.

- Option 2: A different approach is to compute the primary MT and primary MTs at the same time. The intuition for doing so is that this can allow for a primary MT to constructed that provides the opportunity to build good backup MTs. This approach is complicated by the fact that each MT with $|L|$ links has $|L|$ backup MTs. Therefore the optimization is over at least $|L| + 1$ MTs. In addition, this approach, as discussed, does not optimize using any criteria specified over any other primary MTs. This could lead to a poor set of primary MTs (e.g., many MTs traverse the same link).

  better suited when just one backup MT?

3.4.7 How to Efficiently Install/Activate Pre-computed Backup MTs?

Once backup MTs have been computed, it is an open question as to how to efficiently install pre-computed backup MTs. Here we briefly highlight three possible approaches.

1. Follow the approach specified in Section 3.4.4: install backup MTs *after* a link failure. This approach may be slow, as it requires each switch in the backup

MT to be signaled separately in order to install and activate the backup MTs. However, a benefit is that no extra state (i.e., flow table entries of backup MTs) is installed in the network, thereby reducing the necessary space overhead at each switch.

2. Use a technique similar to the one proposed by Kotani et al. [40] in which entire backup MTs are pre-installed in the network and activated by the root node of each affected MT. This approach is fast because only the root needs to be signaled to activate the backup MTs. On the other hand, this approach is wasteful in terms of space, as each switch must store a flow entry for each backup MT is belongs to. This can be problematic because, for each network link, l, we precompute a backup MT for each MT using l. This amounts to computing a backup MT for each link of each MT. Specifically, if we let $T = \{T_1, T_2, \ldots T_m\}$ be the set of all MTs, where each $T_i = (V_i, E_i) \in T$ is a source-based MT, then we need to pre-compute $\sum_{\forall T_i \in T} |E_i|$ many backup MTs.

3. Hybrid of (1) and (2). Under this approach, we reuse the flow entries of primary MTs as much as possible and only pre-install the non-overlapping sections of the backup MTs. For example, consider primary MT, $T_i = (V_i, E_i)$, and backup MT, $T_i' = (V_i', E_i')$. Let the set of shared link between the primary and backup MT be $O_i = E_i \cap E_i'\ T_i'$. For T_i' we only pre-install flow entries for portions of the MT using $(E_i -' O_i)$ and reuse the flow entries from the primary MT that use O_i. [14]

   This approach requires less state to be pre-installed than approach (2) and incurs less signaling overhead (measured from the time after a link fails) to install and activate backup MTs than approach (1).

[14] Not sure of the details of "reusing" the O_i flow entries and linking this with pre-installed parts of the backup MT $(E_i -' O_i)$.

The effectiveness of these approaches will be determined by the following open questions. First, is the time overhead of signaling a switch from the controller significant? If no, then the first option becomes attractive. If so, then we may consider minimizing control signaling overhead as in option two and MIN-CONTROL. Second, is the space overhead resulting from storing flow entries for pre-installed backup MTs significant? This depends both on the hardware capabilities of a switch and the number of MTs (both primary and backup) required by the network.

3.4.8 Multiple Link Failures

OUT OF SCOPE: WILL ONLY COMMENT ON POSSIBLE APPROACHES!

Handling multiple simultaneous link failures

- The backup MT algorithm would need to be modified to take a vector of links.
- It's possible that we could face combinatorial explosion in trying to precompute backup MTs for all simultaneous link failures. But maybe this is not an issue because we use a separate (centralized) control plane to compute and store backup MTs and due to the smaller scale of grid?
- Out of scope but useful to have some commentary on this topic.

3.4.9 Node Failures

Handling failure of node v:

- Can equate v's failure to (1) the simultaneous failure of all of v's adjacent links and (2) The set of nodes in the network becomes $V - \{v\}$.

- Need to solve the problem of multiple simultaneous link failures first, and then possibly modify this algorithm to accept the new topology (i.e., the one without v). However, modifying said algorithm may not be the best approach, as devising an entirely new algorithm may yield better results.
- Out of scope but useful to have some commentary on this topic.

3.4.10 Evaluation Ideas

- Should compare vs having duplicate MTs as Gridstat proposes. Faster recovery but more traffic?

3.5 My Notes on Actual Algorithms

3.5.1 Failed-Link details

```
def single_switch_count_query(switch,window_size=30):
    """ this function needs to be able specify which switch is being
        queried.  option doesn't seem available w/ Frenetic """

    return (Select(counts) *
            Every(window_size))

def pairwise_packet_count_query(switch1,switch2,window_size=30):
    """ won't work because the counts are not read synchronously """

    single_switch_count_query(switch1,window_size)
    single_switch_count_query(switch2,window_size)
```

Since we are using Frenetic that has a declarative query language, let's divide the task into two parts: (1) how to specify the query and (2) translating the query into OpenFlow commands (compilation).

What needs to be specified:

- The two switches.
- Length of sampling window.
- A synchronous read of packet counts is desired.
- ?? Which switch is upstream and which is downstream ?? probably makes sense to have this, especially if links are bidirectional (i.e., may not be implicit).
- Optionally, how frequent sampling should take place (e.g. every minute, hour, etc)

After parsing the query, the Frenetic compiler would need to translate the query into a set of OpenFlow commands.

TODO *address query cost. Frenetic defines query cost as a function of the number of packets that need to be diverted to the controller.*

3.5.2 Min-Flows Algorithm

Algorithm 3.4.5.3 computes a single backup MT at-a-time for each primary tree using ℓ. It ensures that no backup MT uses ℓ nor the maximum loaded link, $\hat{\ell}$, by "removing" these links from G before initiating a multicast tree computation. [15] We make no assumptions about the algorithm to compute a multicast tree, rather we allow this to be any algorithm for multicast tree computation (e.g., a Steiner tree approximation).

[15] ℓ and $\hat{\ell}$ are not actually removed from G, rather Algorithm 3.4.5.3 works with a copy of G that is identical to G except that the copy does not contain links ℓ and $\hat{\ell}$.

Depending on the order that backup MTs are computed the $\hat{\ell}$ may change. Because Algorithm 3.4.5.3 computes backup MTs in an arbitrary order it possible that some order of computing backup MTs causes the final $\hat{\ell}$ to be larger than its value before the backup MT computations started.

Notes on Algorithm 3.4.5.3

- Q: Does the order matter?

  A: Unlikely, but can occur if many links process a similar number of flows to $\hat{\ell}$ (i.e., many links are overloaded and there is no choice but to route via one of these links).

- Is greedy because it does not recompute any previously computed backup MTs.

- does a single pass in computing the backup MTs.

- Summary Attempt 1: The idea is to create a copy of G, G', and remove ℓ from G' to ensure that no backup MT uses ℓ. In each iteration, we (re)compute and remove the link with the maximum number of flows traversing it, based on the set of primary MTs and newly installed backup MTs.

- Summary Attempt 1: To compute a backup MT, Algorithm 3.4.5.3 first determines the link with the most flows traversing it, $\hat{\ell}$, over a modified graph (G') and a modified set of MTs (T'). G' is identical to G except that ℓ and $e\hat{l}l$ are removed. This ensures that neither link is used when computed the backup MT. T' contains the set of all primary MTs not using ℓ along with any backup MTs already computed for ℓ. In each iteration, $\hat{\ell}$, is (re)computed because this value may change as a result of installing backup MTs in G' in previous iterations.

3.6 Chapter Conclusion and Future Work

In this final technical chapter, we considered the problem of recovery from link failures in a smart grid communication network. We presented the outline for an algorithm that uses OpenFlow to detect packet loss inside the network of switches and a set of algorithms to precompute backup multicast trees to be installed after a link fails. Our algorithms for computing backup multicast trees differ from those in the literature because each algorithm optimizes for the long-term survivability of the communication network.

For each algorithm proposed in the chapter (Appleseed, Failed-Link, Pcount, Min-Flows, Min-Sinks, Min-Control), we plan to supplement our basic algorithm description with a detailed specification of the algorithm steps, implement the algorithm in OpenFlow, characterize its complexity, and evaluate its implementation through simulations.

Time permitting, we plan to propose algorithms for computing backup multicast trees that consider indirectly affected flows. Recall from Section 3.4.4 that indirectly affected flows are ones using a path that shares at least one link contained in a backup tree. As a result, indirectly affected flows may experience packet loss due to increased traffic from redirected flows. We conjecture that an algorithm that minimizes the number of indirectly affected flows reduces the system-wide affects (spatially) of installing backup multicast trees.

A similar algorithm is possible that minimizes the number of indirectly affected data sinks.

Algorithm 3.5.1: Naive (Greedy?) MIN-FLOWS Algorithm

Input: (G, T, F, ℓ), such that l is a link in G and each $f \in F$ specifies the maximum per-packet delay it can tolerate.

Output: A set, T_ℓ^b, containing a backup MT for each MT using ℓ s.t. if all MTs in T_ℓ^b were installed, then across all network links, the max # of flows traversing a single link is minimized.

$G' \leftarrow G$
$E' \leftarrow E \backslash \{\ell\}$; /* remove ℓ from G' */
$T_\ell^p \leftarrow$ all primary MTs using ℓ
$T' \leftarrow T' \backslash \{T_\ell^p\}$; /* remove each primary MT using ℓ */
$T_\ell^b \leftarrow \emptyset$
for each $T_{\ell i}^p \in T_\ell^p$ **do**
    $\hat{\ell} \leftarrow l \in E'$ with maximum number of flows traversing it
    $E' \leftarrow E \backslash \{\hat{\ell}\}$; /* remove max loaded link */
    $T_{\ell i}^b \leftarrow computeMcastTree(G', T', T_{\ell i}^p)$; /* e.g., Steiner Tree */
    $T' \leftarrow T' \cup T_{\ell i}^b$; /* install T_ℓ^b in G' */
    $T_\ell^b \leftarrow T_\ell^b \cup T_{\ell i}^b$
    $E' \leftarrow E' \cup \{\hat{\ell}\}$; /* re-add $\hat{\ell}$ to E' b/c ℓ might diff in next iter */
end
return T_ℓ^b

CHAPTER 4

THESIS TIMELINE AND FUTURE WORK

In this chapter we provide a timeline of planned future work (Section 4.1) and briefly comment on additional topics for research that fall outside the scope of this thesis (Section 4.2).

4.1 Planned Future Work and Timeline

Because the work described in Chapters 1 and 2 is complete, all planned future work is focused on completing the research proposed in Chapter 3. For each algorithm described in Chapter 3 – Appleseed, Failed-Link, Pcount, Min-Flows, Min-Sinks, and Min-Control – we plan to supplement its basic description with a detailed specification of its algorithm steps, implement the algorithm in OpenFlow, and evaluate its implementation. We estimate this requires six months of steady work to complete.

We now provide a more detailed specification of thesis milestones and anticipated dates of completion:

1. Extend my unicast flow-based specification of Failed-Link to work for multicast flows. *(Feb.)*

2. Complete OpenFlow-based implementation for Failed-Link using the POX controller [1]. *(Feb.)*

[1] `https://openflow.stanford.edu/display/ONL/POX+Wiki`

3. Use OpenFlow emulator, Mininet [2], to test and profile FAILED-LINK. *(March)*

4. Provide detailed design and specification of MIN-FLOWS, MIN-SINKS, and MIN-CONTROL. *(March)*

5. Complete OpenFlow-based implementation for MIN-FLOWS, MIN-SINKS, and MIN-CONTROL using POX controller. *(April)*

6. Complexity analysis of MIN-FLOWS, MIN-SINKS, and MIN-CONTROL. *(April)*

7. Run simulations to evaluate MIN-FLOWS, MIN-SINKS, and MIN-CONTROL using Mininet. *(May)*

8. Write conference paper based on Chapter 3 research. *(June)*

Throughout this process, I plan to write the remaining sections of the last technical thesis chapter as dictated by the results derived from each step specified above.

4.2 Future Work Outside the Scope of this Thesis

In this section, we comment on topics for future work that will not be considered as a part of this thesis.

- **Chapter 1**. One challenging problem is to find the worst possible false state a compromised node can inject. Some options include the minimum distance to all nodes (e.g., our choice for false state used in Chapter 1), state that maximizes the effect of the count-to-infinity problem, and false state that contaminates a bottleneck link.

- **Chapter 2**. The success of the greedy algorithms suggests that the IEEE bus systems have special topological characteristics. Finding and investigating

[2] `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet`

these properties could be a fruitful exercise that might provide more insight into why `greedy` and `xvgreedy` yield such encouraging results. Additionally, a valuable contribution would be to implement the integer programming approach proposed by Xu and Abur [66] to solve FullObserve, as this would provide valuable data points to measure the relative performance of `greedy`.

- **Chapter 3**. Because our algorithms for backup multicast tree computation are not fully specified, implemented, nor evaluated, we are unable to comment on any future work related to this chapter.

APPENDIX A

PSEUDO-CODE AND ANALYSIS OF DISTANCE VECTOR RECOVERY ALGORITHMS

A.1 Distance Vector Recovery Algorithm Pseudo-Code

Building on the notation specified in Table 1.1 (from Section 1.2), we define some additional notation here to 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**

APPENDIX B

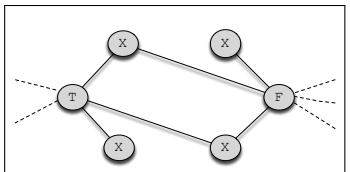
ADDITIONAL PMU PLACEMENT PROOFS

In Chapter 2 we proved that FullObserve (Section 2.3.2), MaxObserve (Section 2.3.3), FullObserve-XV (Section 2.3.4), and MaxObserve-XV (Section 2.3.5) 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 2.3.2 - 2.3.5. Then, we provide pseudo-code and complexity proofs for the approximation algorithms described in Section 2.4. These proofs consider graphs with both zero-injection and injection buses.

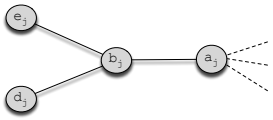
B.1 NP-Completeness Proofs for PMU Placement in Zero-Injection Graphs

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 [15] prove FullObserve is NP-Complete for zero-injection graphs.

Our proofs follow the same strategy outlined in Section 2.3.1: we reduce for P3SAT to show each problem is NP-Complete. Recall that our proofs from Chapter 2 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 2.3.3. Our proof for the theorem below (Theorem B.1) is similar to that for Theorem 2.4.

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 [15], 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 2.2.2 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 [15] 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.

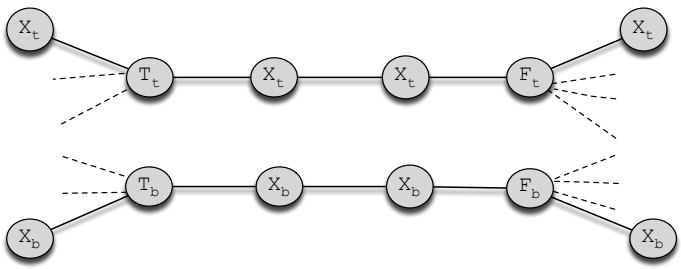


Figure B.2. Variable gadget used in Theorem B.2 proof. The dashed edges are connections to clause gadgets.

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 [15]. We use the proof in [15] 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 2.3.4. The proof for Theorem B.2, below, closely follows the structure of Theorem 2.5'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 [15]. 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 2.3.5. The proof below for Theorem B.3 closely resembles the proof for Theorem 2.7.

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 2.3(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:

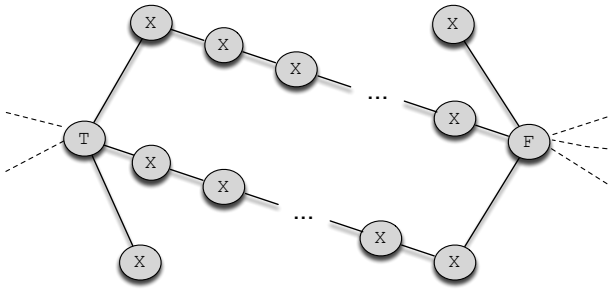
- $s \geq t$: then we get $2s \geq 2s + 2t \Rightarrow t = 0$.
- $s < t$: then we get $2s \geq 4t \Rightarrow s \geq 2t$, and since we assume here $0 \leq s < t$ this leads to a contradiction and so this case cannot occur.

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. □

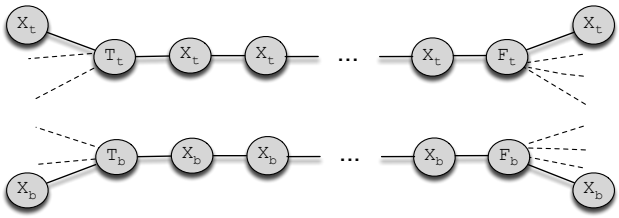
B.1.4 Extending Gadgets to Cover a Range of m and $|V|$ values

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.



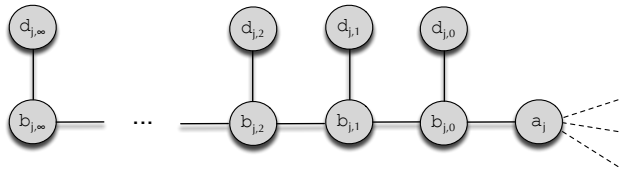
(a) Extended variable gadget used for MaxObserve.



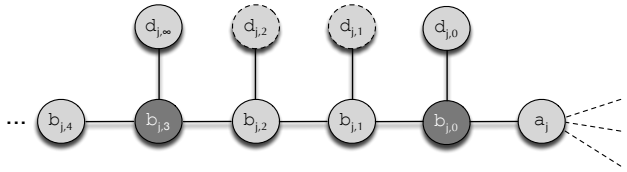
(b) Extended variable gadget used for MaxObserve-XV.

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 2.4 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

```
1: Φ_G ← ∅
2: for k iterations do
3:    maxObserved ← 0
4:    C ← all cross-validated node pairs in (V − Φ_G)
5:    for each {v_1, v_2} ∈ C do
6:       numObserved ← 0
7:       for each u ∈ (Φ_G ∪ {v_1, v_2}) do
8:          add PMU to v_1 and v_2
9:          apply O1 at u and update numObserved
10:      end for
11:      repeat
12:         flag ← False
13:         for each w ∈ (V − (Φ_G ∪ {v_1, v_2})) do
14:            if w ∈ (V_Z ∩ Φ_G^R) and w has 1 unobserved neighbor then
15:               apply O2 at w and update numObserved
16:               flag ← True
17:            end if
18:         end for
19:      until flag = False
20:      if numObserved > maxObserved then
21:         greedyNodes ← {v_1, v_2}
22:         maxObserved ← numObserved
23:      end if
24:   end for
25:   Φ_G ← Φ_G ∪ greedyNodes
26: end for
```

a-time – we are able to directly apply the analysis from Theorem 2.8 in this proof. Therefore, we conclude the complexity of `xvgreedy` is $O(k(d^2n+dn^3)) = O(dkn^3)$. $\square$

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] ProgrammableFlow PF5820 switch. http://www.necam.com/SDN/.

[4] Rocketfuel. http://www.cs.washington.edu/research/networking/rocketfuel/maps/weights/weights-dist.tar.gz.

[5] Aazami, A., and Stilp, M.D. Approximation Algorithms and Hardness for Domination with Propagation. *CoRR abs/0710.2139* (2007).

[6] Almes, G., Kalidindi, S., and Zekauskas, M. A one-way packet loss metric for ippm. Tech. rep., RFC 2680, September, 1999.

[7] Ammann, P., Jajodia, S., and Liu, Peng. Recovery from Malicious Transactions. *IEEE Trans. on Knowl. and Data Eng. 14*, 5 (2002), 1167–1185.

[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] Bienstock, D., and Günlük, O. Computational experience with a difficult mixed integer multicommodity flow problem. *Mathematical Programming 68*, 1-3 (1995), 213–237.

[13] 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.

[14] 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.

[15] Brueni, D. J., and Heath, L. S. The PMU Placement Problem. *SIAM Journal on Discrete Mathematics 19*, 3 (2005), 744–761.

[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] Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review* (2007), vol. 37, ACM, pp. 1–12.

[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] Chen, S., Günlük, O., and Yener, B. The multicast packing problem. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 311–318.

[20] Chenine, M., Zhu, K., and Nordstrom, L. Survey on priorities and communication requirements for pmu-based applications in the nordic region. In *PowerTech, 2009 IEEE Bucharest* (2009), IEEE, pp. 1–8.

[21] 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.

[22] Curtis, Andrew R, Mogul, Jeffrey C, Tourrilhes, Jean, Yalagandula, Praveen, Sharma, Puneet, and Banerjee, Sujata. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 254–265.

[23] 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.

[24] Dijkstra, E., and Scholten, C. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11 (1980).

[25] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[26] Dughmi, S. Submodular functions: Extensions, distributions, and algorithms. a survey. *CoRR abs/0912.0322* (2009).

[27] 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.

[28] 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).

[29] 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.

[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] Garcia-Lunes-Aceves, J. J. Loop-free Routing using Diffusing Computations. *IEEE/ACM Trans. Netw. 1*, 1 (1993), 130–141.

[32] 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.

[33] Gyllstrom, D., Vasudevan, S., Kurose, J., and Miklau, G. Efficient recovery from false state in distributed routing algorithms. In *Networking* (2010), pp. 198–212.

[34] Gyllstrom, D., Vasudevan, S., Kurose, J., and Miklau, G. Recovery from False State in Distributed Routing Algorithms. Tech. Rep. UM-CS-2010-017, University of Massachusetts Amherst, 2010.

[35] 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.

[36] 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.

[37] Jefferson, D. Virtual Time. *ACM Trans. Program. Lang. Syst. 7*, 3 (1985), 404–425.

[38] 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.

[39] Kodialam, M., and Lakshman, TV. Dynamic routing of bandwidth guaranteed multicasts with failure backup. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (2002), IEEE, pp. 259–268.

[40] 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.

[41] 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.

[42] Li, G., Wang, D., and Doverspike, R. Efficient distributed mpls p2mp fast reroute. In *Proc. of IEEE INFOCOM* (2006).

[43] Lichtenstein, D. Planar Formulae and Their Uses. *SIAM J. Comput. 11*, 2 (1982), 329–343.

[44] Liu, P., Ammann, P., and Jajodia, S. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases 8*, 1 (2000), 7–40.

[45] 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.

[46] Lu, Q., and Zhang, H. Implementation of approximation algorithms for the multicast congestion problem. In *Experimental and Efficient Algorithms*. Springer, 2005, pp. 152–164.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Nemhauser, G., Wolsey, L., and Fisher, M. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming 14*, 1 (1978), 265–294.

[54] Neumnann, R. Internet routing black hole. *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems 19*, 12 (May 1997).

[55] Padmanabhan, V., and Simon, D. Secure Traceroute to Detect Faulty or Malicious Routing. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 77–82.

[56] Pfaff, B., et al. Openflow switch specification version 1.1.0 implemented (wire protocol 0x02), 2011.

[57] Pointurier, Y. Link failure recovery for mpls networks with multicasting. Master's thesis, University of Virginia, 2002.

[58] Psounis, K. Active networks: Applications, security, safety, and architectures. *Communications Surveys & Tutorials, IEEE 2*, 1 (1999), 2–16.

[59] 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.

[60] 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.

[61] Tam, AS-W, Xi, K., and Chao, J. H. A fast reroute scheme for ip multicast. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE* (2009), IEEE, pp. 1–7.

[62] Tian, A.J., and Shen, N. Fast reroute using alternative shortest paths. draft-tian-frr-alt-shortest-path-01.txt, July 2004.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] Xu, B., and Abur, A. Optimal Placement of Phasor Measurement Units for State Estimation. Tech. Rep. PSERC Publication 05-58, October 2005.

[68] 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&.

[69] 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.