Towards Practical Policies for Network Control

by

Anurag Rai

B.S., Brigham Young University (2010)
M.S., Brigham Young University (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Signature redacted
Author ........
Department of Electrical Engineering and Computer Science
August 22, 2018

Signature redacted
Certified by ........................................
Eytan H. Modiano
Professor of Aeronautics and Astronautics
Thesis Supervisor

Signature redacted
Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students
Abstract

In the past three decades, several network control policies geared towards achieving throughput optimality have been developed. However, rarely any of these policies have been used in practice. We study three different issues that impede this transition and propose solutions that can facilitate the move.

Throughput optimal routing policies are known to have poor delay performance because of packets traversing loops in the network. In the first part of the thesis, we develop a new distributed policy that forwards packets along directed acyclic graphs (DAGs) to avoid the looping problem. This policy uses a link reversal algorithm to improve the DAGs in order to support any achievable traffic demand.

In the second part, we address the problem of optimal routing in overlay networks. Since most devices attached to the existing network do not support throughput optimal routing, a gradual move by forming an overlay network has been proposed in the literature. We develop a new algorithm that the overlay devices can use to achieve the maximum throughput. This algorithm requires the knowledge of the queue lengths of the legacy network, which might not be available. Hence, we also propose a method based on linear regression to estimate these quantities.

To build a high throughput and robust overlay network, it is important to know the topology of the underlying network. However, the network owners usually keep the topology information private. Hence, in the third part of the thesis, we consider the problem of inferring the topology of a network using the measurements available at the end nodes. We use the interference information about the paths to formulate the topology inference problem as an integer program. We develop polynomial time algorithms to solve it optimally for networks with tree and ring topologies. Finally, we use the insights from these algorithms to develop a heuristic for identifying general topologies.
Acknowledgments

I would like to express my gratitude towards my advisor Eytan Modiano for taking me into his group, despite coming from a different background compared to the other members, and then training me and guiding me through the PhD process. Without his continuous help, I definitely would not have completed this degree.

I would also like to thank my committee members Prof. Dimitri Bertsekas and Prof. Mohammad Alizadeh for taking their extremely valuable time to listen to me present on several occasions. Thank you for asking me tough questions and giving me valuable suggestions.

Thanks to Georgios and Chih-ping for helping me with my first project and my first major publication as a PhD student. This really helped me get going with my research.

I also want to thank my undergraduate and master's advisor from BYU, Sean Warnick, for giving me the confidence and the background necessary to apply and get accepted into the PhD program. Also, thanks for keeping in touch even after I left his lab.

Thanks to my friends in CNRG for their moral and technical support. Our technical discussions have helped me build confidence in my abilities, and our lunchtime and dinnertime discussions ranging from pop culture, politics to morality has helped me grow as a person. Especial thanks to Igor for helping me improve my billiards skills.

I would like to thank my roommates at Willoughby St. The people I have met while at this apartment and the relationships I have formed are at least as important to me as my degree. Not only I got to learn about politics, policy and science from the experts, I also had the opportunity to learn a great deal about food, language, and culture from all around the world. I would like to thank everyone for watching television with me. Thanks to Joern and Kavita for hosting me in DC, thanks to Sanita for feeding me so many times, thanks Yenal for educating me on fine dining, thanks Isadora for taking me rock climbing, thanks Sandra for showing me your
beautiful town and for the best paella, thanks Gab for trying to teach me to skate, thanks Alex for encouraging me to travel to Europe, thanks Nil for the enthusiasm to go to the beach, thanks Jelmi for educating me on molecular biology, and thanks to Andy and Gerry for providing such a great place to live.

I would also like to thank my other friends in Boston and across the US. Thanks Maria and Juan for hosting me in Valladolid. Thanks to Annette and Phil for hosting me at Urbana. Thanks Kyu-Seob for hosting me in the Bay Area. Thanks Minu and Subarna, and Biba and Saurav for many, many wonderful dinners. Thanks to Manoj and Sameer for travelling with me and hosting me at your places. I would also like to extend my gratitude to my college roommates from Westwood for teaching the young me how to survive in this foreign land. I would not be here without your help.

I would like to thank my family in Nepal and in the US for their love and support. Thanks to my parents Ashok and Sushila, and brother Ujwal for their constant encouragement. Also thanks to my uncles, aunts, and all my cousins for believing in me. Thanks to all your support, I am finally done!

My research was supported by the grants from National Science Foundation, CNS-1116209 and CNS-1524317, from Office of Naval Research, N00014-12-1-0064, and from DARPA I2O and Raytheon BBN Technologies under Contract No. HROO 1115-C-0097.
Contents

1 Introduction 17
  1.1 Related Work ........................................... 20
    1.1.1 Reducing Delay in Backpressure Routing ............... 20
    1.1.2 Routing in Overlay Networks ......................... 21
    1.1.3 Discovering Network Topology ....................... 22
  1.2 Summary of Our Contributions .......................... 23
    1.2.1 Loop Free Backpressure Algorithm .................... 23
    1.2.2 Optimal Routing Algorithm in Overlay Networks ........ 23
    1.2.3 Topology Discovery Using Path Interference .......... 24

2 Loop-Free Backpressure Routing 27
  2.1 System Model and Definitions .......................... 29
    2.1.1 Network model ..................................... 29
    2.1.2 Flow equations .................................... 30
    2.1.3 Properties of queue overload vector ................ 32
  2.2 Link-Reversal Algorithms ............................... 33
    2.2.1 Initial DAG \( D_0 \) ................................ 34
    2.2.2 Overload detection ................................. 35
    2.2.3 Link reversal ...................................... 36
    2.2.4 Arrivals outside stability region .................... 40
  2.3 Distributed Dynamic Algorithm ........................ 41
    2.3.1 Multi-source single destination networks ............. 45
    2.3.2 Preventing dead ends ................................ 46
2.3.3 Algorithm modification for topology changes .......................... 47

2.4 Complexity analysis ................................................................. 48

2.5 Simulation Results ................................................................. 50
  2.5.1 Fixed topology ................................................................. 51
  2.5.2 Randomly changing topology ............................................. 53

2.6 Comparison with Enhanced Backpressure ................................... 54

2.7 Multicommodity simulation .................................................... 56

2.8 Conclusion ............................................................................. 58

2.9 Appendix .................................................................................. 59
  2.9.1 Lemma 7 ........................................................................... 59
  2.9.2 Proof of Lemma 6 ............................................................... 61
  2.9.3 Proof of Theorem 3 ............................................................. 62
  2.9.4 Lemma 8 ........................................................................... 65

3 Distributed Algorithm for Throughput Optimal Routing in Overlay Networks ................................................................. 69

3.1 Model ....................................................................................... 71
  3.1.1 Overlay ............................................................................... 72
  3.1.2 Underlay ............................................................................. 74

3.2 Background ............................................................................... 75

3.3 Centralized solution .................................................................. 77

3.4 Fluid Formulation and Distributed Solution ............................... 79
  3.4.1 Dual problem ...................................................................... 80
  3.4.2 Distributed solution ......................................................... 82
  3.4.3 Queue-lengths as dual variables ......................................... 83
  3.4.4 Underlay sources and destinations ..................................... 86
  3.4.5 Rate control ....................................................................... 86

3.5 Unknown Underlay Queues ...................................................... 87
  3.5.1 Delay based approaches ................................................... 87
  3.5.2 Priority probe approach ................................................... 89
### 3.6 Simulation Results
- OORP on the counterexample to OBP
- Estimated Tunnel Backlog
- Background traffic
- Rate control

### 3.7 Conclusion

### 3.8 Appendix: Proof of Theorem 5
- Stationary policy $\pi$
- Analysis of TBP

### 4 Backlog Estimation and Control for Overlay Networks
- Network Model
- Backlog Estimation
  - Backlog and delay
  - Backlog and packets in flight
  - PiF based MMSE estimator for multi-class Jackson networks
- Linear Models from Data
  - Linear model using least squares regression
  - Piecewise linear model using the MARS algorithm
  - Data preparation
  - Simulation result
- Control for Overlay Networks
  - The OORP algorithm
  - Simulation result
- Learning-based network tomography
  - Shared bottlenecks
  - Detecting discrimination between flows
- Conclusion

### 5 Topology Discovery Using Path Interference
- Model
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Network Model</td>
<td>131</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The Interference Matrix $\mathcal{F}$</td>
<td>132</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Minimal topology</td>
<td>133</td>
</tr>
<tr>
<td>5.2</td>
<td>Integer Programming Formulation</td>
<td>134</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Integer program</td>
<td>134</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Example</td>
<td>136</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Upper bound</td>
<td>137</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Lower bound</td>
<td>139</td>
</tr>
<tr>
<td>5.2.5</td>
<td>A sufficient condition for optimality</td>
<td>141</td>
</tr>
<tr>
<td>5.3</td>
<td>Proof of Theorem 3</td>
<td>142</td>
</tr>
<tr>
<td>5.4</td>
<td>Identifying Trees</td>
<td>143</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Algorithm</td>
<td>143</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Analysis</td>
<td>146</td>
</tr>
<tr>
<td>5.5</td>
<td>Identifying Rings</td>
<td>149</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Algorithm</td>
<td>149</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Analysis</td>
<td>150</td>
</tr>
<tr>
<td>5.6</td>
<td>Identifying General networks</td>
<td>152</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Algorithm</td>
<td>152</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Simulation result</td>
<td>154</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusion</td>
<td>156</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>159</td>
</tr>
</tbody>
</table>
List of Figures

2-1 Illustration of the Gafni-Bertsekas link reversal when the dashed link in Figure 2-1(a) is lost. At every iteration, the algorithm reverses all the links incident to the nodes with no outgoing link, here represented by the colored nodes. .................................................. 34

2-2 Iterative process to find the DAG $D_{k^*}$ that supports the throughput $\lambda$. 34

2-3 Illustration for the proof of Lemma 3. ................................. 38

2-4 A link $\{a, b\}$ in the network in Fig. 2-3 before and after link reversal. Before the reversal, the flow $f_{ab}$ is zero on $(a, b)$. After the reversal, an $\epsilon$ flow can be sent over $(b, a)$ so that $(\hat{q}_a, \hat{q}_b) <_{\text{lex}} (q_{k,a}^{\min}, q_{k,b}^{\min})$, while the rest of the flow allocation remains the same. ................................. 39

2-5 Different min-cuts in a unit capacity line network. While every min-cut creates a bottleneck for the network, the smallest min-cut is the first bottleneck and has the smallest number of nodes in the source side of the cut. ................................................................. 42

2-6 CDF of the number of iterations taken by the link reversal algorithm to obtain the optimal DAG for Erdős-Rényi networks with $|N|$ nodes. 50

2-7 Figure (a) depicts the original network. Figures (b)-(e) are the various stages of the DAG. The red nodes represent the overloaded nodes, and the dashed line shows the boundary of the overloaded and the non-overloaded nodes. ................................................................. 52

2-8 Average backlog in the network (Fig. 2-7(a)) with fixed topology for the Loop Free Backpressure (LFBP) and the Backpressure (BP) algorithms. 52
Evolution of total backlog in the network over time for $\rho = 0.5$. The backlog grows at the rate of $\lambda = 7.5$ until the first reversal at $t = 150$ because the initial DAG was chosen to have zero throughput.

Initial DAG for the LFBP algorithm chosen so that the LFBP needs several iterations to reach the optimal DAG. All the links have capacity six.

Average backlog in the network with random link failures (Fig. 2-10) for the Loop Free Backpressure algorithm and the Backpressure algorithm.

Ring network topology. All links are bidirectional with unit capacity. Traffic goes from node 1 to node 10.

Average backlog and delay in a ring network where the source and the destination nodes are neighbors.

Average backlog and delay in a multicommodity network with fixed topology for BP, EBP and LFBP algorithms. Both EBP and LFBP performs significantly better than BP. In this topology, EBP performs better than LFBP because most of the throughput can be obtained by using the short paths.

A partition of the node set $N$ where $A_k = C \cup D$ and $B = C \cup E$.

Let $(A_k, A_k^c)$ be the smallest min-cut. We showed that $s \in B_1$. Say, $c_{C_1,A^c} \geq c_{B_1,C_1}$ then the cut $(B_1, B_1^c)$ has the capacity of $c_{B_1,A^c} + c_{B_1,C_1} \leq cap(A_k, A_k^c)$. This contradicts the assumption that $(A_k, A_k^c)$ is the smallest min-cut. So, $c_{C_1,A^c} < c_{B_1,C_1}$.

Let $(A_k, A_k^c)$ be the smallest min-cut. We showed that $d \in C$. Say, $c_{AB} > c_{BC}$ then the cut $(B \cup A_k, (B \cup A_k)^c)$ has the capacity of $c_{BC} + c_{A_kC} < c_{AB} + c_{A_kC} = cap(A_k, A_k^c)$. This contradicts the assumption that $(A_k, A_k^c)$ is the smallest min-cut. So, $c_{AB} < c_{BC}$.
2-18 Here $l_i$ represents the sum of the capacities of the links going from one partition to the next in the DAG $D_k$, and $l'_i$ represents the sum of the link capacities in the DAG $D_{k+1}$. For example, $l9$ and $l9'$ represent the links that go from $(A_k \cup A_{k+1})^c$ to $(A_k \cap A_{k+1})$ in DAGs $D_k$ and $D_{k+1}$ respectively.

3-1 Overlay network architecture. If the overlay node A has traffic for node D, it can either route it directly using the tunnel from A to D or relay it through other overlay node B or C.

3-2 A physical network and its corresponding overlay network.

3-3 Counterexample for throughput optimality of the Overlay Backpressure Policy of [37].

3-4 Link (3, 4) never builds a queue as the flow gets bottlenecked by (2, 3).

3-5 The actual queue-length of a single FIFO queue and its estimate calculated using delay. The arrivals happen at the rate of 2 packets per time-slot for the first $\tau$ time-slots, and there are no arrivals after that. Service rate is fixed at 1 packet per time-slot.

3-6 Performance of different routing algorithms on the overlay network shown in Figure 3-3(a).

3-7 Physical and overlay network topology for simulations in Sections 3.6 B and 3.6 C.

3-8 Performance of OORP under different measures of tunnel backlog.

3-9 Performance of OORP in a network with background traffic.

3-10 Topology for the rate control experiment. The dotted lines show the tunnel assigned to the third commodity and the background traffic.

3-11 Throughput achieved by the rate control algorithm with OORP converges to the rate that maximizes utility.

4-1 Feedback control system
An overlay network with three overlay (white) nodes and three underlay (gray) nodes. The underlay uses fixed, single path routing algorithm and FIFO service discipline. The overlay nodes can send packets directly to their destination or forward them through other overlay nodes by using a dynamic routing algorithm.

Backlog vs. delay

A 3-link underlay network. The underlay nodes are shown as gray nodes. The dotted lines show the two tunnels in the network.

MMSE estimator for the underlay network shown in Figure 4-4 with Jackson network assumptions.

Cartoon of a linear and a piece-wise linear model fit to data.

The piecewise linear model of the backlog as a function of PiF for the underlay network shown in Figure 4-4.

Comparison of the linear model obtained by least squares regression and the piecewise linear model obtained by MARS. The bar chart shows the squared coefficient of variation for the mean squared error (MSE). The numbers on top of the bars show the improvement in the MSE obtained by the piecewise linear model over the linear model.

Topology for the OORP simulation. The network consists of 4 overlay (white) nodes, 10 underlay (gray) nodes, and 18 tunnels. The underlay network uses shortest path routing. The dotted lines show some of the tunnels available to the overlay node 1.

Performance comparison of OORP under various tunnel-backlog estimation schemes. The methods developed in this chapter (solid lines) perform very well without any support from the underlay network.

Different ways three tunnels can share bottlenecks.

Two topologies in Figures 5-1(a) and 5-1(b) produce the same \( \mathcal{F} \) matrix. The white nodes are overlay and gray nodes are underlay, and the network uses the shortest path routing.
5-2 Recovering a network topology by solving the ILP. . . . . . . . . . . . 137
5-3 Example of an execution of Steps 1 to 3 of Algorithm 1. . . . . . . . . 139
5-4 First two iterations of the tree identification algorithm. The third iteration (not shown) recovers the complete graph in Figure 5-4(a). . 145
5-5 If tunnel \( l \) interferes with the most number of tunnels then all the nodes connected to \( l_2 \) must be leaf nodes. Otherwise, there exists a tunnel \( (n_1, n, ..., l_3) \) that interferes with more tunnels than tunnel \( l \). . . . . . . . . . . . 146
5-6 Example of Procedure 15 at work. Node \( p(1) \) has three neighbors \( p(2), p(3) \) and \( p(4) \). Procedure 15 first attempts to identify two nodes, e.g. 2 and 4, by minimizing the number of tunnel intersections. Then node 3 is identified by using the property that tunnel \( (3, ..., 1) \) doesn’t interfere with the tunnels \( (2, ..., 4) \) or \( (4, ..., 2) \). . . . . . . . . . . . . . 153
5-7 Comparison of Algorithm 16 and RGD1. . . . . . . . . . . . . . . . 157
Chapter 1

Introduction

The backpressure routing algorithm [66] for achieving maximum throughput in a multicommodity network was developed by Tassiulas and Ephremides in 1992. This algorithm stabilizes all the queues in a multi-commodity, multi-hop wireless network for any traffic that is stabilizable, a property often referred to as throughput optimality. This algorithm is distributed, i.e. only the neighbor nodes need to share state information, and it is robust to topology and link capacity changes. Essentially, this algorithm provides a simple, distributed and robust solution to a multi-commodity max-flow problem. Backpressure routing possesses many of the characteristics that one would strive for while designing a communication network.

Backpressure routing has received a significant amount of attention in the literature and has generated thousands of follow up publications. It has been modified to be used in different networking scenarios such as switches [43], wireless networks with power control and time-varying channels [53], optical networks [14], etc. In fact, it has led to the creation of a new area in networking called throughput optimal network control [27].

In spite of the significant amount of research, these throughput optimal algorithms have had little to no success in terms of actual deployments. Some of this can be attributed to the problems inherent to Backpressure: e.g. high delay [62], out of order packet arrivals at the destination [40], large buffer requirement [44], etc. Another reason is that the deployment of such algorithms require new hardware
with new capabilities which means a complete overhaul of the legacy network. We study three different issues that impede the transition and propose solutions that can facilitate the move.

It is well known that the delay performance of backpressure is poor [15]. The high delay is attributed to a property of backpressure that allows the packets to loop within the network instead of moving towards the destination. To observe this problem, imagine a network with a single packet in it. Backpressure sends packets from larger queues to smaller queues, and in this network, the queue containing the single packet is the largest. Hence, the packet can be transmitted to any of the neighboring nodes, including a neighbor that the packet has already traversed, creating a loop. In the first part of this thesis, Chapter 2, we address this problem by constraining the routing to a directed acyclic graph (DAG), and then improving the DAG by using a new link reversal algorithm similar to the ones purposed in [25]. Initially, we generate an arbitrary DAG and use backpressure routing over it. If the initial DAG has a max-flow smaller than the traffic demand, parts of the network become overloaded. By reversing the direction of the links that point from non-overloaded to overloaded nodes a new DAG with a lower overload is obtained. Iterating over this process, our distributed algorithm gradually converges to a DAG that supports any traffic demand feasible in the network. We also design a scheme to maintain loop free paths even when the network topology is changing over time. Hence, we reduce the delay of backpressure while maintaining the robustness and throughput optimality properties.

The second problem, considered in Chapters 3 and 4 of this thesis, involves optimal routing in overlay networks. In order to implement the backpressure routing algorithm, each router in the network needs to be able to perform several tasks other than forwarding, such as maintaining queues per commodity and computing the difference between the size of the queues on the two ends of each link. These functionalities are not provided by the majority of the currently available routers, and a complete overhaul of the network is impractical. So, in Chapter 3, we propose an incremental approach where we build an overlay network on top of the existing
legacy network. The overlay nodes are capable of implementing any dynamic routing policy, and the legacy underlay has a fixed, single path routing scheme and uses a simple work-conserving forwarding policy. The overlay network can improve the achievable throughput and robustness to link failures in the legacy network by using multiple routes, which consist of direct routes and indirect routes through other overlay nodes. We show that the existing heuristic for routing in such overlay networks is suboptimal, and then develop an optimal dynamic routing algorithm called the Optimal Overlay Routing Policy (OORP). OORP is derived using the classical dual subgradient descent method, and it can be implemented in a distributed manner.

Although the OORP algorithm can be entirely implemented at the overlay nodes, it requires the underlay queue length information in order to operate. This measurement might not be readily available to the overlay nodes. Hence, in Chapter 4, we develop a new method that learns the underlay backlog information from historical data. Our method, based on linear regression, does not require any knowledge of the network topology and uses only the measurements that are available at the end nodes, namely delay experienced by the packets and the number of packets in flight. We show that the backlog estimation scheme in combination with the control algorithm of Chapter 3 can achieve maximum throughput in an overlay network setting.

To build a high throughput and robust overlay network, it is important to know the topology of the underlying network [3, 38]. However, the network owners usually keep the topology information private. Hence, in the third part of the thesis, Chapter 5, we consider the problem of inferring the topology of a network using the measurements available at the end nodes without any cooperation from the internal nodes. To this end, building upon the linear regression model of the backlog from Chapter 4, we provide a simple method to obtain path interference which identifies whether two paths in the network intersect with each other. Using this information, we formulate the topology inference problem as an integer program, then we develop polynomial time algorithms to solve it optimally for networks with tree and ring topologies. Finally, we use the insights developed from these algorithms to design a heuristic for identifying general topologies. Simulation results show that our heuristic outperforms
a recently proposed algorithm that uses distance measurements for topology discovery.

1.1 Related Work

1.1.1 Reducing Delay in Backpressure Routing

To mitigate the issue of high delay in Backpressure prior works propose delay-aware backpressure techniques. Backpressure enhanced with hop count bias is first proposed in [53]. An alternative backpressure modification that utilizes shortest path information was shown to be throughput optimal in [62]. These techniques give preference to the shorter paths when the load is low. As the load increases, they use longer and longer paths until all the possible paths are used.

A different line of work proposes to learn the network topology using backpressure and then use this information to enhance routing decisions. In [70] backpressure is constrained to a subgraph which is discovered by running unconstrained backpressure for a time period and computing the average number of packets routed over each link. Learning is effectively used in scheduling [32] and utility optimization [33] for wireless networks.

In our work, we limit the routing to a DAG, and if necessary, improve the DAG by using a new link reversal algorithm. The link-reversal algorithms were introduced in [25] as a means to maintain connectivity in networks with volatile links. These distributed algorithms react to any topological changes to obtain a DAG such that each node has a loop-free path to the destination. In [55], one of the link-reversal algorithms was used to design a routing protocol (called TORA) for multihop wireless networks. Although these algorithms provide loop free paths and guarantee connectivity from the nodes to the destination, they do not maximize throughput. Thus, the main goal of our work is to create a new link-reversal algorithm and combine it with the backpressure algorithm to construct a distributed throughput optimal algorithm with improved delay performance.
1.1.2 Routing in Overlay Networks

There has been a significant amount of research on overlay networks. Overlay networks have been used to improve the performance and capabilities of computer networks for a long time. The Internet itself started as a data network built on top of the telephone network. An overlay architecture to improve the robustness of the Internet was proposed in [3], where alternate overlay paths are used to overcome path loss in the underlay network. Placement for the overlay node to improve path diversity was studied in [31]. Architectures for designing overlay networks that improve different quality of service metrics have been proposed in [45], [63]. Currently overlay is being used extensively for applications such as content delivery, multicast, etc.

However, throughput optimal routing in overlay network is quiet recent. This problem was first studied in [37]. The main goal of that paper was to place the minimum number of overlay nodes into an existing underlay in order to maximize network throughput. In particular, the authors show that with just a few overlay nodes, maximum network throughput can be achieved. In addition,[37] also shows that the backpressure routing algorithm, which is known to be optimal in a wide range of scenarios, leads to a loss in throughput when used in an overlay network. The authors of [37] propose a heuristic for optimal routing called the Overlay Backpressure Policy (OBP). An optimal backpressure like routing algorithm for a special case, where the underlay paths do not overlap with each other, was given in [57], which also proposes a threshold based heuristic for general overlay networks. The schemes presented in [37] and [57] are very similar and were conjectured to be throughput optimal.

Our algorithm, OORP, achieves throughput optimality in the overlay network, however, it needs the queue backlog information from the underlay. Since this information might not be available, we need to estimate it using measurements available at the overlay network. Similar problem is faced by all control algorithms for the Internet. To overcome this, TCP algorithms such as Tahoe [36] infer the congestion from packet losses, i.e. a successful packet implies low congestion and a dropped
packet implies high congestion. Other versions such as TCP-Vegas [13] use delay experienced by the packets as a measure of congestion along the path, and yet others use explicit congestion notification [1]. In an overlay network setting, the authors of [37] use the number of packets that have been sent along a route but have not made it to the destination, the packets in flight, as an estimate of the congestion along the route. This method makes a simple inference: the higher the number of packets in flight, the higher the congestion along the path. In our work, we show that either using delay or the packets in flight is not sufficient for stabilizing the network and propose a new learning based approach.

1.1.3 Discovering Network Topology

Prior work on topology discovery can be divided into two main categories: algorithms that require cooperation from the internal nodes and the algorithms that do not. Many algorithms for topology discovery, usually designed for the purpose of mapping the Internet, use ICMP commands like traceroute [64, 21, 29]. These methods require some level of cooperation from the network providers. The other methods, that fall under the category of network tomography [69, 17], use data that can be measured directly at the end nodes. Our method falls under this category as we do not seek any information from the internal nodes.

In the network tomography literature significant attention has been given to the discovery of tree networks. Papers such as [18, 22, 54] use probing mechanisms to infer single source multiple destination trees. There is also some work on combining these single source trees to form a multi-source multi-destination networks [19, 61]. In [41], the authors provide a method for identifying minimal trees with multiple sources and multiple destinations by using distance measurements.

In [2], the authors develop an algorithm called RGD1 that attempts to discover a general network topology. It uses a set for four nodes that share a link, called quatrets and uses them to build an approximation of the entrie network. The discovery of the quatret and placement of the nodes in the topology requires the shortest path distance between the nodes, which is inferred using packet delay.
1.2 Summary of Our Contributions

1.2.1 Loop Free Backpressure Algorithm

For a single commodity network, we propose the loop free backpressure (LFBP) algorithm, a distributed routing scheme that eliminates loops and retains the throughput optimality property. This is achieved by exploiting the properties of backpressure regarding the queue growth rates in an overloaded network. To prove the correctness of this algorithm, we study the lexicographic optimization of the queue overload. For the situation that there is a queue overload, i.e. the DAG does not support the traffic arrival rate, our novel link-reversal algorithm reverses link direction based on the overload conditions to form a new DAG. Specifically, we reverse links going from a non-overloaded node to an overloaded node. We show that this will result in a DAG with lexicographically smaller queue growth rates. We show that the number of reversals required to obtain the optimal DAG is pseudo-polynomial in the number of nodes in the network. Furthermore, we provide a mechanism so that this algorithm can be used in a network with changing topology.

The simulation results of LFBP show a significant delay improvement over backpressure in static and dynamic networks. We extend the LFBP algorithm to networks with multiple commodities, and provide a simulation result to show its delay improvement over backpressure. We also compare LFBP against the Enhanced Backpressure algorithm of [53] to show that these algorithms can outperform each other depending on the network topology and traffic load.

1.2.2 Optimal Routing Algorithm in Overlay Networks

We begin by providing a counterexample to show that OBP, the heuristic from [37], is in fact not throughput optimal. To derive the optimal policy, we notice that the suboptimality of backpressure arises from its failure to accurately account for congestion in the underlay paths. Traditional backpressure doesn’t keep track of the packets in the underlay which can lead the overlay nodes to send too many packets
into the underlay creating instability of underlay queues. We will first develop a centralized solution which achieves optimality by limiting the traffic injected into the underlay so that the underlay queues are always bounded. Then we use the intuition gained from this policy to develop a distributed solution which uses the queue backlog information in the underlay to compute the amount of flow transmitted into each underlay path. This policy implicitly favors underlay paths that are less congested and preserves stability of all the queues.

We also develop a new method to estimate the backlog along each path in the underlay network, which is necessary to execute the OORP algorithm. We build linear and piece-wise linear models of the backlog as a function of the number of packets in flight (PiF) by using regression based techniques. The model is trained with the historical PiF and delay data. We then use this model to produce estimates of the backlog and show that when used in conjunction with OORP, its performance is very close to that of a system with perfect state information.

1.2.3 Topology Discovery Using Path Interference

In order to discover a network topology, we use the path interference information. We provide a new way to check whether two given paths in a network interfere. We use the interference pattern of the paths to formulate an integer linear program (ILP) that obtains the network that has the fewest number of links and supports the given interferences. The solution provides a new method to discover a general network topology. We derive an upper and a lower bound of the ILP, and use the lower bound to obtain a condition under which a network is an optimal solution to the ILP.

We design two polynomial time algorithms to recover tree and ring networks and show that if the network is in fact a tree or a ring, the algorithms solve the ILP optimally. Conversely, we can also use these algorithms to eliminate trees and ring networks from the possible networks generating a given interference pattern. Inspired by the tree and the ring algorithms, we develop a polynomial time heuristic to identify general networks. This scheme attempts to separate the trees from the rings in the given network using the interference pattern, then use the tree algorithm on the trees
and an enhanced version of the ring algorithm on the rest of the graph. We combine the inferred graphs to form the final result. Using simulations we show that this method outperforms the RGD1 algorithm of [2].
Chapter 2

Loop-Free Backpressure Routing

Throughput and delay are two common metrics used to evaluate the performance of communication networks. For networks that exhibit high variability, such as mobile ad hoc networks, the dynamic backpressure routing policy [66] is a highly desirable solution, known to maximize throughput in a wide range of settings. However, the delay performance of backpressure is poor [15]. The high delay is attributed to a property of backpressure that allows the packets to loop within the network instead of moving towards the destination. In this chapter we improve the delay performance of backpressure by constraining the routing to loop free paths.

To eliminate loops in the network, we assign directions to the links such that the network becomes a directed acyclic graph (DAG). Initially, we generate an arbitrary DAG and use backpressure routing over it. If the initial DAG has max-flow smaller than the traffic demand, parts of the network become overloaded. By reversing the direction of the links that point from non-overloaded to overloaded nodes a new DAG with a lower overload is obtained. Iterating over this process, our distributed algorithm gradually converges to a DAG that supports any traffic demand feasible in the network. Hence the loop-free property is achieved without the loss of throughput.

Prior work identifies looping as a main cause for high delays in backpressure routing and proposes delay-aware backpressure techniques. Backpressure enhanced with hop count bias is first proposed in [53] to drive packets through paths with smallest hop counts when the load is low. An alternative backpressure modification that uti-
lizes shortest path information is proposed in [62]. A different line of works proposes to learn the network topology using backpressure and then use this information to enhance routing decisions. In [70] backpressure is constrained to a subgraph which is discovered by running unconstrained backpressure for a time period and computing the average number of packets routed over each link. Learning is effectively used in scheduling [32] and utility optimization [33] for wireless networks. In our work we aim to eliminate loops by restricting backpressure to a DAG, then dynamically improving the DAG by reversing the links in a distributed manner.

The link-reversal algorithms were introduced in [25] as a means to maintain connectivity in networks with volatile links. These distributed algorithms react to any topological changes to obtain a DAG such that each node has a loop-free path to the destination. In [55], one of the link-reversal algorithms was used to design a routing protocol (called TORA) for multihop wireless networks. Although these algorithms provide loop free paths and guarantee connectivity from the nodes to the destination, they do not maximize throughput. Thus, the main goal of this chapter is to create a new link-reversal algorithm and combine it with the backpressure algorithm to construct a distributed throughput optimal algorithm with improved delay performance. The main contributions of this chapter are as follows:

- For a single commodity network, we study the lexicographic optimization of the queue growth rate. We develop a novel link-reversal algorithm that reverses link direction based on the overload conditions to form a new DAG with lexicographically smaller queue growth rates.

- We propose the loop free backpressure (LFBP) algorithm, a distributed routing scheme that eliminates loops and retains the throughput optimality property. This is achieved by exploiting the properties of backpressure regarding the queue growth rates in an overloaded network.

- Our simulation results of LFBP show a significant delay improvement over backpressure in static and dynamic networks.

- We extend the LFBP algorithm to networks with multiple commodities, and
provide a simulation result to show its delay improvement over backpressure.

2.1 System Model and Definitions

2.1.1 Network model

We consider the problem of routing single-commodity data packets in a network. The network is represented by a graph $G = (N, E)$, where $N$ is the set of nodes and $E$ is the set of undirected links $\{i, j\}$ with capacity $c_{ij}$. Packets arrive at the source node $s$ at rate $\lambda$ and are destined for a receiver node $d$. Let $f_{\text{max}}$ denote the maximum flow from node $s$ to $d$ in the network $G$. The quantity $f_{\text{max}}$ is the maximally achievable throughput at the destination node $d$. In this chapter, we do not solve the link scheduling problem, i.e. we assume that all links can be scheduled at the same time.

To avoid unnecessary routing loops, we restrict forwarding along a directed acyclic graph (DAG) embedded in the graph $G$. An optimal DAG exists to support the max-flow $f_{\text{max}}$ and can be found by: (i) computing a feasible flow allocation $(f_{ij})$ that yields the max-flow $f_{\text{max}}$ in $G$ (e.g. using [24]); (ii) trimming any positive flow on directed cycles; (iii) defining an embedded DAG by assigning a direction for each link $\{i, j\}$ according to the direction of the flow $f_{ij}$ on that link. Since backpressure achieves the max-flow of a constrained graph [27], performing backpressure routing over the optimal DAG supports $\lambda$.

This centralized approach is unsuitable for communication networks, especially when the link capacities are time-varying or when the network undergoes frequent topology changes. In such situations, the optimal embedded DAG also changes with time, which requires constantly repeating the above offline process. Instead, it is possible to use a distributed adaptive mechanism that reverses the direction of links until a DAG that supports the current traffic demand is found. In this chapter we propose an algorithm that reacts to the traffic conditions and changes in network topology by switching the direction of some links. To understand the properties of the link-reversing operations, we first study the fluid level behavior of a network under
overload conditions.

2.1.2 Flow equations

Consider an *embedded* DAG $D_k = (N_k, E_k)$ in the network graph $G$, where $N_k = N$ is the set of network nodes and $E_k$ is the set of directed links. For each undirected link $\{i, j\} \in E$, either $(i, j)$ or $(j, i)$ belongs to $E_k$ (but not both). Each directed link $(i, j)$ has the capacity of the undirected counterpart $\{i, j\}$, which is $c_{ij}$. Let $f_{k}^{\text{max}}$ be the maximum flow of the DAG $D_k$ from the source node $s$ to the destination node $d$. Any embedded DAG has smaller or equal max-flow with respect to $G$, $f_{k}^{\text{max}} \leq f^{\text{max}}$.

For two disjoint subsets $A$ and $B$ of nodes in $D_k$, we define $\text{cap}_k(A, B)$ as the total capacity of the directed links going from $A$ to $B$, i.e.,

$$\text{cap}_k(A, B) = \sum_{(i, j) \in E_k; i \in A, j \in B} c_{ij}. \quad (2.1)$$

A cut is a partition of nodes $(A, A^c)$ such that $s \in A$ and $d \in A^c$. A cut $(A_k, A_k^c)$ is a min-cut if it minimizes the expression $\text{cap}_k(A_k, A_k^c)$ over all cuts. By the max-flow min-cut theorem $f_{k}^{\text{max}} = \text{cap}_k(A_k, A_k^c)$, where $(A_k, A_k^c)$ is a min-cut of the DAG $D_k$. We remark that a cut in $D_k$ is also a cut in $G$ or another embedded DAG. However, the value of $\text{cap}_k(A, A^c)$ for partition $(A, A^c)$ depends on the graph considered (see summation in (2.1)), and thus the min-cuts may differ substantially per DAG.

We consider the network as a time-slotted system, where slot $t$ refers to the time interval $[t, t + 1)$, $t \in \{0, 1, 2, \ldots\}$. Each network node $n$ maintains a queue $Q_n(t)$, where $Q_n(t)$ also denotes the queue backlog at time $t$. We have $Q_d(t) = 0$ for all $t$ since any packet reaching the destination is removed from the network immediately. Let $A(t)$ be the number of exogenous packets arriving at the source node $s$ in slot $t$.

Under a routing policy that forwards packets over the directed links defined by the DAG $D_k$, let $F_{ij}(t)$ be the number of packets that are transmitted over the directed link $(i, j) \in E_k$ in slot $t$; the link capacity constraint states that $F_{ij}(t) \leq c_{ij}$ for all $t$.

---

1The notation $D_k$ of an embedded DAG is useful in the chapter; it will denote the DAG that is formed after the $k$th iteration of the link-reversal algorithm.
The queues $Q_n(t), n \neq d$, are updated over slots according to

$$Q_n(t) = Q_n(t - 1) + 1_{[n=s]}A(t) + \sum_{i:(i,n) \in E_k} F_{in}(t) - \sum_{j:(n,j) \in E_k} F_{nj}(t), \quad (2.2)$$

where $1_{[\cdot]}$ is an indicator function.

To study the overload behavior of the system we define the queue overload (i.e., growth) rate at node $n$ as

$$q_n = \lim_{t \to \infty} \frac{Q_n(t)}{t}. \quad (2.3)$$

Additionally, define the exogenous packet arrival rate $\lambda$ and the flow $f_{ij}$ over a directed link $(i, j)$ as

$$\lambda = \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} A(\tau), \quad f_{ij} = \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} F_{ij}(\tau),$$

where the above limits are assumed to exist almost surely (see [28] for details). Using the recursion (2.2), taking time averages and letting $t \to \infty$, we have the fluid-level equation:

$$q_n = 1_{[n=s]} \lambda + \sum_{i:(i,n) \in E_k} f_{in} - \sum_{j:(n,j) \in E_k} f_{nj}, \quad \forall n \in N \setminus \{d\} \quad (2.4)$$

$$0 \leq f_{ij} \leq c_{ij}, \quad \forall (i, j) \in E_k. \quad (2.5)$$

Equations (2.4) and (2.5) are the flow conservation and link capacity constraints, respectively. A network node $n$ is said to be overloaded if its queue growth rate $q_n$ is positive, which implies that $Q_n(t) \to \infty$ as $t \to \infty$ (see (2.3) and [51]). Summing (2.4) over $n \in N$ yields

$$\sum_{i:(i,d) \in E_k} f_{id} = \lambda - \sum_{n \in N} q_n, \quad (2.6)$$

where $\sum_{i:(i,d) \in E_k} f_{id}$ denotes the throughput received at the destination $d$. Therefore, equation (2.6) states that the received throughput is equal to the exogenous arrival rate $\lambda$ less the sum of queue growth rates $\sum_{n \in N} q_n$ in the network.
2.1.3 Properties of queue overload vector

If the traffic arrival rate $\lambda$ is strictly larger than the maximum flow $f_{k}^{\text{max}}$ of the DAG $D_k$, then from (2.6),

$$\sum_{n \in N} q_n = \lambda - \sum_{i; (i, d) \in E_{k}} f_{id} \geq \lambda - f_{k}^{\text{max}} > 0,$$

which implies that $q_n > 0$ for some node $n \in N$. Let $q = (q_n)_{n \in N}$ be the queue overload vector. A queue overload vector $q$ is feasible in the DAG $D_k$ if there exist overload rates $(q_n)_{n \in N}$ and flow variables $(f_{ij})_{(i, j) \in E_{k}}$ that satisfy (2.4) and (2.5). Let $Q_k$ be the set of all feasible queue overload vectors in $D_k$. We are interested in the lexicographically smallest queue overload vector in set $Q_k$. Formally, given a vector $u = (u_1, \ldots, u_N)$, let $\bar{u}_i$ be the $i$th maximal component of $u$. We say that a vector $u$ is lexicographically smaller than a vector $v$, denoted by $u <_{\text{lex}} v$, if $\bar{u}_1 < \bar{v}_1$ or $\bar{u}_i = \bar{v}_i$ for all $i = 1, \ldots, (j-1)$ and $\bar{u}_j < \bar{v}_j$ for some $j = 2, \ldots, N$. If $\bar{u}_i = \bar{v}_i$ for all $i$, then the two vectors are lexicographically equal, represented by $u =_{\text{lex}} v$.\(^2\) The above-defined vector comparison induces a total order on the set $Q_k$, and hence the existence of a lexicographically smallest vector is always guaranteed [26].

Lemma 1 ([28]). Let $q_k^{\text{min}}$ be the lexicographically smallest vector in the queue overload region $Q_k$ of the DAG $D_k$. We have the following properties:

1. The vector $q_k^{\text{min}}$ is unique in the set $Q_k$.

2. The vector $q_k^{\text{min}}$ minimizes the sum of queue overload rates, i.e., it is a solution to the optimization problem:

$$\begin{align*}
\text{minimize} & \quad \sum_{n \in N} q_n, \quad \text{subject to } q \in Q_k
\end{align*}$$

(direct consequence of Theorem 1 in [28]). Due to (2.6), the corresponding throughput is maximized.

\(^2\)Lexicographic order is also known as dictionary order. Two vectors $u = (3, 2, 1, 2, 1)$ and $v = (1, 2, 3, 2, 2)$ satisfy $u <_{\text{lex}} v$ because $\bar{u}_1 = \bar{v}_1 = 3$, $\bar{u}_2 = \bar{v}_2 = \bar{u}_3 = \bar{v}_3 = 2$, and $\bar{u}_4 = 1 < \bar{v}_4 = 2.$
3. A feasible flow allocation vector \((f_{ij})_{(i,j)\in E_k}\) induces \(q_k^{\text{min}}\) if and only if over each link \((i,j)\in E_k\) the following holds:

\[
\begin{align*}
\text{if } q_i < q_j, & \quad \text{then } f_{ij} = 0, & (2.8) \\
\text{if } q_i > q_j, & \quad \text{then } f_{ij} = c_{ij}. & (2.9)
\end{align*}
\]

In general, there are many flow allocations that yield the maximum throughput. Focusing on those that additionally induce \(q_k^{\text{min}}\) has two advantages. First, these allocations lead to link-reversal operations that improve the max-flow of the DAG \(D_k\). Second, the backpressure algorithm can be used to perform the same reversals and improve the max-flow. We will use these observations to combine link-reversal algorithms with backpressure routing.

### 2.2 Link-Reversal Algorithms

The link-reversal algorithms given in [25], henceforth called the Gafni-Bertsekas link reversal, were designed to maintain a path from each node in the network to the destination. One algorithm relevant to this chapter is the full reversal method. This algorithm is triggered when some nodes \(n \neq d\) lose all of their outgoing links. At every iteration of the algorithm, nodes \(n\), that have no outgoing link, reverse the direction of all their incoming links. This process is repeated until all the nodes other than the destination have at least one outgoing link. When the process stops these nodes are guaranteed to have a path to the destination. The example in Figure 2-1, taken from [25], illustrates this algorithm at work.

Although the full reversal algorithm guarantees connectivity, the resulting throughput may be significantly lower than the maximum possible. Hence, in this chapter we shift the focus from connectivity to maximum throughput. Specifically, we propose a novel link-reversal algorithm that produces a DAG which supports the traffic demand \(\lambda\), assuming \(\lambda \leq f^\text{max}\).

We propose an iterative algorithm which begins with an embedded DAG and at
Figure 2-1: Illustration of the Gafni-Bertsekas link reversal when the dashed link in Figure 2-1(a) is lost. At every iteration, the algorithm reverses all the links incident to the nodes with no outgoing link, here represented by the colored nodes.

Figure 2-2: Iterative process to find the DAG $D_{k^*}$ that supports the throughput $\lambda$. Each iteration produces a new embedded DAG that supports an improved lexicographically smallest overload vector. Within each iteration, an implied routing policy operates on the current DAG yielding the lexicographically optimal flow allocation. A sketch of this process is shown in Figure 2-2.

### 2.2.1 Initial DAG $D_0$

We assume that each node in the network has a unique ID. We use these IDs as a topological ordering to the nodes. So the initial DAG can be created simply by directing each link to go from the node with the lower ID to the node with the higher ID. If the unique IDs are not available, the initial DAG can be created by using a strategy such as the one given in [55].
2.2.2 Overload detection

Given a DAG $D_k$, $k = 0, 1, 2, \ldots$, we suppose that there is a routing policy $\pi$ that yields the lexicographically minimal queue overload vector $q_{k}^{\text{min}}$. Then we use the vector $q_{k}^{\text{min}}$ to detect node overload and decide whether a link should be reversed.

If the data arrival rate $\lambda$ is less than or equal to the maximum flow $f_k^{\text{max}}$ of the DAG $D_k$, then there exists a flow allocation $(f_{ij})$ that supports the traffic demand and yields zero queue overload rates $q_n = 0$ at all nodes $n \in N$. By the second property of Lemma 1 and nonnegativity of the overload vector, the queue overload vector $q_{k}^{\text{min}}$ is zero. Thus, the throughput under policy $\pi$ is $\lambda$ according to (2.6), and the current DAG $D_k$ supports $\lambda$.

On the other hand, if the arrival rate $\lambda$ is strictly larger than the maximum flow $f_k^{\text{max}}$, by the second property in Lemma 1 the maximum throughput is $f_k^{\text{max}}$ and the queue overload vector $q_{k}^{\text{min}} = (q_{k,n}^{\text{min}})_{n \in N}$ is nonzero because we have from (2.7) that

$$\sum_{n \in N} q_{k,n}^{\text{min}} > \lambda - f_k^{\text{max}} > 0.$$ 

We may therefore detect the event “DAG $D_k$ supports $\lambda$” by testing whether the overload vector $q_{k}^{\text{min}}$ is zero or non-zero.

The next lemma shows that if DAG $D_k$ does not support $\lambda$ then it contains at least one under-utilized link (our link-reversal algorithm will reverse the direction of such links to improve network throughput).

**Lemma 2.** Suppose that the traffic demand $\lambda$ satisfies

$$f_k^{\text{max}} < \lambda \leq f_k^{\text{max}}.$$

where $f_k^{\text{max}}$ is the max-flow of the DAG $D_k$ and $f_k^{\text{max}}$ is the max-flow of the undirected network $G$. Then there exists a link $(i, j) \in E_k$ such that $q_{k,i}^{\text{min}} = 0$ and $q_{k,j}^{\text{min}} > 0$.

Such a policy $\pi$ can simply solve an optimization problem offline to compute the required flow allocation. In Section 2.3, we develop a distributed algorithm using backpressure that does not require the computation of the lexicographically optimal overload vector. We use this vector only to prove the properties of our link-reversal algorithm.
Proof of Lemma 2. Let $A_k$ be the set of overloaded nodes under a flow allocation that induces the lexicographically minimal overload vector $q_k^{\text{min}}$ in the DAG $D_k$; the set $A_k$ is nonempty due to $\lambda > f_k^{\text{max}}$ and (2.7). It follows that the partition $(A_k, A_k^c)$ is a min-cut of $D_k$ (see Lemma 7 in the Appendix). By the max-flow min-cut theorem, the capacity of the min-cut $(A_k, A_k^c)$ in $D_k$ satisfies $\operatorname{cap}_k(A_k, A_k^c) = f_k^{\text{max}} < f^{\text{max}}$.

The proof is by contradiction. Let us assume that there is no directed link that goes from the set $A_k^c$ to $A_k$ in the DAG $D_k$. It follows that $\operatorname{cap}_k(A_k, A_k^c)$ is the sum of capacities of all undirected links between the sets $A_k$ and $A_k^c$, i.e.,

$$\operatorname{cap}_k(A_k, A_k^c) = \sum_{i \in A_k, j \notin A_k} c_{ij},$$

which is equal to the value of the cut $(A_k, A_k^c)$ in graph $G$. Since the value of any cut is larger or equal to the min-cut, applying the max-flow min-cut theorem on $G$ we have

$$f^{\text{max}} \leq \sum_{i \in A_k, j \notin A_k} c_{ij} = \operatorname{cap}_k(A_k, A_k^c) = f_k^{\text{max}},$$

which contradicts the assumption that $f_k^{\text{max}} < \lambda \leq f^{\text{max}}$. \hfill \qed

2.2.3 Link reversal

We consider the link-reversal algorithm (Algorithm 1) that reverses all the links that go from an overloaded node to a non-overloaded node. Lemma 2 shows that such links always exist if the DAG $D_k$ has insufficient capacity to support the traffic demand $\lambda \leq f^{\text{max}}$ under the lexicographically minimum overflow vector $q_k^{\text{min}}$. This reversal yields a new directed graph $D_{k+1} = (N, E_{k+1})$.

The rest of the section focuses on proving that this algorithm obtains a DAG that supports the traffic demand $\lambda$ within a finite number of iterations. We begin by showing that every intermediate graph produced by this algorithm is a DAG.

Lemma 3. The directed graph $D_{k+1}$ is acyclic.

---

4The set $A_k^c$ contains the destination node $d$ and is nonempty.
Algorithm 1 Link-Reversal Algorithm

1: for all $(i, j) \in E_k$ do
2:     if $q_{k,i}^{\text{min}} = 0$ and $q_{k,j}^{\text{min}} > 0$ then
3:         $(j, i) \in E_{k+1}$
4:     else
5:         $(i, j) \in E_{k+1}$
6:     end if
7: $k \leftarrow k + 1$
8: end for

Proof of Lemma 3. Recall that $A_k$ is the set of overloaded nodes in the DAG $D_k$ under the lexicographically minimum queue overload vector $q_k^{\text{min}}$. Let $L_k \subseteq E$ be the set of undirected links between $A_k$ and $A_k^c$. Algorithm 1 changes the link direction in a subset of $L_k$. More precisely, it enforces the direction of all links in $L_k$ to go from $A_k$ to $A_k^c$.

We complete the proof by construction in two steps. First, we remove all links in $L_k$ from the DAG $D_k$, resulting in two disconnected subgraphs that are DAGs themselves. Second, consider that we add a link in $L_k$ back to the network with the direction going from $A_k$ to $A_k^c$. This link addition does not create a cycle because there is no path from $A_k^c$ to $A_k$, and the resulting graph remains to be a DAG. We can add the other links in $L_k$ one-by-one back to the graph with the direction from $A_k$ to $A_k^c$; similarly, these link additions do not create cycles. The final directed graph is $D_{k+1}$, and it is a DAG. See Fig. 2-3 for an illustration.

The next lemma shows that the new DAG $D_{k+1}$ supports a lexicographically smaller optimal overload vector (and therefore potentially better throughput) than the DAG $D_k$.

Lemma 4. Let $D_k$ be a DAG with the maximum flow $f_k^{\text{max}} < \lambda \leq f^{\text{max}}$. The DAG $D_{k+1}$, obtained by performing Algorithm 1 over $D_k$, has the lexicographically minimum queue overload vector satisfying $q_{k+1}^{\text{min}} <_{\text{lex}} q_k^{\text{min}}$.

Proof of Lemma 4. Consider a link $(a, b) \in E_k$ such that $q_{k,a}^{\text{min}} = 0$ and $q_{k,b}^{\text{min}} > 0$; this link exists by Lemma 2. From the property (2.8), any feasible flow allocation $(f_{ij})$
that yields the lexicographically minimum overload vector \( q_k^{\text{min}} \) must have \( f_{ab} = 0 \) over link \((a, b)\). The link-reversal algorithm reverses the link \((a, b)\) so that \((b, a) \in E_{k+1}\) in the DAG \(D_{k+1}\). Consider the following feasible flow allocation \((f'_{ij})\) on the DAG \(D_{k+1}\):

\[
f'_{ij} = \begin{cases} 
\epsilon & \text{if } (i, j) = (b, a) \\
0 = f_{ji} & \text{if } (i, j) \neq (b, a) \text{ but } (j, i) \text{ is reversed} \\
& \text{if } (i, j) \text{ is not reversed}
\end{cases}
\]

where \( \epsilon < q_k^{\text{min}}_{b} \) is a sufficiently small value. In other words, the flow allocation \((f'_{ij})\) is formed by reversing links and keeping the previous flow allocation \((f_{ij})\) except that we forward an \( \epsilon \)-amount of overload traffic from node \( b \) to \( a \). Let \( \bar{q} = (\bar{q}_n)_{n \in N} \) be the
\( q_{a}^{\text{min}} = 0 \quad q_{b}^{\text{min}} > 0 \quad \hat{q}_{a} = \epsilon \quad \hat{q}_{b} = q_{b}^{\text{min}} - \epsilon \)

(a) Link \((a, b)\) before the link reversal. (b) Link \((b, a)\) after the link reversal.

Figure 2-4: A link \(\{a, b\}\) in the network in Fig. 2-3 before and after link reversal. Before the reversal, the flow \(f_{ab}\) is zero on \((a, b)\). After the reversal, an \(\epsilon\) flow can be sent over \((b, a)\) so that \((\hat{q}_{a}, \hat{q}_{b}) <_{\text{lex}} (q_{k,a}^{\text{min}}, q_{k,b}^{\text{min}})\), while the rest of the flow allocation remains the same.

resulting queue overload vector. We have

\[
\begin{align*}
\hat{q}_{b} &= q_{k,b}^{\text{min}} - \epsilon < q_{k,b}^{\text{min}}, \quad \hat{q}_{a} = \epsilon > q_{k,a}^{\text{min}} = 0, \quad \text{and} \\
\hat{q}_{n} &= q_{k,n}^{\text{min}}, \quad n \notin \{a, b\}.
\end{align*}
\]

Therefore, \(\hat{q} <_{\text{lex}} q_{k}^{\text{min}}\) (see Fig. 2-4 for an illustration). Let \(q_{k+1}^{\text{min}}\) be the lexicographically minimal overload vector in \(D_{k+1}\). It follows that \(q_{k+1}^{\text{min}} \leq_{\text{lex}} \hat{q} <_{\text{lex}} q_{k}^{\text{min}}\), completing the proof. \(\square\)

**Theorem 1.** Suppose the traffic demand is feasible in \(G\), i.e., \(\lambda \leq f_{\text{max}}\), and the routing policy induces the overload vector \(q_{k}^{\text{min}}\) at every iteration \(k\). Then, the link-reversal algorithm will find a DAG whose maximum flow supports \(\lambda\) in a finite number of iterations.

**Proof of Theorem 1.** The link-reversal algorithm creates a sequence of DAGs \(\{D_{0}, D_{1}, D_{2}, \ldots, D_{k}\}\), where \(q_{k}^{\text{min}} = 0\). From Lemma 4, we know that a strict improvement in the lexicographically minimal overload vector is made after each iteration, i.e.,

\[
q_{0}^{\text{min}} >_{\text{lex}} q_{1}^{\text{min}} >_{\text{lex}} q_{2}^{\text{min}} >_{\text{lex}} \cdots
\]

The lexicographically minimal overload vector is unique in a DAG by Lemma 1, the DAGs \(\{D_{0}, D_{1}, D_{2}, \ldots, D_{k}\}\) must all be distinct. Since there are a finite number of unique embedded DAGs in a network, in a finite number of iterations the link-reversal algorithm will find a DAG \(D_{k}\) that has the lexicographically minimal overload vector
\( q_{k_i}^{\min} = 0 \) and the maximum flow \( f_{k}^{\max} \geq \lambda \). Note that such a DAG \( D_k \) exists because the undirected graph \( G \) has the maximum flow \( f^{\max} \geq \lambda \). 

Hence, when \( \lambda \leq f^{\max} \), the process of obtaining the lexicographically smallest overload vector and using Algorithm 1 to produce new DAGs eventually finds that DAG that supports the arrival rate.

### 2.2.4 Arrivals outside stability region

We show that even when \( \lambda > f^{\max} \), the link reversal algorithm will stop reversing the links in a finite number of iterations, and it will obtain the DAG that supports the maximum throughput \( f^{\max} \). We begin by examining the termination condition of our algorithm and show that if the algorithm stops at iteration \( k \), then the DAG \( D_k \) supports the max-flow of the network.

**Lemma 5.** Consider the situation when \( \lambda > f_{k}^{\max} \). If there is no link \((i, j)\) such that \( q_{k_i}^{\min} = 0 \) and \( q_{k_j}^{\min} > 0 \), then \( f^{\max} = f_{k}^{\max} \) and \( \lambda > f^{\max} \). That is, if there are no links to reverse at iteration \( k \), and \( q_{k_i}^{\min} > 0 \), then the throughput of \( D_k \) is equal to \( f^{\max} \).

**Proof of Lemma 5.** Let \( A_k \) be the set of overloaded nodes under a flow allocation that induces the lexicographically minimal overload vector \( q_{k_i}^{\min} \) in the DAG \( D_k \). We know that \((A_k, A_k^c)\) is a min-cut of the network from Lemma 7 (in the appendix), so

\[
\text{cap}_k(A_k, A_k^c) = f_{k}^{\max}.
\]

Suppose the link reversal algorithm stops after iteration \( k \), i.e. at iteration \( k \) there are no links to reverse. In this situation, there is no link \((i, j)\) such that \( q_{k_i}^{\min} = 0 \) and \( q_{k_j}^{\min} > 0 \), so by property (9), all the links between \( A_k \) and \( A_k^c \) go from \( A_k \) to \( A_k^c \). The capacity of the cut \((A_k, A_k^c)\) is given by

\[
\text{cap}_k(A_k, A_k^c) = \sum_{i \in A_k, j \in A_k^c} c_{ij}.
\]

This is equal to the capacity of the cut \((A_k, A_k^c)\) in the undirected network \( G \). So
\[ f_{\text{max}} \leq \text{cap}_k(A_k, A_k^c) = f_k^{\text{max}}. \] Because \( f_k^{\text{max}} \) cannot be greater than \( f_{\text{max}} \), \( f_k^{\text{max}} = f_{\text{max}} \). By assumption \( \lambda > f_k^{\text{max}} \), so \( \lambda > f_{\text{max}} \).

When \( \lambda > f_{\text{max}} \), this lemma shows that the link reversal algorithm stops only when the DAG achieves the maximum throughput of the network. Hence, if the DAG doesn't support the maximum throughput, then there exists a link that can be reversed. After each reversal, Lemma 3 holds, so the directed graph obtained after the reversal is acyclic. We can modify Lemma 4 to show that every reversal produces a DAG that supports an improved lexicographically optimal overload vector. We can combine these results to prove the following theorem.

**Theorem 2.** Suppose the traffic demand is not feasible in \( G \), i.e., \( \lambda > f_{\text{max}} \), and the routing policy induces the overload vector \( q_k^{\text{min}} \) at every iteration \( k \). Then, the link-reversal algorithm will find a DAG whose maximum flow supports \( f_{\text{max}} \) in a finite number of iterations.

### 2.3 Distributed Dynamic Algorithm

In the previous sections we developed a link reversal algorithm based on the assumption that we have a routing policy that lexicographically minimized the overload vector \( q_k^{\text{min}} \). The algorithm used \( q_k^{\text{min}} \) to identify the cut \( (A_k, A_k^c) \), then reversed all the links that went from the nodes in \( A_k^c \) to the nodes in \( A_k \). We then used the properties of lexicographical minimization to show that repeating this process for some iterations results in a DAG that supports the arrival rate \( \lambda \). We note that any algorithm that can identify this cut \( (A_k, A_k^c) \) can be used to perform the reversals, regardless of whether or not it minimizes the overload vector, because it would perform the exact same reversals as the one in Algorithm 1.

In this section, we develop a new method for identifying the cut \( (A_k, A_k^c) \) using the backpressure routing algorithm and perform the link-reversals with it. Then we use the results from the previous section to claim that this process also obtains the optimal DAG.
We begin by showing that the cut \((A_k, A_k^c)\) is a unique min-cut of the DAG defined below as the *smallest min-cut*. An example to illustrate this concept is given in Figure 2-5. We will then develop a threshold based algorithm that uses the queue backlog information of backpressure to identify this min-cut. This will enable us to perform the same reversal that we performed in the previous section without computing the lexicographically minimal overload vector.

**Definition 1.** We define the smallest min-cut \((X^*, X^{*c})\) in the DAG \(D_k\) as the min-cut with the smallest number of nodes in the source side of the cut, i.e., \((X^*, X^{*c})\) solves

\[
\begin{align*}
\text{minimize:} & \quad |X| \\
\text{subject to:} & \quad (X, X^c) \text{ is a min-cut of } D_k.
\end{align*}
\]

![Smallest min-cut](image)

Figure 2-5: Different min-cuts in a unit capacity line network. While every min-cut creates a bottleneck for the network, the smallest min-cut is the first bottleneck and has the smallest number of nodes in the source side of the cut.

The algorithm starts by creating an initial DAG \(D_0\) using the method presented in Section 2.2.1. Then, we use the backpressure algorithm to route the packets from the source to the destination over \(D_0\). Let \(Q_n(t)\) be the queue length at node \(n\) in slot \(t\). The backpressure algorithm can be written as in Algorithm 2. It simply sends packets on a link \((i, j)\) if node \(i\) has more packets than \(j\).

Since backpressure is throughput optimal [66], if the arrival rate is less than \(f_0^{\text{max}}\), then all queues are stable. If the arrival rate is larger than \(f_0^{\text{max}}\), the system is unstable and the queue length grows at some nodes. In this case, the next lemma shows that if we were using a routing policy that produced the optimal overload vector \(q_{k}^{\text{min}}\), the set of all the overloaded nodes \(A_k\) and the non-overloaded nodes \(A_k^c\) form the smallest min-cut of the DAG \(D_k\).
Algorithm 2 Backpressure algorithm (BP)

1: for all \((i, j) \in E_k\) do
2: \hspace{1em} if \(Q_i(t) \geq Q_j(t)\) then
3: \hspace{1em} Transmit \(\min\{c_{ij}, Q_i(t)\}\) packets from \(i\) to \(j\)
4: \hspace{1em} end if
5: \hspace{1em} Update \(Q_i(t)\)
6: end for
7: Update \(Q_j(t)\)

Lemma 6. Let \(A_k\) be the set of overloaded nodes under a flow allocation \((f_{ij})\) that induces the lexicographically minimum overload vector in the DAG \(D_k\). If \(|A_k| > 0\), then \((A_k, A_k^c)\) is the unique smallest min-cut in \(D_k\).

Proof of Lemma 6. The proof is in Appendix 2.9.2.

Essentially, at every iteration, the link reversal algorithm of Section 2.2 discovers the smallest min-cut \((A_k, A_k^c)\) of the DAG \(D_k\) and reverses the links that go from \(A_k^c\) to \(A_k\). Now the following theorem shows that the backpressure algorithm can be augmented with some thresholds to identify the smallest min-cut.

Theorem 3. Assume that \((A_k, A_k^c)\) is the smallest min-cut for DAG \(D_k\) with a cut capacity of \(f_{\text{max}}^k = \text{cap}(A_k, A_k^c) < \lambda\). If packets are routed using the backpressure routing algorithm, then there exist finite constants \(T\) and \(R\) such that the following happens:

1. For some \(t < T\), \(Q_n(t) > R\) for all \(n \in A_k\), and
2. For all \(t\), \(Q_n(t) < R\) for \(n \in A_k^c\).

Proof of Theorem 3. We will prove the two claims separately. The proof will use the fact that the smallest min-cut forms the first bottleneck for the DAG \(D_k\) which will overload \(A_k^c\) and prevent backlog to build in \(A_k^c\). The detailed proofs for both claims are given in the Appendix 2.9.3.

Each node \(n\) has a threshold-based smallest min-cut detection mechanism. When we start using a particular DAG \(D_k\), in each time-slot, we check whether the queue
crosses a pre-specified threshold $R_k$. Any queue that crosses the threshold gets marked as overloaded. After using the DAG $D_k$ for $T_k$ timeslots, all the nodes that have their queue marked overloaded form the set $A_k$. When the time $T_k$ and threshold $R_k$ are large enough, the cut $(A_k, A_k^c)$ is the smallest min-cut as proven in Theorem 3. After determining the smallest min-cut, an individual node can perform a link reversal by comparing its queue’s overload status with its neighbor’s. All the links that go from a non-overloaded node to an overloaded node are reversed to obtain $D_{k+1}$. The complete LFBP algorithm is given in Algorithm 3.

**Algorithm 3** LFBP (Executed by node $n$)

1: Input: sequences $\{T_k\}, \{R_k\}$, unique ID $n$
2: Generate initial DAG $D_0$ by directing each link $\{n,j\}$ to $(n,j)$ if $n < j$, to $(j,n)$ if $j > n$.
3: Mark the queue $Q_n$ as not overloaded
4: Initialize $t \leftarrow 0$, $k \leftarrow 0$
5: while true do
6: Use BP to send/receive packets on all links of node $n$
7: if $(Q_n(t) > R_k)$ then
8: Mark $Q_n$ as overloaded.
9: end if
10: $t \leftarrow t + 1$
11: $T_k \leftarrow T_k - 1$
12: if $T_k = 0$ then
13: Reverse all links $(j,n)$ such that $Q_j$ is not overloaded and $Q_n$ is overloaded.
14: $k \leftarrow k + 1$
15: Mark $Q_n$ as not overloaded
16: end if
17: end while

This algorithm simply adds link reversal to BP, hence the complexity of LFBP is just the sum of the BP and link-reversal algorithm. At each time-slot the link reversal algorithm checks whether a node is overloaded, so the computation required is $O(|N|)$. BP requires $O(|E|)$ computation at each time-slot because the algorithm computes the differential backlog for each link. Hence, the total computation required by the network for one time-slot of LFBP is $O(|N| + |E|)$.

Finally we give the following corollary that shows that the LFBP algorithm finds
the optimal DAG.

**Corollary 1.** Suppose the traffic demand is feasible in $G$, i.e., $\lambda \leq f_{\text{max}}$. Then, the LFBP algorithm (Algorithm 3) will find a DAG, whose maximum flow supports $\lambda$, in a finite number of iterations.

*Proof of Corollary 1.* Theorem 3 shows that LFBP identifies the smallest min-cut $(A, A^c)$ for the DAG $D_k$. Lemma 6 shows that $A$ is the set of overloaded nodes, and $A^c$ is the set of non-overloaded nodes in a flow allocation that induces the lexicographically minimal overload vector. LFBP reverses the links going from $A^c$ to $A$, which is also the reversals performed by the link reversal algorithm (Algorithm 1). Hence, by Theorem 1, LFBP obtains the DAG that supports $\lambda$. □

Good choices for the thresholds $T_k$ and $R_k$ are topology dependent. When the value of $R_k$ is too small, nodes that are not overloaded might cross the threshold producing a false positive. If the value of $R_k$ is large but $T_k$ is small, the overloaded nodes might not have enough time to develop the backlog to cross $R_k$ which produces false negatives. Hence, a good strategy is to choose a large $R_k$ so that the non-overloaded nodes don’t (or rarely) cross this threshold, then chose a large $T_k$ such that the overloaded nodes have enough time to build the backlog to cross $R_k$. Optimizing these thresholds requires further research. Note that our algorithm performance degrades graciously with false positives/negatives. Even when it detects the smallest min-cut incorrectly, the actions of the algorithm preserve the acyclic structure. Thus, in the subsequent iterations the algorithm can improve the DAG again.

### 2.3.1 Multi-source single destination networks

There are many scenarios, e.g. sensor networks, when several nodes in the networks need to send data to a central destination. The Gafni-Bertsekas link-reversal algorithm in [25] was also designed for this situation. In such networks, Algorithm 3 obtains a DAG that supports the given arrivals, provided that the arrivals are supportable by the undirected network. We can see this by transforming the multi source network into an equivalent single source network.
Let us consider a network with arrival of rate \( \lambda_n \geq 0 \) at node \( n; \lambda_d = 0 \). We can do the following transformation to convert this network into a single-source single-destination network where the result of Corollary 1 holds. We create a fictitious source \( s' \) with an arrival rate of \( \lambda_{s'} = \sum_n \lambda_n \) then add fictitious links \((s', n)\) with capacity \( \lambda_n \) for all \( n \). We know that Algorithm 3, finds a DAG that supports the arrival rate \( \lambda_{s'} \) in this modified network. The only way to stabilize this network is to have an arrival of rate \( \lambda_n \) on each node \( n \). Hence, this DAG must also stabilize the multi-source network.

### 2.3.2 Preventing dead ends

There can be several DAGs that support a given arrival rate in a particular undirected network. Some of these DAGs can include dead-ends, i.e. a node that has no path to the destination. When a packet reaches such a node, it gets stuck within the network forever. Moreover, having dead ends cannot improve the throughput of a network. If we perform a flow allocation, for the optimal throughput, none of the flows can pass through a dead end node. In Algorithm 3, the dead end nodes either never receive any packets because they are unreachable from the source, or they receive minimal packets after some time because they build a high backlog. In either case, this algorithm achieves the required throughput in the long run. Nevertheless having dead ends is an undesired phenomenon, and we would like to avoid it.

To remove dead-ends, we propose to use the Gafni-Bertsekas link reversal once the network is stable. When the source node detects that it is no longer overloaded, it can broadcasts a message to all the nodes informing them to perform such a reversal. We know that the Gafni-Bertsekas link-reversal obtains a DAG where all the nodes have a path to the destination, hence, it results in a dead-end free DAG. Also, from Proposition 2 in [25], we also know that any node that has a path to the destination does not perform a reversal at any point of the algorithm. That is, all the existing paths from a node to the destination stay intact during the iterations. This says that the algorithm does not decrease the throughput of the network from the source to the destination. Hence, it will produce a dead end free DAG that also supports the
2.3.3 Algorithm modification for topology changes

In this section we consider networks with time-varying topologies, where several links of graph $G$ may appear or disappear over time. Although the DAG that supports $\lambda$ depends on the topology of $G$, our proposed policy LFBP can adapt to the topology changes and efficiently track the optimal solution. Additionally, the loop free structure of a DAG is preserved under link removals.

To handle the appearance of new links in the network smoothly, we will slightly extend LFBP to guarantee the loop free structure. For a DAG $D_k$, every node $n$ stores a unique state $x_n(k)$ representing its position in the topological ordering of the DAG $D_k$. The states are maintained such that they are unique and all the links go from a node with the lower state to a node with the higher state. When a new link $\{i, j\}$ appears we can set its direction to go from $i$ to $j$ if $x_i(k) < x_j(k)$ and from $j$ to $i$ otherwise. Since this assignment of direction to the new link is in alignment with the existing links in the DAG, the loop-free property is preserved.

The state for each node $n$ can be initialized using the unique node ID during the initial DAG creation, i.e. $x_n(0) = n$. Then whenever a reversal is performed the state of node $n$ can be updated as follows:

$$x_n(k) = \begin{cases} 
  x_n(k-1) - 2^k \Delta, & \text{if } n \text{ is overloaded}, \\
  x_n(k-1), & \text{otherwise}.
\end{cases}$$

Here, $\Delta$ is some constant chosen such that $\Delta > \max_{i,j \in N} x_i(0) - x_j(0)$. Note that this assignment of state is consistent with the way the link directions are assigned by the link reversal algorithm. The states for the non-overloaded nodes are unchanged, so the links between these nodes are unaffected. Also, the states for all the overloaded nodes are decreased by the same amount $2^k \Delta$, so the direction of the links between the overloaded nodes is also preserved. Furthermore, the quantity $-2^k \Delta$ is less than the lowest possible state before the $k$th iteration, so the overloaded nodes have a lower
state than the non-overloaded nodes. Hence, the links between the overloaded and non-overloaded nodes go from the overloaded nodes to the non-overloaded nodes.

In this scheme, the states $x_n$ decrease unboundedly as more reversals are performed. In order to prevent this, after a certain number of reversals, we can rescale the states by dividing them by a large positive number. This decreases the value of the state while maintaining the topological ordering of the DAG. The number of reversals $k$ can be reset to 0, and a new $\Delta$ can be chosen such that it is greater than the largest difference between the rescaled states.

2.4 Complexity analysis

To understand the number of iteration the link-reversal algorithm takes to obtain the optimal DAG, we analyze the time complexity of the algorithm.

**Theorem 4.** Let $C$ be a vector of the capacities of all the links in $E$, and let $I$ be the set of indices $1, 2, ..., |E|$. Define $\delta > 0$ to be the smallest positive difference between the capacity of any two cuts. Specifically, $\delta$ is the solution of the following optimization problem

$$\min_{A, B \subseteq I} \sum_{a \in A} c_a - \sum_{b \in B} c_b$$

subject to: $\sum_{a \in A} c_a > \sum_{b \in B} c_b$.

The number of iterations taken by the link reversal algorithm before it stops is upper bounded by $|N| \left\lfloor \frac{f_{\text{max}}}{\delta} \right\rfloor$, where $f_{\text{max}}$ is the max-flow of the undirected network.

**Proof of Theorem 4.** After each iteration of the link-reversal algorithm, either the max-flow of the DAG increases, or the max-flow stays the same and the number of nodes in the source side of the smallest min-cut increases (see Lemma 8 in the Appendix). We can bound the number of consecutive iterations such that there is no improvement in the max-flow. In particular, every such iteration will add at least one node to the source set. So, it is impossible to have more than $|N| - 2$ such iteration.
Hence, every $|N|$ iterations we are guaranteed to have at least one increase in the max-flow.

Max-flow is equal to the min-cut capacity, and min-cut capacity is defined as the sum of link capacities. Say, the max-flow of DAG $D_{k+1}$ is greater than that of $D_k$. Let $A$ be the set of indices (in the capacity vector $C$) of the links in the min-cut of $D_{k+1}$, and $B$ be the set of indices of the links in the min-cut of $D_k$. This choice of $A$ and $B$ forms a feasible solution to the optimization problem given in the theorem statement. Since the optimal solution $\delta$ lower bounds all the feasible solutions in the minimization problem, the increase in the max-flow must be greater than or equal to $\delta$.

Every $|N|$ iteration the max-flow increases at least by $\delta$. Hence, the DAG supporting the max-flow $f_{\text{max}}$ is formed within $\lceil |N| f_{\text{max}} / \delta \rceil$ iterations. \hfill $\square$

**Corollary 2.** In a network where all the link capacities are rational with the least common denominator $D \in \mathbb{N}$, the number of iterations is upper bounded by $(|N|D f_{\text{max}})$.

*Proof of Corollary 2.* Since the capacities are rational we can write the capacity of the $i^{th}$ link as $c_i = \frac{N_i}{D}$, where $N_i$ is a natural number. From the definition of $\delta$ in Theorem 4, we get $\delta$ to be the value of the following optimization problem:

$$\min_{A,B \subseteq \mathbb{N}} \frac{1}{D} \left( \sum_{a \in A} N_a - \sum_{b \in B} N_b \right)$$

subject to: $\sum_{a \in A} N_a > \sum_{b \in B} N_b$.

All the $N_{(.)}$ are integers, so to satisfy the constraint we must have the difference $\sum_{a \in A} N_a - \sum_{b \in B} N_b \geq 1$. Hence $\delta \geq \frac{1}{D}$. Using this value of $\delta$ in Theorem 4, we can see that the number of iterations is upper bounded by $(|N|D f_{\text{max}})$. \hfill $\square$

**Corollary 3.** In a network with unit capacity links, the number of iterations the link-reversal algorithm takes to obtain the optimal DAG is upper bounded by $|N||E|$.\hfill $\square$

*Proof of Corollary 3.* The max-flow $f_{\text{max}} \leq |E|$. So, by Corollary 2, the number of iterations is upper bounded by $|N||E|$. \hfill $\square$
Figure 2-6: CDF of the number of iterations taken by the link reversal algorithm to obtain the optimal DAG for Erdős-Rényi networks with \(|N|\) nodes.

We conjecture that these upper bounds are not tight, and finding a tighter bound will be pursued in the future research. We simulated the link reversal algorithm in Erdős-Rényi networks \((p = 0.5)\) with \(|N| = 10, 20, \ldots, 50\). For each \(|N|\) we generated \(10^6\) different graphs and randomly assigned capacities to the links. The link reversal algorithm started with a random initial DAG. We found that it took less than 2 iterations on average to find the optimal DAG. A plot of the empirical CDF is given in Figure 2-6. We also performed similar simulations with completely connected graphs with random link capacities. This experiment produced similar results. It took less than 2 iterations on average and a maximum of 5 iterations to find the optimal DAG.

A worst case lower bound for the number of iteration is \(|N|\). This lower bound can be achieved in a line network where the initial DAG has all of its links in the wrong direction.

### 2.5 Simulation Results

We compare the delay performance of the LFBP algorithm and the BP algorithm via simulations. We will see that the network with the LFBP routing has a smaller backlog on average under the same load. This shows that the LFBP algorithm has
a better delay performance. We consider two types of networks for the simulations: a simple network with fixed topology, and a network with grid topology where the links appear and disappear randomly.

2.5.1 Fixed topology

We consider a network with the topology shown in Figure 2-7(a). The edge labels represent the link capacities. The undirected network has the maximum throughput of 15 packets per time slot. Figure 2-7(b) shows the initial DAG $D_0$. Instead of running the initial DAG algorithm of Section 2.2.1, here we choose a zero throughput DAG to test the worst-case performance of LFBP. The arrivals to the network are Poisson with rate $\lambda = 15\rho$, where we vary $\rho = .5, .55, ..., .95$. For the LFBP algorithm, we set the overload detection threshold to $R_k = 60$ for all $n, k$. To choose this parameter, we observed that the backlog buildup in normal operation rarely raises above 60 at any non-overloaded node. We also choose the detection period $T_1 = 150$ and $T_k = 50$ for all $k > 1$. This provides enough time for buildup, which improve the accuracy of the overload detection mechanism.

We simulate both algorithms for one million slots, using the same arrival process sample path. Figures 2-7(c) - 2-7(e) show the various DAGs that are formed by the LFBP algorithm at iterations $k = 1, 2, 3$. We can see that the nodes in the smallest min-cut get overloaded and the link reversals gradually improve the DAG until the throughput optimal DAG is reached.

Figure 2-8 compares the total average backlog in the network for BP and LFBP, which is indicative of the average delay. A significant delay improvement is achieved by LFBP, for example at load 0.5 the average delay is reduced by 66%. We observe that the gain in the delay performance is more pronounced when the load is low. In low load situations, the network doesn’t have enough “pressure” to drive the packets to the destination and so under BP the packets go in loops. Figure 2-9 shows the evolution of the average backlog over time for a specific load of 0.5. We can see that the backlog grows until $t = T_1 = 150$ because the initial DAG has zero throughput. After the reversals start the backlog decreases and converges around the average
Figure 2-7: Figure (a) depicts the original network. Figures (b)-(e) are the various stages of the DAG. The red nodes represent the overloaded nodes, and the dashed line shows the boundary of the overloaded and the non-overloaded nodes.

backlog of 30.

Figure 2-8: Average backlog in the network (Fig. 2-7(a)) with fixed topology for the Loop Free Backpressure (LFBP) and the Backpressure (BP) algorithms.
2.5.2 Randomly changing topology

To understand the delay performance of the LFBP algorithm on networks with randomly changing topology, we consider a network where 16 nodes are arranged in a $4 \times 4$ grid. All the links are taken to be of capacity six. For the LFBP algorithm, we choose a random initial DAG with zero throughput shown in Figure 2-10. The source is on the upper left corner (node 1) and the destination is on the bottom right (node 16).

![Initial DAG for the LFBP algorithm](image)

Figure 2-10: Initial DAG for the LFBP algorithm chosen so that the LFBP needs several iterations to reach the optimal DAG. All the links have capacity six.
In the beginning of the simulations all 24 network links are activated. At each time slot an active link fails with a probability $10^{-4}$ and an inactive link is activated with a probability $10^{-3}$. The maximum throughput of the undirected network without any link failures is 12. Clearly on average, each link is "on" a fraction $\frac{10}{11}$ of the time, and thus the average maximum throughput of the undirected network with these link failure rates is $\frac{10}{11} \times 2 \times 6 = 10.9$. The arrivals to the networks are Poisson with rate $\lambda = 10.9\rho$, where $\rho = .1, .2, ..., .6$. For the LFBP algorithm, the detection threshold is set to $R_k = 100$ and the detection period is $T_k = 30$ for all $n, k$. These parameters were chosen so that there are several reversals before a topology change occurs in the undirected network. The simulation was carried out for a million slots.

Figure 2-11 compares the average backlog of LFBP and BP. In the low load scenarios LFBP reduces delay significantly (by 85% for load = 0.1) even though the topology changes challenge the convergence of the link-reversal algorithm. As the load increases, both the algorithms begin to obtain a similar delay performance.

![Figure 2-11: Average backlog in the network with random link failures (Fig. 2-10) for the Loop Free Backpressure algorithm and the Backpressure algorithm.](image)

2.6 Comparison with Enhanced Backpressure

We compare the performance of LFBP against the Enhanced Backpressure (EBP) algorithm from [53]. EBP aims to reduce the delay by sending more traffic on the
shorter paths. This is accomplished by including pre-computed lengths of the shortest path in the weight calculation. EBP is very similar to the algorithm in [62].

![Ring network topology](image)

Figure 2-12: Ring network topology. All links are bidirectional with unit capacity. Traffic goes from node 1 to node 10.

To compare the performance of LFBP and EBP, we simulate these algorithms in a network with a bidirectional ring topology as shown in Figure 2-12. The network has only one commodity going from node 1 to node 10. To reach the destination, the packets can either traverse the nodes 1, 2, ..., 10, or they can use the direct link (1,10).

The results of the experiment is given in Figure 2-13. We can see that LFBP performs better than EBP for higher loads. In order to support a high load in this topology, the longer path (1,2,...,10) must be used. LFBP uses both long and the short path equally, however EBP tries to send most of its traffic through the link (1,10). Note that under EBP, even a packet that has reached node 3, 4 or 5 prefers to use the path through the link (1,10) as this is the shorter path to node 10. EBP performs better for low loads because these loads can be supported by using just the shortest path which requires only one hop.

It is easy to see that LFBP doesn't always perform better than EBP, even under high load. LFBP spreads the traffic throughout the network whereas EBP concentrates it on the shorter paths. Therefore, if a network can be stabilized without using the longer paths, EBP would perform better. We will see this situation in the next section where we simulate these algorithms on a grid network.
2.7 Multicommodity simulation

We extend the link reversal algorithm to the networks with multiple commodities. The multi-commodity algorithm is identical to the single commodity algorithm, with the exception that we now use the multicommodity backpressure of [66]. Each node $n$ maintains a queue $Q_y^n(t)$ for each commodity $y$. Each commodity is assigned its own initial DAG. A pseudocode for the multicommodity LFBP that we used is given in Algorithm 4. An important direction for future research is to determine whether
the claims proven for a single commodity in the previous sections extend to the multicommodity case.

Algorithm 4 Multicommodity LFBP (Executed by $n$)

1: Input: sequences $\{T_k\}, \{R_k\}$, unique ID $n$
2: For each commodity $y$, generate initial DAG $D^y_0$ by directing \( \{n, j\} \) to \((n, j)\) if $n < j$, to \((j, n)\) if $j > n$.
3: Mark all queues $Q^y_n$ as not overloaded
4: Initialize $t \leftarrow 0, k \leftarrow 0$
5: while true do
6: Use Multicommodity BP to send/recv packets on all links of node $n$
7: for all $y$ do
8: if $(Q^y_n(t) > R_k)$ then
9: Mark this $Q^y_n$ as overloaded.
10: end if
11: end for
12: $t \leftarrow t + 1$
13: $T_k \leftarrow T_k - 1$
14: if $T_k = 0$ then
15: for all $y$ do
16: Reverse links $(j, n)$ in $D^y_k$ if $Q^y_j$ is not overloaded and $Q^y_n$ is overloaded.
17: end for
18: $k \leftarrow k + 1$
19: Mark all queues as not overloaded
20: end if
21: end while
22: end while

The link-reversal algorithm checks for the overload for each commodity on each node, so the computation required for the link-reversal in a $Y$-commodity network at each time-slot is $O(Y|N|)$. Also, multi-commodity BP requires $O(Y|E|)$ computation at each time-slot because it computes the differential backlog for each link for each commodity. So, the computation required by the network for one time-slot of Multicommodity LFBP is $O(Y(|N| + |E|))$.

For the simulation, we consider a network arranged in a $4 \times 4$ grid as shown in Figure 2-10. Each link has a capacity of 6 packets per time-slot. There are three commodities in the network defined by the source destination pairs (1,16), (4,13) and (5,8). For the LFBP algorithm, each commodity starts with the same initial DAG given in Figure 2-10.

57
We use the arrival rate vector $\lambda^\text{max} = [7.18, 6.96, 9.86]$, which is a max-flow vector for this network computed by solving a linear program. We scale this vector by various load factors $\rho$ ranging from 0.1 to 0.9. The arrivals for each commodity $i$ is Poisson with rate $\rho \lambda^i_{\text{max}}$. In the beginning of the LFBP simulation, $\lfloor 500/\rho \rfloor$ dummy packets are added to the source of each commodity. This is helpful in low load cases because it forces the algorithm to find a DAG with high throughput, and avoids stopping at a DAG that only supports the given (low) load. $R_k$ was chosen to be 50 and $T_k = 50$ for all $k > 0$. For the EBP simulation, the length of the shortest paths were scaled by the maximum link capacity, 6, in order to improve its performance as suggested in [53].

Figure 2-14 shows the average backlog and delay in the network for different loads under backpressure, enhanced backpressure and multicommodity LFBP. We can see that both LFBP and EBP have significantly improved delay performance compared to backpressure. We can also see that EBP outperforms LFBP. In a grid topology, most of the throughput can be obtained by using short paths. EBP keeps the traffic focused in these paths, whereas LFBP spreads the traffic throughout the network equally which causes higher delay.

2.8 Conclusion

Backpressure routing and link reversal algorithms have been separately proposed for time-varying communication networks. In this chapter we show that these two distributed schemes can be successfully combined to yield good throughput and delay performance. We develop the Loop-Free Backpressure Algorithm which jointly routes packets in a constrained DAG and reverses the links of the DAG to improve its throughput. We show that the algorithm ultimately results in a DAG that yields the maximum throughput. Additionally, by restricting the routing to this DAG we eliminate loops, thus reducing the average delay. Future investigations involve optimization of the overload detection parameters and studying the performance of the scheme on the networks with multiple commodities.
Figure 2-14: Average backlog and in a multicommodity network with fixed topology for BP, EBP and LFBP algorithms. Both EBP and LFBP performs significantly better than BP. In this topology, EBP performs better than LFBP because most of the throughput can be obtained by using the short paths.

2.9 Appendix

2.9.1 Lemma 7

**Lemma 7.** Consider a DAG \( D_k \) with source node \( s \), destination node \( d \), and arrival rate \( \lambda \). Let \( A_k \) be the set of overloaded nodes under the flow allocation \( (f_{ij}) \) that yields the lexicographically minimum overload vector. If \(|A_k| > 0\), then \((A_k, A_k^c)\) is a min-cut.

59
of the DAG $D_k$.

**Proof of Lemma 7.** First we show that $(A_k, A^c_k)$ is a cut, i.e., the source node $s \in A_k$ and the destination node $d \in A^c_k$. The destination node $d$ has zero queue overload rate $q_d = 0$ because it does not buffer packets; hence $d \in A^c_k$. We show $s \in A_k$ by contradiction. Assume $s \notin A_k$. The property (2.8) shows that there is no flow going from $A^c_k$ to $A_k$, i.e.,

$$\sum_{(i,j) \in E_k: i \in A^c_k, j \in A_k} f_{ij} = 0.$$ 

The flow conservation equation applied to the collection $A_k$ of nodes yields

$$\sum_{n \in A_k} q_n = \sum_{(i,n) \in E_k: n \in A_k} f_{in} - \sum_{(n,j) \in E_k: j \in A_k, n \in A_k} f_{nj} = -\sum_{(n,j) \in E_k: n \in A_k, j \in A^c_k} f_{nj} \leq 0,$$

which contradicts the assumption that the network is overloaded (i.e., $|A_k| > 0$).

Note that in the above equation $\lambda$ does not appear because of the premise $s \notin A_k$.

By the max-flow min-cut theorem, it remains to show that the capacity of the cut $(A_k, A^c_k)$ is equal to the maximum flow $f^\text{max}_k$ of the DAG $D_k$. Under the flow allocation $(f_{ij})$ that induces the lexicographically minimal overload vector, the throughput of the destination node $d$ is the maximum flow $f^\text{max}_k$ (see Lemma 1). It follows that

$$f^\text{max}_k = \lambda - \sum_{i \in N} q_i = \lambda - \sum_{i \in A_k} q_i$$

$$= \sum_{(i,j) \in E_k: i \in A_k, j \in A^c_k} f_{ij}$$

$$= \sum_{(i,j) \in E_k: i \in A_k, j \in A^c_k} c_{ij} = \text{cap}_k(A_k, A^c_k).$$

where (2.10) uses (2.7) and $q_i = 0$ for all nodes $i \notin A_k$, (2.11) follows the flow conservation law over the node set $A_k$, and (2.12) uses the property (2.9) in Lemma 1. \qed
2.9.2 Proof of Lemma 6

Proof of Lemma 6. Lemma 7 shows that \((A_k, A_k^c)\) is a min cut of the DAG \(D_k\). It suffices to prove that if there exists another min-cut \((B, B^c)\), i.e., \(A_k \neq B\) and \(\text{cap}_k(A_k, A_k^c) = \text{cap}_k(B, B^c)\), then \(A_k \subset B\). The proof is by contradiction. Let us assume that there exists another min-cut \((B, B^c)\) such that \(A_k \notin B\). We have the source node \(s \in A_k \cap B\) and the destination node \(d \in A_k^c \cap B^c\). Consider the partition \(\{C, D, E, F\}\) of the network nodes such that \(C = A_k \cap B, D = A_k \setminus B, E = B \setminus A_k\) and \(F = N \setminus (A_k \cup B)\) (see Fig. 2-15). Since \(A_k \not\subset B \) and \(A_k \neq B\), we have \(|D| > 0\). Also, we have \(s \in C\) and \(d \in F\). Let \((f_{ij})\) be a flow allocation that yields the lexicographically minimum overload vector in \(D_k\). Properties (2.8) and (2.9) show that

\[
\begin{align*}
    f_{ij} &= c_{ij}, \forall i \in A_k, j \in A_k^c, \quad \text{(2.13)} \\
    f_{ij} &= 0, \forall i \in A_k^c, j \in A_k. \quad \text{(2.14)}
\end{align*}
\]

The capacity of the cut \((B, B^c)\) in the DAG \(D_k\), defined in (2.1), satisfies

\[
\text{cap}_k(B, B^c) = \text{cap}_k(B, D) + \text{cap}_k(B, F), \quad \text{(2.15)}
\]

where \(B^c = D \cup F\). Under the flow allocation \((f_{ij})\), we have

\[
\text{cap}_k(B, D) = \sum_{(i,j) \in E_k: i \in B, j \in D} c_{ij} \geq \sum_{(i,j) \in E_k: i \in B, j \in D} f_{ij}. \quad \text{(2.16)}
\]
Applying the flow conservation equation to the collection of nodes in $D$ yields

$$\sum_{(i,j) \in E_k : i \in B, j \in D} f_{ij} \geq \sum_{i \in D} q_i + \sum_{(i,j) \in E_k : i \in D, j \in F} f_{ij}.$$  \hspace{1cm} (2.17)

In (2.17), the first term is the sum of incoming flows into the set $D$; notice that there is no incoming flow from $F$ to $D$ because of the flow property (2.14). The second term is the sum of queue overload rates in $D$. The last term is a partial sum of outgoing flows leaving the set $D$, not counting flows from $D$ to $B$; hence the inequality (2.17). From the flow property (2.13), the outgoing flows from the set $D$ to $F$ satisfy

$$\sum_{(i,j) \in E_k : i \in D, j \in F} f_{ij} = \sum_{(i,j) \in E_k : i \in D, j \in F} c_{ij}.$$  \hspace{1cm} (2.18)

Combining (2.15)-(2.18) yields

$$\text{cap}_k(B, B^c) = \text{cap}_k(B, D) + \text{cap}_k(B, F)$$

$$\geq \sum_{i \in D} q_i + \sum_{(i,j) \in E_k : i \in D, j \in F} c_{ij} + \text{cap}_k(B, F)$$

$$> \sum_{(i,j) \in E_k : i \in D, j \in F} c_{ij} + \text{cap}_k(B, F)$$

$$= \text{cap}_k(A_k \cup B, F),$$ \hspace{1cm} (2.19)

where the second inequality follows that all nodes in $D$ are overloaded and $q_n > 0$ for all $n \in D$. Inequality (2.19) shows that there exists a cut $(A_k \cup B, F)$ that has a smaller capacity, contradicting that $(B, B^c)$ is a min-cut in the DAG $D_k$. Finally, we note that the partition $(A_k, A_k^c)$ is unique because the lexicographically minimal overload vector is unique by Lemma 1.

\[\square\]

### 2.9.3 Proof of Theorem 3

**Proof of the first claim.** First we will show that the queue at the source $Q_s(t)$ crosses any arbitrary threshold $R_1$. We know that for some node $n \in A_k$, $Q_n(t) \to \infty$ as
$t \to \infty$ because the external arrival rate to the source $s \in A_k$ is larger than the rate of departure from set $A_k$, i.e. $\lambda > \text{cap}(A_k, A_k^c)$. The backpressure algorithm sends packets on a link $(i,j)$ only if $Q_i(t) > Q_j(t)$. Hence, at any time-slot if a node $b \neq s$ has a large backlog, then one of its parents $p$ must also have a large backlog. $Q_p$ can be slightly smaller than $Q_b$ because $Q_b$ might also receive packets from other nodes at the same time-slot. Specifically, $Q_p(t) > Q_b(t+1) - \sum_i c_{ib}$. Performing the induction on the parent of $p$ we can see that the source node must have a high backlog when any node in $A_k$ develops a high backlog. Note that the network is a DAG and the node $n$ received packets form the source to develop its backlog, so the induction much reach the source node. Hence, when $Q_b(T_1) \gg R_1$, $Q_s(t) > R_1$ for some $t < T_1$.

Now we will show that every node in $A_k$ crosses the threshold $R$. Let $B_1 \subseteq A_k$ be the set of nodes such that $Q_n(t) > R_1$ for some time $t < T_1$. We showed that $s \in B_1$. We will show that when $B_1 \neq A_k$, there exists some set $B_2$, such that (i) $B_1 \subset B_2$, and (ii) for every node $n \in B_2$, $Q_n(t) > R_2$ for some $t < T_2$. Here, $R_2$ and $T_2$ are large thresholds.

Assume $B_1 \neq A_k$. Let $C_1 = A_k \setminus B_1$, i.e all nodes in $C_1$ haven’t crossed the threshold $R_1$ until time $T_1$. Let $c_{B_1C_1}$ be the total capacity of the links going from $B_1$ to $C_1$, and $c_{C_1A_k^c}$ be the total capacity of the links going from $C_1$ to $A_k^c$. We have $c_{B_1C_1} > c_{C_1A_k^c}$ because $(A_k, A_k^c)$ is the smallest min-cut (see Figure 2-16). When the backlogs of the nodes of $B_1$ are much larger than the nodes of $C_1$, the nodes in $C_1$ receive packets from $B_1$ at the rate of $c_{B_1C_1}$ packets per time-slot, and no packets are sent in the reversed direction. The rate of packets leaving the nodes in $C$ is upper bounded by $c_{B_1A_k^c}$ which is smaller than the incoming rate. Hence, at least one node $n' \in C$ must collect a large backlog, say larger than $R_2 < R_1$. So, each node in the set $B_2 = B_1 \cup \{n'\}$ have a backlog larger than $R_2$ at some finite time $T_2$.

Now using induction we can see that for $B_m$ where $m < |A_k|$, $B_m = A_k$ and all the nodes in $B_m$ cross a threshold $R = \min\{R_1, \ldots, R_m\}$ by time $T = \max\{T_1, \ldots, T_m\}$. \[ of the second claim. \]

We will use the following fact to prove this claim: for any subset of nodes $S$, if the number of packets entering $S$ is lower than or equal to the number of packets leaving $S$ on every time-slot, then the total backlog in $S$ doesn’t grow. So,
Let \((A_k, A_k^c)\) be the smallest min-cut. We showed that \(s \in B_1\). Say, 
\[c_{C_1A^c} \geq c_{B_1C_1}\] then the cut \((B_1, B_1^c)\) has the capacity of 
\[c_{B_1A^c} + c_{B_1C_1} \leq \text{cap}(A_k, A_k^c)\].
This contradicts the assumption that \((A_k, A_k^c)\) is the smallest min-cut. So, 
\[c_{C_1A_k^c} < c_{B_1C_1}\].

the backlog in each node of \(S\) is bounded.

Assume a node \(b\) develops a backlog \(Q_b(t) > R_1\). Here \(R_1\) is a chosen such that

\[R_1 = |A_k^c| \sum_{i,j \in A_k^c} c_{ij} + \max_{n \in A_k^c} Q_n(0).\]

Consider a subset \(B\) of \(A_k^c\) such that for every node \(i \in B\) and \(j \in C = A_k^c \setminus B\), 
\[(Q_i(t) - Q_j(t)) > c_{ij}\]. The sets \(B\) and \(C\) must be nonempty because \(Q_b(t)\) is large and \(Q_d(t)\) is zero, that is \(b \in B\) and \(d \in C\). Note that backpressure doesn’t send any data from \(C\) to \(B\).

Let \(c_{AB}\) be the capacity of the links going from \(A\) to \(B\), and let \(c_{BC}\) be the capacity of the links going from \(B\) to \(C\). So, the number of packets entering \(B\) at timeslot \(t\)
is upper bounded by \( c_{AB} \). The number of packets leaving \( B \) is equal to \( c_{BC} \). Since \( (A, A^c) \) is the smallest min-cut, \( c_{AB} \leq c_{BC} \) (see Figure 2-17). Hence, the number of packets entering \( B \) is less than or equal to the number of packets leaving it at time \( t \).

Therefore as soon as one of the nodes crosses threshold \( R_1 \), the sum backlog becomes bounded. We can choose a threshold \( R \gg R_1 \) such that this threshold is never crossed by any nodes in \( A_k^c \).

\[ \Box \]

2.9.4 Lemma 8

**Lemma 8.** Consider the case when \( \lambda > f_k^{\max} \). The link reversal algorithm is applied on DAG \( D_k \) to obtain \( D_{k+1} \). Let \( (A_k, A_k^c) \) and \( (A_{k+1}, A_{k+1}^c) \) be the smallest min-cuts of \( D_k \) and \( D_{k+1} \) respectively. Then, either \( \cap k(A_k, A_k^c) > \cap k+1(A_{k+1}, A_{k+1}^c) \), or \( \cap k(A_k, A_k^c) = \cap k+1(A_{k+1}, A_{k+1}^c) \) and \( |A_{k+1}| > |A_k| \)

![Diagram](image URL)

Figure 2-18: Here \( l_i \) represents the sum of the capacities of the links going from one partition to the next in the DAG \( D_k \), and \( l'_i \) represents the sum of the link capacities in the DAG \( D_{k+1} \). For example, \( l_9 \) and \( l'_9 \) represent the links that go from \( (A_k \cup A_{k+1})^c \) to \( (A_k \cap A_{k+1}) \) in DAGs \( D_k \) and \( D_{k+1} \) respectively.

**Proof.** Consider the partitioning of the nodes as shown in Figure 2-18. For \( i = 1, \ldots, 12 \), \( l_i \) represents the sum of the capacities of the links going from one partition to the next in the DAG \( D_k \), and \( l'_i \) represents the sum of the link capacities in the DAG \( D_{k+1} \). The capacities of the smallest min-cut, before and after the reversal are
given by
\[ \text{cap}_k(A_k, A_k^c) = l_2 + l_5 + l_{10} + l_{12} \]

\[ \text{cap}_{k+1}(A_{k+1}, A_{k+1}^c) = l_4' + l_7' + l_{10}' + l_{11}' \]
respectively. Note that only the links that are coming into \( A_k \) are different in \( D_k \) and \( D_{k+1} \). So
\[ l_i = l_i' \text{ for } i = 3, 4, 7, 8. \] (2.20)

Because of the reversal there are no links coming into \( A_k \) in the DAG \( D_{k+1} \):
\[ l_1', l_6', l_9', l_{11}' = 0. \] (2.21)

After the reversal, the incoming links to \( A_k \) become outgoing from \( A_k \),
\[ l_{10}' = l_{10} + l_9. \] (2.22)

(Thermgponding equations for \( l_2', l_5' \) and \( l_{12}' \) are omitted because they are not necessary for the proof). Since \( (A_k, A_k^c) \) is a min-cut,
\[ l_5 \leq l_7. \] (2.23)

This is true because otherwise the cut \( (A_k \cup A_{k+1}, (A_k \cup A_{k+1})^c) \) in the DAG \( D_k \) has a smaller capacity then the min cut \( (A_k, A_k)^c \). Specifically, let us assume \( l_5 > l_7 \). Then, we get the contradiction:
\[
\text{cap}_k(A_k \cup A_{k+1}, (A_k \cup A_{k+1})^c) = l_2 + l_7 + l_{10} \\
< l_2 + l_5 + l_{10} + l_{12} \\
= \text{cap}_k(A_k, A_k)^c
\]

First we will show that if \( A_k \setminus A_{k+1} \neq \phi \), then the capacity of the DAG must have increased. The proof is by contradiction.

66
Let us assume that the throughput didn’t increase. So,

\[ cap_k(A_k, A_k^c) \geq cap_{k+1}(A_{k+1}, A_{k+1}^c) \]
\[ = l'_4 + l'_7 + l'_{10} + l'_{11} \]
\[ = l_4 + l_7 + l_{10} + 0 \]  \hspace{1cm} (2.24)
\[ \geq l_4 + l_5 + l_{10} \]  \hspace{1cm} (2.25)
\[ = cap_k(A_k \cap A_{k+1}, (A_k \cap A_{k+1})^c). \]  \hspace{1cm} (2.26)

(2.24) follows from (2.20) and (2.21), and (2.25) follows from (2.23). Since \( A_k \setminus A_{k+1} \neq \phi \) by assumption, \(|A_k| > |A_k \cap A_{k+1}|\). This leads to a contradiction, because in DAG \( D_k \) the cut \((A_k \cap A_{k+1}, (A_k \cap A_{k+1})^c)\) is smaller than the smallest min-cut \((A_k, A_k^c)\).

Hence, \( cap_k(A_k, A_k^c) < cap_{k+1}(A_{k+1}, A_{k+1}^c) \).

Next, we will consider the case \( A_k \setminus A_{k+1} = \phi \). Using (2.23),

\[ cap_k(A_k, A_k^c) = l_5 + l_{10} \leq l_7 + l_{10}. \]

In this situation, we again have two cases. First, if \( A_k = A_{k+1} \) we know that \( l'_{10} > l_{10} \) and \( l_7 = 0 \). Hence, \( cap_k(A_k, A_k^c) < l'_{10} = cap_{k+1}(A_{k+1}, A_{k+1}^c) \).

Second, if \( A_k \subset A_{k+1} \), then \(|A_k| > |A_{k+1}|\) and

\[ l'_{10} \geq l_{10}. \]  \hspace{1cm} (2.27)

Using (2.20) and (2.27) \( cap_k(A_k, A_k^c) \leq cap_{k+1}(A_{k+1}, A_{k+1}^c) \). \( \square \)
Chapter 3

Distributed Algorithm for Throughput Optimal Routing in Overlay Networks

Optimal routing algorithms\(^1\) have received a significant amount of attention in the literature for the past two decades (e.g. [66, 6, 52, 46]), however, they have had limited success in terms of implementations. One of the main reasons behind the lack of traction is that these policies require additional functionalities that are not supported by the legacy devices. For example, most of these algorithms need the network to be composed of homogeneous nodes that possess the ability to implement a dynamic routing policy. In contrast, many legacy networks use a single path routing scheme with a work-conserving forwarding policy such as FIFO, and hence can support only a fraction of the achievable throughput. Thus, an implementation of a throughput optimal scheme usually requires a complete overhaul of the network. An overlay architecture for a gradual move towards optimal routing was proposed in [37]. This architecture integrates overlay nodes capable of dynamic routing into an underlay network of legacy devices (see Figure 3-1 for an example). In this chapter, we develop a throughput optimal dynamic routing algorithm for such overlay networks.

---

\(^1\)A routing algorithm is throughput optimal if it can stabilize any traffic that can be stabilized by some routing algorithm.
Overlay networks have been used to improve the performance and capabilities of computer networks for a long time. The Internet itself started as a data network built on top of the telephone network. An overlay architecture to improve the robustness of the Internet was proposed in [3], where alternate overlay paths are used to overcome path loss in the underlay network. Placement for the overlay node to improve path diversity was studied in [31]. Architectures for designing overlay networks that improve different quality of service metrics have been proposed in [45], [63]. Currently overlay is being used extensively for applications such as content delivery, multicast, etc.

In [37], the authors study the problem of placing the minimum number of overlay nodes into an existing underlay in order to maximize network throughput. In particular, the authors show that with just a few overlay nodes, maximum network throughput can be achieved. However, [37] also shows that the backpressure routing algorithm of [66], which is known to be optimal in a wide range of scenarios, leads to a loss in throughput when used in an overlay network. Then the authors of [37] proposes a heuristic for optimal routing called the Overlay Backpressure Policy (OBP). An optimal backpressure like routing algorithm for a special case, where the underlay paths do not overlap with each other, was given in [37]. This paper also proposes a threshold based heuristic for general overlay networks. The schemes presented in [37]
and [57] are very similar and were conjectured to be throughput optimal.

In this chapter, we provide a counterexample to show that OBP is in fact not throughput optimal and develop a new optimal routing policy. To derive the optimal policy, we notice that the suboptimality of backpressure arises from its failure to accurately account for congestion in the underlay paths. Traditional backpressure doesn’t keep track of the packets in the underlay which can lead the overlay nodes to send too many packets into the underlay creating instability of underlay queues. We will first develop a centralized solution which achieves optimality by limiting the traffic injected into the underlay so that the underlay queues are always bounded. Then we use the intuition gained from this policy to develop a distributed solution which uses the queue backlog information in the underlay to compute the amount of flow transmitted into each underlay path. This policy implicitly favors underlay paths that are less congested and preserves stability of all the queues.

This chapter is organized as follows. We describe our model in the next section. In section III, we provide a counterexample to the OBP routing policy. Then in section IV, we provide a centralized stochastic policy that is throughput optimal for overlay networks. In section V, we develop a distributed policy based on the dual subgradient descent method that requires the underlay queue-lengths. In section VI, we propose three approaches to estimating the queue-lengths if they are not available to the overlay nodes. Finally, we verify the performance of our algorithm with extensive simulations.

3.1 Model

We model the network as a graph \((N, E)\) where \(N\) is the set of nodes and \(E\) is the set of directed links. The links are capacitated and the capacity of a link \((i, j) \in E\) is given by \(c_{ij}\). The nodes can be of two types: underlay or overlay. We represent the set of all underlay nodes by \(U\) and the set of all overlay nodes by \(\mathcal{O} = N \setminus U\). The network supports a set of commodities, \(K\), where each commodity \(k \in K\) is defined by a source-destination pair. For the ease of exposition, we will formulate the
problem with all the sources and destinations being overlay nodes. In Section 3.4.4, we discuss how the same solution can be applied when this is not the case. The time is slotted and indexed by \( t \). We remove the time index for notational simplicity if removing it doesn’t create ambiguity.

### 3.1.1 Overlay

The overlay network consists of the controllable nodes \( \mathcal{O} \) which are capable of implementing a dynamic routing algorithm. The links between two overlay nodes can either be a direct edge or a path through the underlay referred to as a tunnel. A tunnel \( l \) is a sequence of nodes \( l_1, l_2, \ldots, l_{|l|} \) where \( |l| \) is the length of the tunnel. We represent the set of all the tunnels in the network by \( L \).

Since a tunnel connects two overlay nodes, \( l_1 \) and \( l_{|l|} \) are overlay nodes, and \( l_{|l|}, \ldots, l_{|l|-1} \) are underlay nodes. When a packet is sent into a tunnel \( l \), node \( l_1 \) encapsulates it into a packet destined to node \( l_{|l|} \) and forwards it onto the underlay node \( l_2 \). The route taken by the tunnel is dictated by the path from \( l_2 \) to \( l_{|l|} \) which is assigned by the underlay. When the packet reaches \( l_{|l|} \), it is decapsulated and enqueued at the node. An example of the different type of links in an overlay network is given in Figure 3-2. This overlay network consists of one direct link \((1,4)\) and three tunnels \((1,3,4), (2,3,4)\) and \((2,3,5)\).

Each overlay node \( i \) maintains a queue for each commodity \( k \) and the backlog is represented by \( Q^k_i \). The number of external commodity \( k \) packets that arrive at node \( i \) represented by \( A^k_i \). Let \( F^k_l \) represents the amount of packets injected into the tunnel \( l \), and \( \bar{F}^k_l \) represents the number of packets that exit tunnel \( l \). The quantities are different because a packet sent into the tunnel might not exit the tunnel for several time-steps. Let \( F^k_{ij} \) represent the number of commodity \( k \) packets that are transmitted on an overlay to overlay link \((i,j)\). Figure 3-2 illustrates the meaning of these variables on a simple network. The backlog of commodity \( k \) packets at overlay
(a) Example of overlay and underlay queues in the physical network. The variables labelling the edges represent the number of packets transmitted for each commodity on each tunnel. 

(b) Corresponding overlay network. Each tunnel in the overlay network is represented by a sequence of nodes traversed by it.

Figure 3-2: A physical network and its corresponding overlay network.

Node $i$ evolves as follows:

$$Q_k^i(t+1) = \left[ Q_k^i(t) - \sum_{j \in O} F_{ij}^k(t) - \sum_{l \in L : l_1 = i} F_l^k(t) + \sum_{j \in O} F_{ji}^k(t) + \sum_{l \in L : l_\| = i} \bar{F}_l^k(t) + A_l^k(t) \right]^+$$

Here, $\{l \in L : l_1 = i\}$ are all the tunnels that start at node $i$, $\{l \in L : l_\| = i\}$ are the tunnels that end at node $i$, and $[.]^+ = \max(.,0)$. Packets are removed at the destination node, hence the backlog of a commodity at its destination is zero.

We assume that all the traffic arrivals $A_l^k$ are i.i.d. with a mean of $\lambda_l^k$. We also assume that the arrival rate vector $\lambda$ is in the interior of the throughput region of the overlay network $\Lambda$ [37]. We will be designing a dynamic routing policy that controls $F_l^k$ and $F_{ij}^k$ at each time-step so that both the overlay and the underlay queues stabilize.
3.1.2 Underlay

The underlay network consists of the uncontrollable nodes $U$. These nodes have a static routing policy which assigns a fixed path between each pair of nodes in the underlay. The paths are assumed to be acyclic and unique, which ensures that all the tunnels are acyclic and that they take a fixed route through the underlay.

An underlay node maintains a queue per outgoing link. The backlog on the queue associated with the link $(a, b)$ is represented by $Q_{ab}$. The queues have infinite buffer space hence packets are not dropped. When a packet arrives at an underlay node, the node looks up the link assigned to it based on its destination and enqueues it on the corresponding link. Since several tunnels of the overlay network can pass through the same underlay link the underlay queues accumulates packets from several different tunnels and commodities. An example of an underlay queue that is shared by several tunnels is presented in Figure 3-2. Packets from both the tunnels $(1,3,4)$ and $(2,3,4)$ are queued on the link $(3,4)$.

The underlay employs a work-conserving forwarding scheme that is “universally stable” as defined in [4]. This assumption ensures that if the number of packets injected into the underlay at each timeslot satisfies the capacity constraints of the tunnels, then the underlay queues are deterministically bounded. Specifically, under a universally stable forwarding policy, an underlay queue corresponding to link $(a, b)$ is always deterministically bounded if

$$\sum_{l \in L : (a, b) \in l} \sum_{k} F^k_l(t) < c_{ab} \forall t. \quad (3.1)$$

Here \( \{l \in L : (a, b) \in l\} \) is the set of tunnels that pass through the link $(a, b)$. We refer to such constraints as the tunnel capacity constraints. Several work-conserving policies that are universally stable are given in [4].
3.2 Background

The problem of optimal routing in an overlay network was first studied in [37], where it was shown that backpressure routing, which is known to be throughput optimal in a range of scenarios, is not optimal for overlay networks, and proposed a heuristic called the Overlay Backpressure Policy (OBP). The OBP heuristic was conjectured to be throughput optimal.

For each tunnel $l$ and commodity $k$ OBP keeps track of the packets in flight $H_l^k$, which is the number of packets that have been transmitted into the tunnel by node $l_1$ but haven’t reached node $l_{|l|}$. The weight for each commodity over the tunnel $W_l^k(t)$ is computed as follows

$$W_l^k(t) = Q_{l_1}^k(t) - H_l^k(t) - Q_{l_{|l|}}^k(t).$$

A link $(i, j)$ that connects two overlay nodes can be thought of as a tunnel $l = (i, j)$ with no underlay node, hence the weight is computed as

$$W_l^k(t) = Q_i^k(t) - Q_{i_{|l|}}^k(t).$$

Then, the commodity with the highest weight sends its packets into the tunnel provided that the weight is positive. A precise description of the OBP is given in Algorithm 5.

This policy makes sense intuitively because it encourages utilizing the tunnels that have less packets in them. When a tunnel is congested, the number of packets in flight is high, which encourages the overlay nodes to use alternate routes and send packets into the tunnel only when the backlog in the overlay is extremely high. This behavior is common to backpressure-based optimal routing algorithms. Moreover, OBP reduces to backpressure routing when all the nodes are overlay nodes.

We present the following counterexample to show that the OBP is not throughput optimal. Consider a network topology given in Figure 3-3(a) where all the links are unit capacity. There are three commodities with source $s_i$ and destination $d_i$. 


Algorithm 5 Overlay Backpressure Policy (OPB):

For each tunnel $l$ at each time-step $t$:

1. Compute the commodity $k^*$ that maximizes the weight $W_l^k(t)$,

\[ k^* \in \arg \max_k W_l^k(t). \]

Ties are broken arbitrarily.

2. Transmit $\mu$ packets into the tunnel where

\[ \mu = \begin{cases} c_{i,l_2} & \text{if } W_l^{k^*}(t) > 0 \\ 0, & \text{otherwise,} \end{cases} \]

where $c_{i,l_2}$ is the capacity of the first link of tunnel $l$.

$i = 1, 2, 3$. The source and the destination are overlay nodes, whereas the nodes 1, 2 and 3 (in gray) are underlay nodes. The underlay nodes use the FIFO queuing discipline\(^2\). Each commodity in this network has two tunnels to the destination, e.g. $(s_1, 1, 2, d_1)$ and $(s_1, 3, 1, 2, d_1)$. Note that the shorter tunnels do not share any links between them. So, if the shorter tunnel is chosen by each commodity, this network can support an arrival rate vector of $[1, 1, 1]$.

\(^2\)From [67] we know that FIFO is throughput optimal for a ring which is the underlay topology in this example.

Figure 3-3: Counterexample for throughput optimality of the Overlay Backpressure Policy of [37].

Let us consider Poisson arrivals with the rate vector of $[0.8, 0.8, 0.8]$, which is
clearly inside the stability region. To support this rate OBP has to send most of its traffic through the shorter tunnels. However, as we show below, congestion can lead traffic to use longer tunnels, which leads to instability. A simulation result showing this instability is given in Figure 3-3(b).

This instability is caused by overlapping tunnels where congestion in one tunnel forces commodities to use the longer tunnels which in turn leads to more congestion. Consider the situation where the number of packets in flight is large for tunnel \((s_1, 1, 2, d_1)\). So, commodity 1 traffic is routed through the tunnel \((s_1, 3, 1, 2, d_1)\). This means that the link \((3,1)\) is being used by commodity 1 packets, which creates congestion for commodity 3 over tunnel \((s_3, 3, 1, d_3)\) forcing commodity 3 traffic onto tunnel \((s_3, 2, 3, 1, d_3)\). This problem continues for the tunnels used by commodity 2, which in turn create congestion for the tunnels of commodity 1 forcing its traffic onto tunnel \((s_1, 3, 1, 2, d_1)\) further exacerbating the situation. This cyclical nature of increased congestion makes all the commodities unstable.

### 3.3 Centralized solution

We begin by providing a centralized optimal routing policy for overlay networks. In this section we assume that the underlay topology is known, and a centralized controller can make the routing decisions at the overlay. The key to obtaining a throughput optimal policy is to realize that the underlay cannot make dynamic decisions, hence, the overlay necessarily has to take into account the capacities of the underlay links while making scheduling decisions.

Our algorithm works by choosing the scheduling decision which minimizes the \(T\)-slot drift of the quadratic Lyapunov function of the overlay queues [27]. This is similar in spirit to the backpressure routing algorithm, which implements a schedule that minimizes the Lyapunov drift at every slot. In our set-up, multislot drift needs to be considered since packets that are sent into the tunnel take several timeslots to come out of the tunnel. In addition to minimizing the drift, we also have to make sure that the underlay queues are bounded. Because we assume that the underlay
forwarding scheme is universally stable, we are able to guarantee that underlay queues are bounded once the tunnel capacity constraints (3.1) are met. Thus the algorithm seeks a scheduling decision that minimizes the drift subject to the tunnel capacity constraints.

To simplify the notation, in this section, a link between two overlay nodes will be viewed as a tunnel which does not comprise of any underlay nodes. We divide the time into $T$-slot duration frames and consider minimizing the $T$-slot drift. At the beginning of each frame, we solve the optimization problem (3.2) in a centralized fashion. The solution to (3.2) minimizes the drift of a quadratic Lyapunov function, while simultaneously satisfying the tunnel capacity constraints. The solution gives us $F_{l}^{k*}$, which is the number of packets commodity $k$ must send into tunnel $l$ in order to minimize the drift. A complete description of the policy is given in Algorithm 6.

**Algorithm 6 Centralized Policy**

1. At the beginning of each frame solve the following optimization problem:

   $$F_{l}^{k*} = \arg \max_{F_{l}^{k}} \sum_{k,l} F_{l}^{k}[Q_{l}^{k}(t) - Q_{l}^{k}(t)]$$

   s.t. $\sum_{l \in L: (a,b) \in l} \sum_{k} F_{l}^{k} \leq c_{ab}, \forall (a,b) \in E$

   $F_{l}^{k} \geq 0$

2. Send $F_{l}^{k*}$ packets of commodity $k$ into tunnel $l$ each time slot in the frame. If $F_{l}^{k*}$ is not an integer, approximate it by sending $p$ packets every $q$ slots so that $\frac{p}{q} \approx F_{l}^{k*}$.

In the special case when a tunnel $l$ does not share any link with other tunnels, we see that $F_{l}^{k*}$ can be computed independently of the other tunnels. The commodity $k^*$ is the one with the highest differential backlog $Q_{l}^{k}(t) - Q_{l}^{k}(t)$ and $F_{l}^{k*}$ is chosen to be the capacity of the smallest link in the tunnel. Thus our algorithm resembles the backpressure routing except for the fact that the packets can face large delays while passing the underlay. However, when there is a shared link, all the tunnels that share the link are required to exchange their backlog information.
Next we show that the Algorithm 6 stabilizes the network queues if the arrival rate vector \( \lambda \) lies in the interior of the stability region. We use \( T \)-slot Lyapunov drift analysis to prove that these queues are strongly stable [27].

**Theorem 5.** For any arrival rate vector \( \lambda \) in the stability region \( \Lambda \) and a large enough frame length \( T \), the policy given in Algorithm 6 stabilizes all the queues in the network.

**Proof.** The proof of the theorem is given in the Appendix.

---

### 3.4 Fluid Formulation and Distributed Solution

The centralized policy in the previous section requires the knowledge of the underlay topology which might not be known to the overlay. Moreover, having a centralized controller is often impractical. We now consider the fluid model of the network and develop a decentralized policy. Fluid models have been successfully utilized to establish the stability of queueing networks (e.g. [12, 20]).

Let \( f_{ij}^k \) be the flow assigned to commodity \( k \) on the link \( (i, j) \in E \), and \( f_{il}^k \) be the flow assigned to commodity \( k \) on the tunnel \( l \in L \). Let \( f \) denote the vector containing all the flow variables. The arrival rate of commodity \( k \) at overlay node \( i \) is represented by \( \lambda_i^k \), and we assume that the vector of arrival rates \( \lambda \) is in the interior of the stability region. For simplicity, we will assume \( \lambda \) to be a constant, however, if it is time-varying, we note that the technical results hold as long as the arrival rate is bounded at each time-step and the expected value \( E[\lambda(t)] \) exists. The problem of stabilizing the network queues can be formulated as a linear program that finds a
feasible flow allocation on all the links and tunnels,

\[
\begin{align*}
\max & \quad 0 \\
\text{s.t.} & \quad \sum_{l:(i,j)\in l} \sum_{k} f_{lk}^k \leq c_{ij}, \forall (i,j) : i \in U, j \in N \quad (3.5) \\
& \quad \sum_{l:(i,j)\in l} \sum_{k} f_{lk}^k \leq c_{ij}, \forall (i,j) : i \in \mathcal{O}, j \in U \quad (3.6) \\
& \quad \sum_{j} f_{ij}^k + \sum_{l:d_l=i} f_{lk}^k - \sum_{j} f_{ji}^k - \sum_{l:d_l=j} f_{lk}^k - \lambda_i^k \geq 0, \forall i \in \mathcal{O}, k \quad (3.7) \\
& \quad \sum_{k} f_{ij}^k \leq c_{ij}, \forall i, j \in \mathcal{O} \quad (3.8) \\
& \quad f_{ij}^k, f_{lk}^k \geq 0, \quad (3.9)
\end{align*}
\]

Here, the inequalities (3.5) are the tunnel capacity constraints which are the fluid version of (3.1). Each one of these constraints correspond to an uncontrollable link, i.e. a link between two underlay nodes or a link that goes from underlay to an overlay node. Inequalities (3.6) are the link capacity constraints corresponding to the first link in the tunnel, i.e. the links that go from an overlay node to an underlay node. This link is responsible for controlling the rate received by the underlay links. Inequalities (3.7) are the flow conservation constraints on the overlay network. Note that the flow conservation constraints are not required for the underlay because for each tunnel \( l \), there is a single route and the flows coming into the underlay are feasible because of (3.5). That is, for a tunnel \( l \), when \( f \) is a feasible solution,

\[ f_{lk}^k = f_{l_{i_1}l_{j_2}}^k = f_{l_{i_2}l_{j_3}}^k = ... = f_{l_{i_{|l|-1}}l_{j_l}}^k. \]

Constraints (3.8) are the capacity constraints for the overlay links.

### 3.4.1 Dual problem

We now formulate the dual problem so that it can be solved with the subgradient descent method [47, 9]. Let \( q_{ij} \) and \( q_{il}^k \) denote the dual variables for the tunnel
constraints (3.5) and the flow conservation constraints (3.7) respectively, and let \( q \) represent the vector containing all the dual variables. The Lagrangian function is given by,

\[
L(f,q) = \sum_{(i,j) \in U} q_{ij} \left( c_{ij} - \sum_{l:(i,j) \in l} \sum_{k} f_{l}^{k} \right) + \sum_{i \in \mathcal{O},k} q_{i}^{k} \\
\left( \sum_{j} f_{ij}^{k} + \sum_{l:l_{1} = i} f_{l}^{k} - \sum_{j} f_{ji}^{k} - \sum_{l:l_{1} = i} f_{l}^{k} - \lambda_{i}^{k} \right) \\
= \sum_{i} \sum_{k} f_{i}^{k} \left( q_{i}^{k} - \sum_{(i,j) \in U} q_{ij} - q_{i}^{k} \right) + \sum_{(i,j) \in U} q_{ij} c_{ij} - \sum_{i \in \mathcal{O},k} q_{i}^{k} \lambda_{i}^{k},
\]

(3.10)

where the second equality is obtained by rearranging the terms so that the flow variables are factored out instead of the dual variables.

Let \( X \) be a set such that any \( f \in X \) satisfies the constraints (3.6), (3.8) and (3.9). Note that these constraints can be enforced locally by an overlay node using only locally available information. This property will be essential in designing the decentralized algorithm. The dual objective function corresponding to the problem (3.5) is

\[
D(q) = \max_{f \in X} L(f, q).
\]

The dual problem is given by,

\[
\min_{q} D(q) \quad \text{(3.11)}
\]

s.t. \( q \geq 0. \)

Since the primal problem (3.5) is a linear program, the duality gap is zero (Slater’s condition [9]). Hence, solution of the dual (3.11) yields a feasible flow allocation.
3.4.2 Distributed solution

The subgradient method works by initializing the dual variables with a value \( q(0) > 0 \), and then iterating on them until it converges to optimal \( q^* \). Each iteration involves computing a subgradient \( g \) of \( D \) at the current value of the dual variables, then updating the dual variables as follows:

\[
q(t + 1) = [q(t) - \alpha(t)g(t)]^+.
\] (3.12)

Here \( \alpha(t) \) is positive scalar step-size. The dual variables are known to converge to the optimal \( q^* \) if the step-sizes \( \alpha(t) \) are chosen appropriately. However, if \( \alpha(t) = \alpha \), then the iterates (3.12) converge to a bounded neighbourhood of \( q^* \) [9].

Let \( f_t^{k*} \) and \( f_{ij}^{k*} \) be the values of flow variables which maximize the Lagrangian \( L(f, q) \) over \( f \in X \) for a fixed \( q \), i.e. \( D(q) = L(f^*, q) \). From [9] we know that a subgradient of \( D(q) \) is given by a vector \( g \) with entries as,

\[
g_{ij} = c_{ij} - \sum_{l(i,j) \in \ell} \sum_k f_{ij}^{k*}, \text{ and} \tag{3.13}
\]

\[
g_i^k = \sum_j f_{ij}^{k*} + \sum_{l(i,j) \in \ell} f_t^{k*} - \sum_j f_{ji}^{k*} - \sum_{l(i,j) \in \ell} f_t^{k*} - \lambda_i^k. \tag{3.14}
\]

Now we can use the recursive equation (3.12) to update the dual variables.

The only necessary step that we haven’t covered so far is the computation of \( f_t^{k*} \) and \( f_{ij}^{k*} \). A careful observation of equation (3.10) and the set \( X \) shows that this is a simple optimization problem that can be solved in a decentralized fashion. The objective is a weighted sum of the flow variables, and the constraints that form \( X \) are the link capacity constraints. At a high level, for each overlay link, the solution chooses the maximum value of the flow variable that corresponds to the commodity with the highest positive weight. A complete algorithm to compute the optimal flow variables and update the dual variables is given in Algorithm 8.
3.4.3 Queue-lengths as dual variables

The subgradient descent algorithm presented in the Algorithm 8 requires the network to explicitly keep track of the dual variables. In order to implement the algorithm in a decentralized fashion, each underlay node \( i \) needs to maintain a dual variable \( q_{ij} \) for each link \((i, j)\), and each overlay node \( i \) needs to maintain a dual variable \( q_i^k \) for each commodity \( k \). This is a reasonable assumption for the overlay nodes, but not justified for the uncontrollable underlay. To get around similar problems of not having a dual variable, works such as [48], [46], etc. have proposed approximating them with the corresponding queue lengths. The argument behind this procedure is that the subgradients are proportional to the change in queue-lengths, so that the queue-lengths will move in the same direction as the dual variables. Next, we give an example in which this proportionality does not hold. In spite of this issue, we show that the queue-lengths can provide a good approximation for the dual variables.

We first observe that the dual variable update equations (3.16) and (3.17) are the same as the queue update equations when the flows sent into the tunnels \( f^k_l \) are feasible for the underlay, i.e. when no queues build up in the underlay. But when the flows do not satisfy the tunnel capacity constraints, the underlay queues build up, and the flows get reduced from their initial value as they pass through the bottleneck links. This decrease in the flow size is not captured in these dual variable update equations (3.16), (3.17). Consider the simple network shown in Figure 3-4. There is one commodity, \( k = 1 \), with source node 1 and destination node 4, and a single tunnel \( l = (1, 2, 3, 4) \). Suppose that at a certain iteration, \( q_1 > q_4^1 \), hence \( f^1_{k^*} = 3 \). This flow into the tunnel gets bottlenecked at link \((2, 3)\) so node 3 only receives a flow of 1. In this situation, equation (3.17) predicts that the queue-length for \( q_{34} \) would increase because a flow of size 3 was sent into the tunnel and the capacity of the link is 2, however this queue can only decrease or stay unchanged at 0.

![Figure 3-4: Link (3, 4) never builds a queue as the flow gets bottlenecked by (2, 3).](image)
To capture this reduction of the flow sizes in the tunnel, we model the queuing in the network as follows:

\[
\hat{q}_i^k(t + 1) = \left[ \hat{q}(t) - \alpha(t) \left( \sum_j f^k_{ij} + \sum_{t \mid t_i = i} f^k_{i} \right) - \sum_{j : (j, i) \in E} f^k_{ji} - \sum_{t \mid t_i = i} \epsilon_t^k(i) f^k_t - \lambda^k_t \right]^+ 
\]

(3.18)

\[
\hat{q}_{ij}(t + 1) = \left[ \hat{q}(t) - \alpha(t) \left( c_{ij} - \sum_k \sum_{t \mid (i, j) \in E} \epsilon_t^k(i, j) f^k_t \right) \right]^+ 
\]

(3.19)

where \( \epsilon_t^k(i), \epsilon_t^k(i, j) \in [0, 1] \) represent the reduction suffered by the corresponding flows before arriving at node \( i \). These quantities are implicitly determined by the network at each time-step depending on the scheduling policy in the underlay. In the example presented above, for any work conserving scheme, \( \epsilon_t^k(3, 4) = 1/3 \). We will show that for any value of \( \epsilon \) in the set \([0, 1]\) the queue-lengths will converge to the optimal dual variables. Let \( q \) be the true subgradient of \( D \) at \( q \), and \( \hat{g} \) be the approximate subgradient after the reduction, then we can represent the queuing equation as

\[
\hat{q}(t + 1) = [\hat{q}(t) - \alpha(t) \hat{g}(t)]^+, 
\]

and \( \hat{g} \geq g \).

Before we prove the convergence, we state the following preliminary lemma.

**Lemma 9.** The vector \( q^* = 0 \) is an optimal solution to the dual problem (3.11).

**Proof.** Since the objective of the primal problem is 0, a feasible solution to the primal is given by any feasible flow allocation \( f^k_{ij} \). Since \( q = 0 \) is a feasible dual solution, and any feasible \( f^k_{ij} \) together with \( q = 0 \) satisfy the complementary slackness condition (Theorem 4.5 in [11]), the proof follows. \( \square \)

This shows that the optimal solution corresponds to queue lengths equal to zero which makes, sense intuitively because any feasible flow allocation in the fluid domain doesn’t require queuing.
Let $G$ be a constant such that it bounds the Euclidean norm of the subgradients of the dual function $D(q)$ for all possible values of $q$, i.e. $G > \|g\|$. From equations (3.13)-(3.14), we can see that the subgradients are bounded because the flow variables are bounded by link capacities and arrival rates are bounded by assumption. So $G$ is finite. For simplicity we fix $\alpha(t) = 1$ and present the following convergence result.

**Theorem 6.** Let us approximate the dual variables $q$ with the queue-lengths $\hat{q}$ that evolve according to equations (3.18)-(3.19). Using the dual subgradient descent algorithm with $\alpha(t) = 1$, the queue lengths converge to the set $S = \{\hat{q} : D(\hat{q}) \leq \frac{1}{2}G^2\}$.

**Proof.** We will show that $\|\hat{q}(t + 1) - q^*\|^2 < \|\hat{q}(t) - q^*\|^2$ when $q(t)$ is outside the set $S$. Because $q^* = 0$ from Lemma 9, it suffices to show that $\|\hat{q}(t + 1)\|^2 < \|\hat{q}(t)\|^2$.

We have,

$$\|\hat{q}(t + 1)\|^2 = \|\hat{q}(t) - \hat{g}\|^2$$

Since $\hat{g} \geq g$,

$$\|\hat{q}(t + 1)\|^2 \leq \|\hat{q}(t) - g\|^2$$

$$= \|\hat{q}(t)\|^2 - 2\hat{q}(t)^T g + \|g\|^2$$

Our algorithm chooses $g$ to be a subgradient of $D(.)$ at $\hat{q}(t)$. So,

$$D(x) \geq D(\hat{q}(t)) + (x - \hat{q}(t))^T g, \forall x \in \mathbb{R}^m,$$

where $m$ is the dimension of $\hat{q}$. Taking $x = 0$,

$$D(\hat{q}(t)) \leq \hat{q}(t)^T g(f(t)^*)$$

So, $\|\hat{q}(t + 1)\|^2 \leq \|\hat{q}(t)\|^2 - 2D(\hat{q}(t)) + G^2$. Hence when, the $\hat{q}$ is far away from the optimal, specifically when $D(\hat{q}(t)) > \frac{1}{2}G^2$, it moves towards the optimum in the next time-step. \qed
Hence, we will use queue-lengths instead of the dual variables in the implementation of OORP. This will allow us to use the policy presented in Algorithm 8 without having to perform the dual variables update.

3.4.4 Underlay sources and destinations

The problem formulation given in beginning of Section V assumes that all the flows go from one overlay node to another. However, this assumption can be easily removed. Any flow that originates in the underlay be routed over a single path using the underlay routing scheme. Hence these flows can be represented simply as a reduction in the link capacities in the constraints (5) and (6) for the links traversed by this flow. Our algorithm stays unchanged because it is agnostic to the change in the link capacities at the underlay. OORP is also optimal when the underlay is a destination because destination nodes do not perform any routing.

3.4.5 Rate control

It is well known that subgradient descent is a general method to solve convex optimization problems. The dual gradient descent algorithm has been used derive distributed solutions to network utility maximization problems (e.g. [47, 46]). In the overlay network setting we can get a distributed solution to the utility maximization problem of the form:

$$\max \sum_{k \in K, i \in O} U_i^k(\lambda_i^k)$$

where $U_i^k(.)$ is concave and strictly increasing. In this setting, we can assume that there is an infinite backlog at the sources, and the rates $\lambda_i^k$ are chosen to maximize the total network utility.

We can use the same derivation technique as in section 3.4.2 to obtain a distributed algorithm. The algorithm is very similar to OORP with an added rate controller at each source. The rate control algorithm so obtained is standard, and the joint rate control and routing algorithm can be written as follows:
3.5 Unknown Underlay Queues

In the previous section we showed that the dual subgradient descent algorithm can be used to compute a feasible rate for each commodity on each link. We also showed that the queue lengths can be used to approximate the subgradient. However, typically legacy devices may not be able to send queue-lengths to the sources. In this section, we will present two approaches to estimate the required queue-length information. The first approach will estimate it using the delay experienced by the packets. The second approach will involve sending probe packets at a certain time intervals that collect the queue-length information in the tunnels.

3.5.1 Delay based approaches

From equation (3.15) it can be seen that in order to compute the subgradients we only need the total backlog in the tunnel, i.e. we don’t need the length of individual queues. A natural approach to estimate the total backlog in a tunnel is by using the time it takes for a packet to traverse it. To implement this method, each tunnel \( l \) maintains a delay variable \( D_l \). When a packet is sent into a tunnel, the sending node stamps the packet with the current time. When the packet exits the tunnel, the difference between the current time and the time-stamp on the packet is used to update \( D_l \). When computing the optimal flow variables, in equation (3.15) of OORP we substitute the sum of the underlay queues-lengths, \( \sum_{(a,b) \in L, a \in U} q_{ab} \), with \( D_l \).

For this approach, we assume that the underlay uses a FIFO queuing model which is a common forwarding scheme. Hence, this method requires no modification to the underlay. A similar approach has been used by TCP Vegas to solve a network utility maximization problem [48].

Although this approach is simple and does not require cooperation from the underlay, the queue-length estimates obtained by this method can be arbitrarily bad. Consider a FIFO queue that is empty at time zero. As shown in Figure 3-5(a), it has an incoming rate of 2 and outgoing capacity of 1. We want to estimate the queue-length at time \( t \) by using packet delays. To see the problem with this approach, let us
consider a situation when 2 packets arrive at the queue at every time-slot for the first $\tau$ time-slots, and no arrivals happen after that. In this situation, the actual queue length grows at the rate of 1 for the first $\tau$ time-slots, and then it decreases at the rate of 1 packet per time-slot until the queue is empty. On the other hand, the delay increases at the rate of $\frac{1}{2}$, and the last packet (that arrives at the $\tau$th time-slot) sees a delay of 100 because there are 99 packets in the queue at that time. So at time $2\tau$ when the queue is emptied, the packet received will have suffered a delay of $\tau$ time-slots giving a queue-length estimate of $\tau$, whereas the actual queue-length at that time is zero. Furthermore, the estimate stays bad until another arrival happens. This problem is illustrated in Figure 3-5(b). These arbitrarily bad estimates lead to sub-optimality of OORP which we will observe in the simulations.

A simple way to improve the estimate is to send empty probe packets when real packets are not available for some time period $T$. A similar approach has been shown to achieve throughput optimality in a special scenario in [56]. This approach quickly identifies when a queue becomes empty in the absence of new data packets, and the control algorithm can react accordingly. Although this approach corrects the estimate within $P$ time-slots, it still suffers from the arbitrarily bad estimation errors. As shown in Figure 3-5(c), at time $2\tau$ the estimate is $\tau$ whereas the actual queue-length is zero. Thus, we propose the following approach using explicit probes.
3.5.2 Priority probe approach

In this approach, we assume that the underlay nodes are capable of stamping the current queue-lengths into a special type of packets called the probe packets. We also assume that these packets are given higher priority compared to the data packets, and they do not consume link capacity because they are very small in size. These packets are generated by each tunnel at a fixed time-intervals $T$. When a probe packet exits a tunnel, the sum of the queue-lengths it has collected can be used to compute the optimal flow variables in Algorithm 8.

We can see that this approach results in a much more accurate estimation of the backlog compared to the delay based approach. However, the value of $T$ can have a significant impact on the performance on the algorithm. We will study its impact in the next section via simulations.

3.6 Simulation Results

We present several simulation results to evaluate the performance of the optimal overlay routing policy (OORP) given in Algorithm 8. First, we will ascertain that this algorithm is in fact optimal for the network in which the OBP policy of [37] was suboptimal. Then we evaluate the effect that different methods of estimating the queue-lengths have on the performance of the algorithm. Next we will study the performance of our algorithm when there is uncontrolled background traffic in the underlay. Finally, we will simulate the rate control algorithm to show that it achieves the maximum throughput.

3.6.1 OORP on the counterexample to OBP

We reconsider the network from Section III, shown in Figure 3-3(a). The network has three commodities and it can support a maximum arrival rate vector of $\lambda_{\text{sim}}^A = [1, 1, 1]$. We simulate the network under three different policies: the backpressure policy (BP), the overlay backpressure policy (OBP) and our policy, OORP. The
simulations are conducted at different loads \( \rho = 0.5, 0.55, ..., 1 \). For each policy the arrivals are Poisson distributed with rates \( \lambda = \rho \lambda_{\text{max}} \). The result of the simulations is given in Figure 3-6.

![Figure 3-6: Performance of different routing algorithms on the overlay network shown in Figure 3-3(a).](image)

The BP algorithm executed on the overlay network does not account for the underlay nodes and simply views a tunnel \( l \) as a link between two overlay nodes with capacity \( c_{1l_2} \). In the plot we can see that this algorithm becomes unstable around the load of 0.56. This is as expected because for each commodity the backpressure policy uses both tunnels equally since they have equal weights. The end node of both the tunnels is a destination, which has zero backlog, and BP does not account for the backlog in the underlay. So the weight for each tunnel of commodity \( k \) is equal to \( Q_{ik}^k \). When the algorithm uses the longer tunnel, the network becomes unstable for relatively low load.

The plot also shows that OBP is suboptimal and OORP achieves maximum throughput. We discussed the suboptimality of OBP in Section III. The main reason was that the OBP policy could not avoid using the longer tunnel which gave raise to a cycle of increased congestion. But in OORP, when the underlay queues-lengths are positive, the shorter tunnels have a higher weight than the longer tunnels. For example, for commodity 1, the weight of the shorter tunnel \((s_1, 1, 2, d_1)\) is \( Q_{s_1}^1 - Q_{12} - Q_{2d_1} \) and the weight of the longer tunnel \((s_1, 3, 1, 2, d_1)\) is \( Q_{s_1}^1 - Q_{31} - Q_{12} - Q_{2d_1} \). So when the underlay queues are large, the longer tunnel needs a lot more backlog at
the source than the shorter tunnel for its weight to be positive. This causes OORP to avoid using the longer tunnels when the network is congested.

3.6.2 Estimated Tunnel Backlog

We consider the network given in Figure 3-7 to observe the effect of estimating the backlog in the tunnels. In this network, all the links are bidirectional, composed of two unidirectional links. The links between an overlay and an underlay node have capacity 2 in each direction. All other links have unit capacity in both directions. We will simulate the network with two commodities. The first commodity is defined by the source-destination pair (1,3) and the second is defined by (2,4). For these commodities the network supports a max-flow vector of $A_{\text{sim}} = [2, 2]$. The simulation is performed at various load levels and the arrivals are Poisson distributed.

The underlay uses the shortest path routing hence creating a large number of available tunnels. Node 1 can send packets to node 3 directly via node 7 or 10 using the tunnels (1,7,5,6,3) and (1,10,7,5,6,3) respectively. However, these tunnels overlap, hence there is no benefit in using both of them. To achieve the throughput of two, node 1 must send its traffic through node 2 and have it forward it to node 3. Similarly node 2 must send some of its traffic through node 1 in order to achieve high throughput. Observing the organization of the tunnels in the network, we can see that using the wrong tunnel might cause the network to lose throughput. In addition, the tunnels form cycles in the overlay topology. These features make this topology challenging for a routing algorithm to achieve the optimal throughput.

The result of the simulations under different load levels is given in Figure 3-8. We can see that the delay approach, which uses packet delay as an estimate of the tunnel backlog, does not provide optimal throughput. Although probing the delay in the network with control packets improves the performance, it is still suboptimal. This happens because when the backlog is large the estimation error of this approach can also be large as described in Section 3.5.1.

We can also see that the probing approach achieves optimal throughput, and its performance is close to that of using the actual queue-lengths. The estimates obtained
by this approach are much more accurate than those from the delay approach because they are not affected by the amount of backlog in the network. When $T$ is increased, the stale estimate is used for a longer time period, so the performance of the algorithm degrades.

### 3.6.3 Background traffic

So far we have assumed that all the traffic in the network belongs to the overlay network. However, in real networks the underlay can be routing other traffic not generated by the overlay nodes. Next, we will study the performance of our algorithm
under such traffic. We expect the OORP to be throughput optimal under stable
background traffic in the underlay because such traffic can be thought of as a reduction
in the link capacities in inequalities (3.5) as described in Section 3.4.4.

We again consider the network from Figure 3-7 with two commodities (1,3) and
(2,4). We inject two flows of background traffic: first going from node 7 to 6 along
the path (7,5,6) with the arrival rate of 0.5, and second going from 8 to 14 along
the path (8, 9, 12, 14) with the arrival rate of 0.2. The arrivals happen according
to the Poisson process. Note that the first background flow blocks commodity 1’s
tunnel (1,7,4,5) and the second flow blocks commodity 2’s tunnel (2,12,14,13,4); both
the tunnels are essential for achieving the max flow vector \( \lambda_{\text{max}}^{\text{sim}B} \). This reduces the
maximum supportable arrival rates for the two commodities to \( \lambda_{\text{max}}^{\text{sim}C} = [1.5, 1.8] \).
Figure 3-9 shows the result of the simulation. We can see that all the approaches
except for the delay based approaches achieve the maximum throughput.

### 3.6.4 Rate control

To observe our rate controller at work, we consider the network from Section 3.6.3
with a minor modification as shown in Figure 3-10. We add a new overlay node 15
and a new commodity (15, 14) to the network. Node 15 connects to node 5 with a
directed link (15, 4) which has a unit capacity. We constrain the third commodity to
use the tunnel provided by the shortest path (5, 6, 9, 12, 14). For all three sources, the utility function is chosen to be $20 \log(\lambda)$ and $M_i^k = 20$. Note that the addition of the third commodity makes the simulation more challenging because the rate that maximizes total throughput is not the same as the rates that maximizes utility. We assume that the backlog information of each tunnel is available to the overlay nodes instantaneously.

![Figure 3-9: Performance of OORP in a network with background traffic.](image)

Constrained by the link capacities and the background traffic, the maximum throughputs for the commodities 1, 2, and 3 are 1.5, 1.8, and 1 respectively. However, the third commodity interferes with both commodities 1 and 2, hence the throughput

![Figure 3-10: Topology for the rate control experiment. The dotted lines show the tunnel assigned to the third commodity and the background traffic.](image)
of [1.5, 1.8, 1] is not achievable. From the plot in Figure 3-11 we can see that the throughput vector converges to [1, 1.3, 0.5] which maximizes the total utility. We can see that this throughput vector has a smaller sum than the sum of the maximum throughput supported in Section 3.6.3. That is, the network could have supported higher throughput by giving zero throughput to the third commodity, but that would have decreased the utility of the network.

![Figure 3-11](image)

Figure 3-11: Throughput achieved by the rate control algorithm with OORP converges to the rate that maximizes utility.

### 3.7 Conclusion

We showed that the existing algorithms for routing traffic in an overlay network are suboptimal, and developed a throughput optimal policy called the Optimal Overlay Routing Policy (OORP). This policy is distributed and can also be used with a rate controller to maximize network utility. Our algorithm requires the knowledge of congestion at the underlay, which might not be available to the overlay nodes. Hence we proposed different approaches to estimating underlay congestion. Simulations results show that OORP outperforms OBP and that estimating congestion using probing mechanism is effective. Future research will include obtaining better estimates for the congestion in the tunnels with minimum support from the underlay nodes and reducing the delay experienced by the packets in the network.
3.8 Appendix: Proof of Theorem 5

3.8.1 Stationary policy $\pi$

In order to prove the stability of the centralized policy, we need a stationary policy that stabilizes the network. For any arrival rates $\lambda$ such that $\lambda + \epsilon \in \Lambda$, $\epsilon > 0$, we know that there exist a feasible flow allocation vectors $(f_{l}^{k})_{l \in L, k \in K}$ such that for any overlay node $n$,

$$\sum_{l \in L: l=-1} f_{l}^{k} - \sum_{l \in L: l=1} f_{l}^{k} = \lambda_{n}^{k} + \epsilon. \quad (3.20)$$

This vector can be obtained by solving the multi-commodity flow problem. We assume that these flow variables can be closely approximated by rational numbers. So there exists integers $p_{l}^{k}$ and $q$ such that $f_{l}^{k} = p_{l}^{k}/q$. The time-slot are divided into $T$ slot long frames.

The policy $\pi$ simply sends $p_{l}^{k}$ amount of commodity $k$ packets every $q$ time-slots in each frame. Because the underlay is using a universally stable forwarding scheme and the burstiness constraints are satisfied, the underlay queues are deterministically bounded by a constant $B$ [4]. Also note that all the capacity constraints are satisfied every $q$ time-slots. Hence, $\pi$ stabilizes $\lambda$.

Let $F_{l}^{k}(t + \tau, \pi)$ represent the number of packets sent into tunnel $l$ by node $l_{1}$ at time $t + \tau$ under policy $\pi$. Let $\tilde{F}_{l}^{k}(t + \tau, \pi)$ represent the number of packets that are received by node $l_{|l|}$ at time $t$ from tunnel $l$ under policy $\pi$. Note that $F_{l}^{k}(t + \tau, \pi) = \tilde{F}_{l}^{k}(t + \tau, \pi)$ only if the tunnel $l$ is a direct link between two overlay nodes. If a tunnel passes through the underlay, it can take a bounded amount of time for the packets to exit the tunnel. Now, we prove the following lemma that will be used in proving the theorem.
Lemma 10. For the proposed randomized policy $\pi$

\[
E \left[ \sum_{j \in L : L_j = n} \sum_{\tau = 0}^{T-1} F^k_j(t + \tau, \pi) - \sum_{l \in L : L_l = n} \sum_{\tau = 0}^{T-1} F^k_l(t + \tau, \pi) \right] Q(t) \geq T(\lambda^k_n + \epsilon) - B, \forall n.
\]

Proof.

\[
E \left[ \sum_{j \in L : L_j = n} \sum_{\tau = 0}^{T-1} F^k_j(t + \tau, \pi) - \sum_{l \in L : L_l = n} \sum_{\tau = 0}^{T-1} F^k_l(t + \tau, \pi) \right] Q(t) = E \left[ \sum_{j \in L : L_j = n} \sum_{\tau = 0}^{T-1} F^k_j(t + \tau, \pi) - \sum_{l \in L : L_l = n} \sum_{\tau = 0}^{T-1} F^k_l(t + \tau, \pi) \right] \geq E \left[ \sum_{j \in L : L_j = n} \sum_{\tau = 0}^{T-1} F^k_j(t + \tau, \pi) - \sum_{l \in L : L_l = n} \sum_{\tau = 0}^{T-1} F^k_l(t + \tau, \pi) - B \right] = T \sum_{l \in L : L_l = n} \frac{p^k_l}{q} - T \sum_{l \in L : L_l = n} \frac{p^k_l}{q} - B = T \sum_{l \in L : L_l = n} f^k_l - T \sum_{l \in L : L_l = n} f^k_l - B = T(\lambda^k_n + \epsilon) - B.
\]

Here $B$ is a finite constant representing the maximum amount of backlog in the underlay network. We use this constant to obtain inequality (3.22) and equation (3.20) to obtain the last equality. \qed
3.8.2 Analysis of TBP

We know that the underlay queues are stable because the traffic injected into the tunnels satisfy the burstiness constraints and the underlay employs a universally stable forwarding policy [4]. Next we prove the stability of overlay queues.

The queue evolution of an overlay node $n$ can be written as:

$$Q_n^{k}(t + 1) = Q_n^{k}(t) - \sum_{l|l_1=n} F_l^{k}(t) + \sum_{l|l_1=n} \bar{F}_l^{k}(t) + A_n^{k}(t)$$

$$\leq \left[ Q_n^{k}(t) - \sum_{l|l_1=n} F_l^{k}(t) \right] + \sum_{l|l_1=n} \bar{F}_l^{k}(t) + A_n^{k}(t)$$

Here $F_l^{k}(t)$ represents the amount of packets injected into the tunnel $l$ at time $t$, and $\bar{F}_l^{k}(t)$ represents the number of packets that exit tunnel $l$ at time $t$.

Then the queue length after $T$ slots can be bounded as follows:

$$Q_n^{k}(t + T) \leq \left[ Q_n^{k}(t) - \sum_{l|l_1=n} \sum_{\tau=0}^{T-1} F_l^{k}(t + \tau) \right] +$$

$$\sum_{l|l_1=n} \sum_{\tau=0}^{T-1} \bar{F}_l^{k}(t + \tau) + \sum_{\tau=0}^{T-1} A_n^{k}(t + \tau)$$

$$\leq \left[ Q_n^{k}(t) - \sum_{l|l_1=n} \sum_{\tau=0}^{T-1} F_l^{k}(t + \tau) \right] +$$

$$\sum_{l|l_1=n} \sum_{\tau=0}^{T-1} \bar{F}_l^{k}(t + \tau) + \sum_{\tau=0}^{T-1} A_n^{k}(t + \tau) + B$$

The first inequality comes from considering all the arrivals and departures in a $T$-slot interval in a single slot. We get the second inequality by bounding the backlog in the underlay nodes by a constant $B$.

Now to prove the theorem, consider the quadratic Lyapunov function

$$L(Q(t)) = \sum_{k,n} (Q_n^{k}(t))^2.$$
The $T$-slot drift is given by:

$$
\Delta_T = \mathbb{E} \left[ L(Q(t + T)) - L(Q(t)) \bigg| Q(t) \right]
\leq T^2 K + \sum_{k, n} Q^k_n(t) (T\lambda^k_n + B) + \sum_{k, n} Q^k_n
$$

$$
- \mathbb{E} \left[ \sum_{\tau=0}^{T-1} \sum_{l, i=1}^{l_i=n} F^k_{l_i}(t + \tau) - \sum_{\tau=0}^{T-1} \sum_{l, |l|=n} F^k_{l}(t + \tau) \bigg| Q(t) \right] \quad \text{(3.25)}
$$

$$
= T^2 K + \sum_{k, n} Q^k_n(t) (T\lambda^k_n + B)
- \mathbb{E} \left[ \sum_{\tau=0}^{T-1} \sum_{k, l} F^k_{l}(t + \tau) \left( Q^k_{l}(t) - Q^k_{l|(t)}(t) \right) \bigg| Q(t) \right] \quad \text{(3.26)}
$$

The TBP policy minimizes the right hand side of inequality (3.25) at every timeslot. Hence it also minimizes the right hand side of inequality (3.26). So we can bound the drift by the rate variables chosen by the stationary policy.

$$
\Delta_T \leq T^2 K + \sum_{k, n} Q^k_n(t) (T\lambda^k_n + B) + \sum_{k, n} Q^k_n
$$

$$
- \mathbb{E} \left[ \sum_{\tau=0}^{T-1} \sum_{l, i=1}^{l_i=n} F^k_{l_i}(t + \tau) \bigg| Q(t) \right]
- \sum_{\tau=0}^{T-1} \sum_{l, |l|=n} F^k_{l}(t + \tau) \quad \text{(3.27)}
$$

$$
\leq T^2 K + \sum_{k, n} Q^k_n(t) (T\lambda^k_n + B) - \sum_{k, n} Q^k_n(T\lambda^k_n
+ T\epsilon - B) \quad \text{(3.28)}
$$

$$
\Delta_T^{TBP} \leq T^2 K - \sum_{k, n} Q^k_n(t)(T\epsilon - 2B) \quad \text{(3.29)}
$$

We use Lemma 10 to obtain (3.28). The drift is negative when $T > 2B/\epsilon$ and the queues are large. From [27] we know that the overlay queues are strongly stable.
Algorithm 7 Optimal Overlay Routing Policy (OORP)

At each time-step \( t \), overlay node \( i \) does the following:

**Optimal flow variables computation (used to obtain the subgradients):**
An overlay to overlay link \((i,j)\) computes the flow variables \( f_{ij}^{k*} \):

- Let \( k_{ij}^{opt} \in \arg \max_k q_{ij}^k - q_{ji}^k \), ties are broken arbitrarily. The weight of commodity \( k_{ij}^{opt} \) in this link is \( W_{ij}^{opt} = q_{ij}^{k_{ij}^{opt}} - q_{ji}^{k_{ij}^{opt}} \).

- For \( k = k_{ij}^{opt} \), \( f_{ij}^{k*} = \begin{cases} c_{ij} & \text{if } W_{ij}^{opt} > 0 \\ 0 & \text{otherwise} \end{cases} \)

- For all \( k \neq k_{ij}^{opt} \), \( f_{ij}^{k*} = 0 \).

Each overlay to underlay link \((i,j)\) computes the flow variable \( f_l^k \) for all \( l : (l_1,l_2) = (i,j) \):

- Let \((l_{opt}, k_{opt}) \in \arg \max_{l \in \{(i,j),k\}} q_{l_1}^k - \sum_{(a,b) \in \{l | a \in U\}} q_{l_2}^k - q_{l_2}^k \).

  Ties are broken arbitrarily. Let the weight of commodity \( k_{opt} \) in the tunnel be

  \[ W_l^{opt} = q_{l_{opt}}^{k_{opt}} - \sum_{(a,b) \in \{l | a \in U\}} q_{l_1}^{k_{opt}} - q_{l_2}^{k_{opt}} \]

  \[ \text{Ties are broken arbitrarily. Let the weight of commodity } k_{opt} \text{ in the tunnel be} \]

- For \((l,k) = (l_{opt}, k_{opt}) \), \( f_l^{k*} = \begin{cases} c_{ij} & \text{if } W_l^{opt} > 0 \\ 0 & \text{otherwise} \end{cases} \)

- For all \((l,k) : l \neq l_{opt} \) or \( k \neq k_{opt} \), \( f_l^{k*} = 0 \).

**Data transmission:**
Transmit \( f_{ij}^{k*} \) amount of commodity \( k \) traffic into each overlay to overlay link \((i,j)\) and \( f_l^{k*} \) amount of commodity \( k \) traffic into each tunnel \( l \).

**Dual variables update:**
Performed by an overlay node \( i \):

\[ q_i^k(t+1) = \left[ q(t) - \alpha(t) \left( \sum_j f_{ij}^{k*} + \sum_{l : l_1 = i} f_l^{k*} - \sum_{j : (j,i) \in E} f_{ji}^{k*} - \sum_{l : l_1 = i} f_l^{k*} - \lambda_i^k \right) \right]^+ \]  \hspace{1cm} (3.16)

Performed by an underlay node \( i \):

\[ q_{ij}(t+1) = \left[ q(t) - \alpha(t) \left( c_{ij} - \sum_{l : (i,j) \in l} \sum_k f_l^{k*} \right) \right]^+ \]  \hspace{1cm} (3.17)
Algorithm 8 Rate control algorithm for the utility maximization problem

At each time-step $t$:

1. Source node $i \in \mathcal{O}$ for commodity $k$ chooses the rate $\lambda_i^{k*}$ as follows:

   \[
   \lambda_i^{k*} = \arg \max_{0 \leq \lambda_i \leq M_i^k} \left( U_i^k(\lambda_i^k) - q_i^k \lambda_i^k \right).
   \]

   Here, $M_i^k$ is a finite upper bound on the rate that the source $i$ can receive.

2. All the overlay nodes use OORP for routing.
Chapter 4

Backlog Estimation and Control for Overlay Networks

Obtaining an estimate of the congestion in a communication network is an essential part of many network control algorithms. Different versions of TCP either use packet drops, delay or explicit notification to estimate congestion ([36], [13], [1]). Other algorithms such as the Backpressure routing algorithm [66] exchanges queue lengths between the neighbors, and overlay routing algorithm such as Overlay Backpressure [37] use the number of in-flight packets as a measure of congestion. Although these schemes work well in many scenarios, incorrect state estimation can lead to poor performance, as observed in the previous chapter. An accurate measure of congestion on a route in a network is given by the amount of backlog along the route. In this chapter, we develop a new method that learns a simplified model of the backlog in the network using historical data and use it to estimate the backlog in real-time. We then apply our scheme to an overlay network setting and show that when used in conjunction with the OORP control algorithm designed in the previous chapter, maximum throughput can be achieved.

A communication network can be viewed as a feedback control system where some or all of the nodes collect feedback which is used to design control actions in order to stabilize the queues or maximize network utility. It is well known that designing controllers for such feedback systems require the knowledge of the state of
the system [5]. However, in some settings this information is not readily available. In control theory, such a problem is resolved by adding an observer to the system that produces an estimate of the required state information, then the estimate is used to design control actions as shown in Figure 4-1. We will focus on designing such a state estimator for communication networks in which overlay nodes try to stabilize an underlay without being able to fully observe the state of the underlay.

![Figure 4-1: Feedback control system](image)

In many optimal control schemes for communication networks, e.g. the famous backpressure algorithm [66], it is assumed that the network provides the state information to the controller. In the specific case of backpressure and backpressure like algorithms, the necessary state information includes the local and the neighbor queue-lengths which are exchanged explicitly at each time-slot. However in many scenarios obtaining such explicit feedback might not be possible. This may be because some legacy devices are not capable of providing the state information or the data packets might travel through devices controlled by parties that are unwilling to share such information.

The Internet is an example of a system where the network doesn’t provide explicit state information. To overcome this, TCP algorithms such as Tahoe [36] infer the congestion from packet losses, i.e., a successful packet implies low congestion and a dropped packet implies high congestion. Other versions such as TCP-Vegas [13] use delay experienced by the packets as a measure of congestion along the path, and yet others use explicit congestion notification [1]. In an overlay network setting, the authors of [37] use the number of packets that have been sent along a route but have not made it to the destination, called the packets in flight (PiF), as an estimate of
the congestion along the route. This method makes a simple inference: the higher the number of packets in flight, the higher the congestion along the path.

It has been shown in [48] that the total backlog in each path is needed in order to maximize network utility. Similarly, in an overlay network setting we showed, in Chapter 3, that a throughput optimal control algorithm requires the total backlog along each available path, and that using a coarse estimation of backlog, such as delay or PiF, can lead to loss of throughput. This is because the delay information is received only when a packet leaves the network, and by that time the network state could have changed significantly. The PiF on the other hand does not take packets from other paths into account which can lead to large estimation errors. In this chapter we propose a new method that combines PiF and delay information to obtain accurate predictions of the backlog.

We build a model of the backlog along a path in the network as a function of PiF by using a nonparametric learning technique called Multiple Adaptive Regression Splines (MARS) [23]. The model is trained with the historical PiF and delay data. The MARS algorithm builds a piecewise linear model which is easy to implement and simple to understand. We then use this model to produce estimates of the backlog and show that when used in conjunction with the OORP control algorithm of Chapter 3, its performance is very close to that of a system with perfect state information.

Furthermore we propose a simplified (linear) version of this model as a new tool of performing network tomography. Tomography is the study of internal characteristic of the network using the data available at the end nodes. Because our approach only needs delay and PiF, which are available at the edges, it is a suitable tool for this purpose. We provide examples that demonstrate how our approach can be used to identify bottlenecks in the network and detect discriminatory behavior by the underlay control algorithms. These examples point to a potentially new direction of research in learning-based network tomography.

This chapter is organized as follows. We describe the model in Section II. In Section III, we formulate the backlog estimation problem and discuss alternative approaches. We describe the MARS algorithm and the process of building the backlog model.
model in Section IV and integrate it with the control algorithm of Chapter 3 in Section V. Finally in Section VI, we describe the application of our model to learning the structure of the network, then we conclude in section VII.

4.1 Network Model

We consider an overlay network where overlay nodes are connected using an existing legacy network (the underlay). The underlay provides a unique, acyclic path from one overlay node to another. When a packet enters the underlay it is routed along a path to its destination. Hence, the overlay nodes see the paths through the underlay as tunnels. Each tunnel connects two overlay nodes, and it is defined by a path that starts at an overlay node, goes through the underlay, and ends at another overlay node. At the overlay, packets can be sent directly to their destination using a direct tunnel, or over a multi-hop path through other overlay nodes. This allows the overlay network to achieve a higher throughput by using multiple paths. A cartoon of an overlay network is given in Figure 4-2.

![Figure 4-2: An overlay network with three overlay (white) nodes and three underlay (gray) nodes. The underlay uses fixed, single path routing algorithm and FIFO service discipline. The overlay nodes can send packets directly to their destination or forward them through other overlay nodes by using a dynamic routing algorithm.](image)

The network is modeled as a graph \((N, E)\) where \(N\) is the set of nodes and \(E\) is the set of edges. The nodes are of two types: overlay and underlay. The set of overlay nodes is represented by \(O\) and the set of underlay nodes is represented by \(U = N \setminus O\). The edges are capacitated, and the capacity of an edge from node \(i\) to node \(j\) is represented by \(c_{ij}\). The network has a set of commodities \(K\), where each commodity \(k \in K\) is defined by a source-destination pair in the overlay. The time is
slotted and represented by \( t \). We often remove the index for clarity if it doesn’t lead to ambiguity.

The underlay nodes maintain a queue for each outgoing edge. The length of the queue for outgoing edge \((i, j)\) at time \( t \) is represented by \( q_{ij}(t) \). Note that these queues can contain packets from different commodities and different tunnels. Unless otherwise stated, the packet forwarding in the underlay is done using the first in first out (FIFO) scheme. Moreover, the underlay network topology and the routes are assumed to be unknown.

The overlay nodes maintain per-commodity queues. The length of the queue in node \( i \) for commodity \( k \) at time \( t \) is represented by \( q^k_i(t) \). At the overlay nodes, the packet scheduling is done using a dynamic control algorithm, such as the Optimal Overlay Routing Policy (OORP) of the previous chapter. The control policy determines the commodity and the number of packets to be sent into each tunnel at each time-slot.

We will represent the set of all the tunnel by \( L \). A tunnel \( l \in L \) is a sequence of nodes \((l_1, l_2, ..., l_{|l|})\) where \(|l|\) is the length of the tunnel. As mentioned in the beginning of the section, \( l_1 \) and \( l_{|l|} \) are overlay nodes, and the rest are underlay nodes. Hence, for each tunnel \( l \), the queue-length \( q^k_{l_1} \) and \( q^k_{l_{|l|}} \) are known to the overlay network because they reside in the overlay nodes. However, \( q_{l_2,l_3}, ..., q_{l_{|l|}-1,l_{|l|}} \) are part of the underlay, hence they are not available to the overlay.

It was shown in [48] and the previous chapter that in order to control the network optimally, the total backlog in each tunnel is required. This quantity is equal to the sum of the queues-lengths in the tunnel. We will refer to this sum as tunnel-backlog, i.e.

\[
   b_l(t) = \sum_{(i,j) \in l} q_{ij}(t).
\]

The main goal of this chapter is to design an estimator \( \hat{b}_l(t) \) to estimate the tunnel-backlog \( b_l(t) \) for each tunnel \( l \) using only the measurements available at the overlay nodes.

In particular we will use two quantities that can be measured directly at the
overlay: packets in flight and delay. The packets in flight in tunnel $l$ at time $t$ is represented by $h_l(t)$. This quantity is available to the overlay immediately at each time-step because the overlay knows how many packets have been sent into the network and how many have been received. The delay experienced by a packet that enters the tunnel at time $t$ is represented by $d_l(t)$. Note that this quantity is available only when the packet exits the tunnel. That is, if a packet enters the tunnel at time $t - \tau$ and exits a tunnel at time $t$, then at time $t$ the overlay obtains $d_l(t - \tau) = \tau$. Also note that $\tau$ depends on the amount of backlog in the tunnel at the time the packet enters tunnel $l$.

### 4.2 Backlog Estimation

We describe two approaches to estimating the tunnel backlog with the information available at the overlay. The first approach uses delay measurements received from the packets. We will show that this approach, although intuitive, does not give a good estimate of the current backlog. However, it does give a good approximation of the tunnel backlog at a past time-slot which will prove valuable for the purpose of training our model.

The second approach models the tunnel backlog as a function of PiF and designs a minimum mean squared error (MMSE) estimator. As an example, we derive an expression for a MMSE estimator for Jackson networks and study a specific topology in order to gain insight into the estimation problem. We then apply this insight to develop estimation schemes that can be used in general networks in Section 4.3.

#### 4.2.1 Backlog and delay

A simple way to estimate the backlog in tunnel $l$ is to use the delay of the most recent packet that exits the tunnel, $e_l(t)$. This is not the same quantity as $d_l(t)$ because $e_l(t)$ is the delay experienced by the packet exiting tunnel $l$ at time $t$ whereas $d_l(t)$ is the delay of the packet entering the tunnel at time $t$. In general, if the bottleneck link has capacity $c$, $d_l(t) \approx \frac{1}{c}h_l(t)$. Unfortunately the quantity $d_l(t)$ is not available in
real time and can only be obtained once the packet exits the tunnel, i.e. we can only observe \( e_l(t) \) at time \( t \).

In Figure 4-3(a) we plot \( e_l(t) \) along with the actual backlog for the network of Figure 4-4. As can be seen, as the backlog increases, the estimation gets worse. The main reason behind this is that by the time the packet exits the tunnel, the queue-lengths have already changed, i.e. the delay estimate is delayed. We described this problem in Chapter 3, where it was shown that using delay instead of backlog in the control algorithm leads to loss in throughput.

Notice that in Figure 4-3(a) the delay is close to a time-shifted version of the
backlog. That is, \( b_t(t) \approx e_t(t + \tau) \) for some \( \tau \geq 0 \) that is proportional to the backlog. In Figure 4-3(b), we plot the backlog and \( d_1(t) \), the delay experienced by a packet that enters the tunnel at time \( t \). We can see that the delay \( d_1(t) \approx b_t(t) \).

Although the delay is not a good estimator of backlog in real-time, it gives a good estimate of the past backlog. This observation is essential to the development of the techniques presented in this chapter because we will need the actual tunnel-backlog in order to train our models. Since the underlay may not provide this backlog data, we will use the historic delay data as a substitute for the backlog data.

### 4.2.2 Backlog and packets in flight

Packets in flight (PiF) is another quantity that can be used to estimate the tunnel-backlog. To motivate this, consider the case where the tunnel \( l \) does not intersect with any other tunnel. In this situation the tunnel-backlog \( b_l(t) \) is exactly equal to the packets in flight \( h_l(t) \). However, when two or more tunnels intersect, each tunnel can have some packets from another tunnel along with its own packets, so \( b_l(t) \) can be larger than \( h_l(t) \).

Tunnel backlog \( b_l \) is the sum of the PiF in tunnel \( l \) and a fraction of the PiFs in other tunnels that are also in tunnel \( l \) (due to intersecting paths). So, building an estimator requires us to determine what fraction of packets in flight from other tunnels are also in tunnel \( l \). If this fraction was constant this process would give us a linear estimator. We unfortunately, however, will learn in the next section that this is not the case.

Let \( q_{ij}^l \) be the number of tunnel \( l \) packets in queue for link \((i,j)\), and let \( q_{ij}^L = (q_{ij}^l)_{(i,j) \in L} \) be the vector of the occupancy of the queues. The length of the queue is given by \( q_{ij} = \sum_{l \in L: (i,j) \in l} q_{ij}^l \). The packets in flight for a tunnel \( l \) is given by \( h_l = \sum_{(i,j) \in l} q_{ij}^l \), and the tunnel-backlog is given by \( b_l = \sum_{(i,j) \in l} q_{ij} \). Notice that by computing the joint distribution of \( q_{ij}^l \), it is possible compute the minimum mean squared error (MMSE) estimator for the tunnel-backlog given the PiF, which is given by

\[
\hat{b}_l(t) = E[b_l | h(t)] \tag{4.1}
\]
where $h(t)$ is the vector of PiFs in all the tunnels.

4.2.3 PiF based MMSE estimator for multi-class Jackson networks

In general calculating the steady state distribution for the queue-occupancy $q^1_{ij}$ in a network of queues is very difficult [8]. For the purpose of illustration, we will focus on the multi-class Jackson networks [8] where the calculation of the steady state distribution is possible, and derive an expression for the MMSE estimator based on PiF. Jackson networks assume that the service time of a packet is exponentially distributed and uncorrelated across different links. This breaks the correlation between the packet arrivals and allows a derivation of an explicit expression of the joint distribution of the queues occupancies. We note that Jackson networks have been used extensively to study and gain intuition in various network settings (see [8] for a nice discussion on the topic).

Let us model the underlay network as a multi-class Jackson network as described in [8, 7]. The packets from different tunnels are considered to belong to different classes. The model is similar to the one described in Section 4.1 except for the following differences:

1. This model is continuous time instead of discrete.
2. The traffic comes into the underlay according to the Poisson process, where $\lambda_l$ is the arrival rate for tunnel $l$.
3. The service time at the underlay queue $q_{ij}$ is exponentially distributed with mean service time $c_{ij}$. Also the service time for a packet is independent between different queues.

From these assumptions, we can see that the load offered by a tunnel $l$ on queue for link $(i, j)$ is given by $\rho^l_{ij} = \frac{\lambda_l}{c_{ij}}$, and the total load on the link is given by $\rho_{ij} = \sum_{l \in L : (i, j) \in l} \rho^l_{ij}$.

Using the results from [7], it is easy to derive the joint equilibrium distribution of the occupancy of each queue in the underlay network. The joint distribution is given
where $G$ is the normalizing constant given by

$$G = \prod_{(i,j) \in E; i \in U} (1 - \rho_{ij}).$$

Using equation (4.2) the MMSE estimator (4.1) for a given network can be obtained by calculating the conditional distribution of the sum of the random variables and taking the expectation.

**Example:** We model the underlay network in Figure 4-4 as a Jackson network and derive the MMSE estimator. This network has two tunnels that intersect at link (5,6). The service rates of the links are given by $c_{35}$, $c_{45}$ and $c_{56}$ and the arrival rates into tunnels 1 and 2 are $\lambda_1$ and $\lambda_2$ respectively.

![Figure 4-4: A 3-link underlay network. The underlay nodes are shown as gray nodes. The dotted lines show the two tunnels in the network.](image)

For notational simplicity let $a = q_{35}^1$, $b = q_{45}^2$, $c = q_{56}^1$ and $d = q_{56}^2$. The tunnel-backlogs and the packets in flight for the two tunnels is given by $b_1 = a + c + d$, $b_2 = b + c + d$, and $h_1 = a + c$ and $h_2 = b + d$.

From equation (4.2) the joint stationary distribution of the queue occupancy is given by:

$$P(a, b, c, d) = (1 - \rho_{35})(1 - \rho_{45})(1 - \rho_{56})\frac{(c + d)!}{c!d!}(\rho_{35})^a(\rho_{45})^b(\rho_{56})^c(\rho_{56})^d$$
and the MMSE estimator for tunnel 1 is given by:

\[ \hat{b}_2 = f(h_1, h_2) = E[b + c + d|a + c = h_1, b + d = h_2]. \]

We also know that:

\[ E[a + b + c + d|a + c = h_1, b + d = h_2] = h_1 + h_2. \]

Using linearity of expectations:

\[ \hat{b}_2 = h_1 + h_2 - E[a|a + c = h_1, b + d = h_2] \]

\[ = h_1 + h_2 - \sum_{a=0}^{h_1} a P[a|a + c = h_1, b + d = h_2]. \]

We can calculate the conditional distribution of \( a \) using the joint distribution given above:

\[ P[a|a + c = h_1, b + d = h_2] \]

\[ = \frac{(\rho_{35})^a (\rho_{56})^{h_1-a} \sum_{k=0}^{h_2} \frac{(h_1-a+k)!}{(h_1-a)k!} \left( \frac{\rho_{36}^2}{\rho_{35}} \right)^k}{(\rho_{35})^{h_1} S(h_1, h_2)}, \text{ if } 0 \leq a \leq h_1, \]

where, \( S(h_1, h_2) = \sum_{k_1=0}^{h_1} \sum_{k_2=0}^{h_2} \frac{(k_1+k_2)!}{k_1 k_2!} \left( \frac{\rho_{36}^1}{\rho_{35}} \right)^{k_1} \left( \frac{\rho_{36}^2}{\rho_{35}} \right)^{k_2}. \)

The estimator for \( \rho_{35} = \rho_{45} = \rho_{56}^1 = \rho_{56}^2 = 0.45 \) is plotted in Figure 4-5(a). Notice that the shape of this estimator is nearly linear, and it appears that it can be well approximated by a piecewise linear function with a few pieces. The error of the estimator, given by the conditional variance, is plotted in Figure 4-5(b). It can be seen that for most of the values of \( h_1 \) and \( h_2 \) the error is very small. Large error occurs when there are very few packets in tunnel 2 and many packets in tunnel 1 which are low probability events.
Figure 4-5: MMSE estimator for the underlay network shown in Figure 4-4 with Jackson network assumptions.

4.3 Linear Models from Data

Obtaining the MMSE estimator as a function of the packets in flights is quiet complex even for a simple Jackson network. Moreover, in order to develop such an estimator several network parameters including the underlay topology and the paths need to be known, which is not possible with an unknown underlay. Hence, we take a different approach in this section and obtain the estimator from data. Motivated by the estimator of the previous section, which seems linear for the most part, we first develop a linear model of tunnel-backlog. However because of the existing nonlinearities, even the optimal linear estimator can result in large errors. Hence, we also propose fitting piecewise linear models to the data by using a technique called MARS. A
cartoon showing how a piecewise linear model can better fit the data by incorporating nonlinearities is given in Figure 4-6.

![Diagram showing linear and piecewise linear models](image)

Figure 4-6: Cartoon of a linear and a piece-wise linear model fit to data.

### 4.3.1 Linear model using least squares regression

We develop a simple linear model of the form

$$ \hat{b}_t = \alpha_l^T h, $$

(4.3)

were $h$ is the vector of the packets in flight in all the tunnels, $\alpha_l$ is a vector of model parameters, and $(.)^T$ represents the transpose of the vector.

Let $H$ be the matrix of historical packets in flights and $B_l$ be the vector of historical backlogs in tunnel $l$ (either actual or estimated through delay). Row $i$ of $H$ is the packets in flights in all the tunnels at a certain time-slot $t$, and the $i$th entry in $B_l$ is the backlog in tunnel $l$ at the same time-slot. It is well known that the vector of parameters $\alpha_l$ that fits the linear model and minimizes the Euclidian norm of the error for the samples is given by

$$ \alpha_l = (H^T H)^{-1} H^T B_l \ [10]. $$

We note that the least squares method can be seen as an approximation of the MMSE estimator under some simplifying assumptions such as having independent and identi-
cally distributed samples of \((h, b_i)\) [10]. Also note that this model is not a special case of the piece-wise linear model that is generated by the MARS algorithm described below because MARS is a greedy algorithm and it is not guaranteed to minimize the error.

### 4.3.2 Piecewise linear model using the MARS algorithm

Multiple Adaptive Regression Splines (MARS) described in [23] is a popular algorithm to fit a piecewise linear function to data. It is an iterative algorithm that is fast and known to work well in practice [50]. MARS builds the piecewise linear model by taking a linear combination of “hinge” (or “hockey-stick”) functions of the form \(\max(h_i - x_i, 0)\) or \(\max(x_i - h_i, 0)\). Here \(h_i\) is an independent variable such as the packets in flight, and \(x_i\) is a constant that represents the location where the two lines (pieces) connect, known as the “knot”. The set of possible basis function \(B\) comprises of all the hinge functions with a knot at each value of each input variable. With a single variable \(h_1\) and \(p\) samples (of \(h_1\)) given by \(x_1, ..., x_p\), the set of basis functions is given by \(B = \{\max(h_1 - x_i, 0), \max(x_i - h_1, 0)\}_{i \in \{1, 2, ..., p\}}\).

The model produced by MARS has the form

\[
\hat{b}_i = f(h) = \sum_{i=1}^{K} (\alpha_i)_i B_i,
\]

where \(B_i \in B\) is the \(i\)th basis function, \(\alpha_i\) is the vector of weights which determines the slope of the lines and \((\alpha_i)_i\) is its \(i\)th element, and \(K\) is the number of basis functions in the model. This model is easy to use and much easier to interpret compared to those produced by other machine learning techniques such as artificial neural networks.

The MARS algorithm is a greedy iterative algorithm that operates in two phases. In the first phase, the algorithm adds one basis function per iteration until the maximum number of basis functions allowed, a parameter to the algorithm, is reached. At each iteration, all the basis functions are tested one at a time by choosing the slope using the least squares regression method. Then the basis function that gives the largest decrease in error is added to the model. This usually causes the model
to over-fit. Thus, in the second phase MARS removes one basis function at a time. Again, the removal is done greedily such that the basis function that adds the least error is removed in each iteration. The removal of the basis function continues until a generalized cross validation (GCV) condition is satisfied. This GCV test weighs having too many terms in the model against the residual error increase in order to curb over-fitting ([23, 35]).

4.3.3 Data preparation

In order to generate the models using either of the algorithms described above, we need a matrix of historical PiFs and a vector of the corresponding tunnel backlogs. The PiF data can be measured by the overlay in each time-slot. However, the historical tunnel-backlog data may not be available at the overlay but can be approximated with the delay data, as discussed in Section 3-B. Before sending a packet into the tunnel, it is stamped with the current time \( t \). The delay experienced in the tunnel can be calculated when the packet exits the underlay. Note that all of this need not happen in real-time. As observed in Section 3-B the tunnel backlog at time \( t \), i.e. the time when the packet entered the tunnel, can be approximated by the delay experienced by the packet in the tunnel. Before feeding the delay data into the regression algorithms, some small adjustments need to be made to the data:

1. Multiple packets can be sent at the same time-slot \( t \) into a tunnel \( l \). We use delay of the first first packet to exit the tunnel as the tunnel-backlog at time \( t \) and disregard the other packets that entered tunnel \( l \) at time \( t \).

2. Packets might not be sent at each time-slot so the backlog at some time-slots can be missing. This can be dealt with by performing a linear interpolation. Suppose a packet sent on time \( t_1 \) incurs a delay of \( d_1 \), no packet is sent on time \( t_2 \), and a packet sent on time \( t_3 \) incurs a delay of time \( d_3 \). Then, delay at \( t_2 \) can be approximated by

\[
d_2 = \frac{d_3 - d_1}{t_3 - t_1} (t_2 - t_1) + d_1.
\]
This works well in practice because packets sent at close time-slots experience similar delay.

3. In systems with store and forward queues, a packet takes one time-slot to traverse a link even when the queue is empty. Hence, we subtract the number of hops from the delay data of each tunnel, where the number of hops is approximated by the minimum delay in the tunnel.

4. If the bottleneck link in the underlay network has non-unit capacity, then the delay must be multiplied by the capacity to obtain the backlog. In this chapter this is not a concern because the main application for the backlog estimation is control, and the controls algorithm achieve stability as long as the estimation is proportional to the actual tunnel-backlog ([48, 46]). However, if a more precise estimate is required, it is possible to use methods such as the ones in [16] or [65] to calculate the bottleneck capacity of the tunnels.

4.3.4 Simulation result

We present the result of a simulation of our approach on the simple underlay topology of Figure 4-4. The arrivals into the underlay are taken to be Poisson distributed with rates \( \lambda_1, \lambda_2 = \rho[0.5, 0.5] \), where \( \rho \) is the load parameter. The service times are constant and all the underlay links have unit capacity. We used an open-source Matlab implementation of the MARS algorithm called ARESLab from [35] to create our piecewise linear models. The training data was collected by running the network at \( \rho = 0.9 \) and recording the delays.

Figure 4-7(a) shows the mean squared error as a function of the number of basis functions. The model with 1 basis function is constant in at least one of the dimensions, so it results in a high error. We can see that with only a few basis functions, the decrease in the error is significant, and the improvement is minimal when the number of basis functions grows larger. Figure 4-7(b) shows the estimator for the backlog in the second tunnel as a function of the packets in flight. We can see that a large portion of the plot consists of a single “piece”, hence we can expect the linear
model to give good estimates as well.

Figure 4-7: The piecewise linear model of the backlog as a function of PiF for the underlay network shown in Figure 4-4.

The performance of the linear and the piecewise linear models is compared Figure 4-8. We define the squared coefficient of variation as

\[ SCV = \frac{E[(b_t - b_i)^2]}{E[b_t]^2}. \]

This quantity is designed to compare the performance of the estimator with respect to the average queue sizes in the tunnel. In the figure we can see that the SCV decreases as the load increases. This shows that the performance of the estimators gets better relative to the backlog as the load increases. The relatively high SCV in the low load regime is because the backlog is small on average, e.g. when the backlog \( b_t = 1 \) and
Figure 4-8: Comparison of the linear model obtained by least squares regression and the piecewise linear model obtained by MARS. The bar chart shows the squared coefficient of variation for the mean squared error (MSE). The numbers on top of the bars show the improvement in the MSE obtained by the piecewise linear model over the linear model.

the estimation is \( \hat{\lambda} = 0 \), the relative error is 100%. We can also see that the piecewise linear model consistently outperforms the linear model. The percent improvement is given on the top of the bars. The relative performance of the two schemes is close near the load where the models were trained at, i.e. \( \rho = 0.9 \).

4.4 Control for Overlay Networks

We use the methodology from the previous section to design an estimator to be used in the control algorithm for overlay networks. We then compare the performance of this estimator with various other methods via simulations.

4.4.1 The OORP algorithm

The OORP algorithm from Chapter 3 aims to stabilize all the queues in an overlay network whenever the traffic vector \( \lambda \) is in the stability region. The stability region consists of the set of all the arrival rate vectors that can be stabilized in the network by some algorithm. At a high level, the OORP algorithm works by moving packets from an overlay node with higher backlog to nodes with lower backlog. As the underlay
does not allow for any control, all the decisions have to be made at the overlay. A simplified description of the algorithm is given in Algorithm 9. Any link from an overlay node to another overlay node, that does not go through the underlay, is modeled as a tunnel with zero tunnel-backlog.

Algorithm 9 Optimal Overlay Routing Policy (OORP)

At each time-step $t$, overlay node $i$ does the following for each tunnel $l$ that starts at that node (i.e. $l_1 = i$):

1. Determine the commodity $k^*$ with the highest weight $w^*$:

   $$ k^* = \arg \max_k q_{l_i}^k - b_l(t) - q_{l_1}^k, \quad \text{and} $$

   $$ w^* = q_{l_i}^{k^*} - b_l(t) - q_{l_1}^{k^*}. $$

   Ties are broken arbitrarily.

2. If $w^* > 0$ then send $c_{l_1l_2}$ amount of traffic of commodity $k^*$ into tunnel $l$. Otherwise, do not sent any traffic into the tunnel.

3. Remove packets at the destination.

We can see that the estimation of the tunnel backlog $b_l(t)$ is necessary to calculate the weights (see step 1), and a bad estimation can cause the network to send wrong commodities into a tunnel or waste capacity by staying idle. Several strategies to estimate $b_l(t)$ have been proposed Chapter 3:

- **Delay-probe approach**: Use the delay of the last exiting packet $e_i(t)$ as the estimate of the backlog. Send dummy packets if no packets are sent for $T$ time-slots. This method results in loss of throughput.

- **Priority-probe approach**: Send special probe packets every $T$ time-slots. The underlay queues give these packets high priority and stamps the queue-lengths into these packets. Essentially, this method obtains the true tunnel-backlog at every $T$ time-slots but needs support from the underlay network.

- **Known tunnel backlog**: The underlay send the tunnel backlog to the overlay at each time-slot. Again, this method also needs support from the underlay.
In the next section we will show that our method of backlog estimation performs better than Priority-probe approach with low time interval between the probe packets. Moreover, our method does not require any help from the underlay.

### 4.4.2 Simulation result

We compare the performance of OORP under various estimation schemes in the network shown in Figure 4-9. We use the same topology that was used in Chapter 3 for the comparisons. This topology has 4 overlay nodes and 10 underlay nodes, and all the links are bidirectional. The links between two underlay nodes have capacity 1 in each direction, and the links between an overlay and an underlay node have capacity 2 in each direction. This overlay network has a total of 18 tunnels, and is particularly difficult to stabilize because many of the tunnels interfere and sending traffic into wrong tunnels can result in instability.

![Figure 4-9: Topology for the OORP simulation. The network consists of 4 overlay (white) nodes, 10 underlay (gray) nodes, and 18 tunnels. The underlay network uses shortest path routing. The dotted lines show some of the tunnels available to the overlay node 1.](image)

The network has two commodities. Commodity 1 goes from node 1 to node 3, and commodity 2 goes from node 2 to node 4. The maximum arrival rates supported by the network for these commodities is $\left[\lambda_1^{\max}, \lambda_2^{\max}\right] = [2, 2]$. We run the simulation under different arrival rates of $\rho A^{\max}$, where $\rho < 1$ is the load. We can see that in
order to achieve a throughput close to $[2, 2]$ commodity 2 has to be routed through overlay node 1 and commodity 1 has to be routed through overlay node 2.

For the purpose of training our learning models, we obtained the delay and packets in flight data by using the Delay-probe method for $10^5$ time-slots at 80% load and used the methodology from the previous section to create a linear and a piecewise linear model with the data. Then we used the model to estimate the tunnel-backlog $b_t(t)$ in the OORP algorithm.

Figure 4-10: Performance comparison of OORP under various tunnel-backlog estimation schemes. The methods developed in this chapter (solid lines) perform very well without any support from the underlay network.

The results of the simulation are given in Figure 4-10. We plot the average queue-lengths of all the queues in the network (both underlay and overlay) at various load levels. Average backlog is a good measure of performance as it can show instability in the network as well as the average delay experienced by packets since average delay is proportional to average backlog. As expected, the Delay-probe method is sub-optimal because the average backlog approaches infinity around 87% load, and the case where the tunnel backlogs are known achieves the best performance.

The estimation methods developed in this chapter achieve stability and good performance compared to the other methods. The linear model trained with the delay data has very good performance on loads as high as 95%. The piecewise linear model generated using the MARS algorithm performs even better. This method
achieves the performance that beats the Priority probe method with $T = 10$ at most load levels and performs comparably at even 99% load. Moreover, our method does not require probe packets.

4.5 Learning-based network tomography

In this section we demonstrate how our learning-based estimation approach can be used to solve certain network tomography problems. In particular, we will use a slightly modified version of the linear model for identifying some internal characteristics of the underlay network. Network tomography has been an active area of research for the past 20 years. The first paper on the topic, [69], developed a method to measure source-destination traffic intensities from the measurements of the traffic intensities at each link. Other papers such as [39] and [60] detect shared bottlenecks by observing packet inter-arrival times and packet delays respectively. There are also several papers that focus on discovering the topology of the underlay network using the delay experienced by the packets in the network, e.g. [68, 2]. Next we briefly describe two important tomography problems that can be addressed using our learning and estimation approach. These examples are by no means a complete solution, but point to a new research direction in learning-based network tomography.

We use a linear model similar to the one developed in Section 4.3.1, with the difference that we will only model the effect of the PiF for tunnels other than $l$ on the backlog of tunnel $l$. So the model becomes

$$\hat{b}_l - h_l = \beta_l^T h_{\setminus l},$$

where $h_{\setminus l}$ is a vector of PiF of all the tunnels except $l$, and $\beta_l^T$ is the transposed vector of model parameters. We use the same data preparation techniques and least squares linear regression as in Section 4.3 to obtain the optimal parameters $\beta_l^T$. Then
we form a \(|L| \times |L|\) matrix of the parameters given by

\[
A = \begin{bmatrix}
1 & (\beta_1)_1 & (\beta_1)_2 & \cdots & (\beta_1)_{|L|-1} \\
(\beta_2)_1 & 1 & (\beta_2)_1 & \cdots & (\beta_2)_{|L|-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(\beta_{|L|})_1 & (\beta_{|L|})_2 & \cdots & (\beta_{|L|})_{|L|-1} & 1
\end{bmatrix}
\]

where \(|L|\) is the number of tunnels in the network. The \((i, j)\) entry of \(A\) can be interpreted as an approximation of the fraction of tunnel \(j\) packets that are also in tunnel \(i\). We will use this matrix to understand the characteristics of the underlay network.

This approach has several properties that can make it a good choice for network tomography. First, the measurements are passive: there is no need to control the traffic pattern for the purpose of tomography or inject special packets into the underlay. Second, the calculations involved in obtaining the model are simple, and the model is very intuitive to understand and reason about. Finally, using PiF in addition to delay gives more information about the network than using just the delay, which is commonly done by other methods in the literature.

### 4.5.1 Shared bottlenecks

Consider the underlay networks shown in Figure 4-11. This figure shows three different ways that three tunnels can share bottleneck links. We want to distinguish between these three cases by using the measurements outside of the network. Note that most other methods such as [60, 68] only deal with two flows at a time, hence they will have difficulty distinguishing the networks in Figure 4-11(a) and Figure 4-11(c).

We simulated the network under Poisson traffic at a load of 95% and collected the delay and packets in flight data. Then we used the methodology described above to
(a) Case 1: All tunnels share a bottleneck link. (b) Case 2: Only two tunnels share a bottleneck link.

(c) Case 3: Each link is shared by two tunnels only.

Figure 4-11: Different ways three tunnels can share bottlenecks.

form the $A$ matrices. The matrices for the three cases are given by

$$
A_1 = \begin{bmatrix}
1 & 1 & 0.99 \\
1 & 1 & 0.99 \\
1 & 1 & 1
\end{bmatrix},
A_2 = \begin{bmatrix}
1 & 0.02 & 0.03 \\
0.01 & 1 & 1 \\
0.03 & 1 & 1
\end{bmatrix},
$$

$$
A_3 = \begin{bmatrix}
1 & 0.54 & 0.49 \\
0.40 & 1 & 0.51 \\
0.52 & 0.47 & 1
\end{bmatrix}.
$$

We can see that the matrices have distinct structures that correspond to the way the tunnels intersect with each other. All the entries of $A_1$ are close to 1, which indicates that the backlog in each tunnel is composed of packets in flight from all the tunnels, hence $A_1$ must correspond to the network in Figure 4-11(a). The matrix $A_2$
has a block diagonal structure. The backlog in the first tunnel is not affected by the PiF in tunnels 2 and 3, and PiF in tunnels 2 and 3 affect each other’s backlog. Hence, this matrix corresponds to the network in Figure 4-11(b). Finally we can see from $A_3$ that each tunnel in its corresponding network has about half the PiF belonging to other tunnels. This behavior corresponds to the network in Figure 4-11(c) because each tunnel in this network intersects with one other tunnel at each link it traverses. For example, tunnel 1 traverses two underlay links (1,2) and (2,3). Hence it includes some packets from tunnel 2 that are queued at link (2,3) and some packets from tunnel 3 that are queued at link (3,1).

4.5.2 Detecting discrimination between flows

Consider an underlay network with two tunnels that share a bottleneck link such as the one shown in Figure 4-4. Our goal here is to determine whether the network discriminates against one flow in favor of another. This is a timely question because of its relevance to network neutrality, which is the highly debated principle that all traffic should be treated equally by the network regardless of their origin. Note that the tomography methods that use only the packet delays will have a difficult time distinguishing the two cases because in each case, whether or not discrimination exists, the delays experienced by the packets in the two tunnels are correlated.

When we obtain the $A$ matrix for the network that does not have priority, we get a matrix of ones, similar to the matrix $A_1$. However, when the network has priority we obtain the following asymmetric matrix:

$$A_{\text{Priority}} = \begin{bmatrix} 1 & 0.04 \\ 6.7 & 1 \end{bmatrix}.$$  

Note that when there is priority, the delay no longer corresponds to the backlog in the tunnel, so we have to interpret the $A$ matrix simply as a relationship between PiF and delay. From the matrix $A_{\text{Priority}}$, we can see that this network prioritizes the first tunnel over the second. The packets in flights in the second tunnel does not
affect the delay of the packets in the first tunnel. However, the PiF in first tunnel affects the delay of the packets in the second tunnel.

The above examples, while by no means a complete solution, indicate that learning techniques such as the ones developed in this paper can play an important role in network tomography.

4.6 Conclusion

We developed a new technique to estimate the backlog in the tunnels of an overlay network, using a learning-based methodology. Our approach uses linear and piecewise linear models of the tunnel-backlog and learns model parameters from historical data. We used these models to optimally control an overlay network and showed that the performance is close to the case where the actual backlogs are known in real-time. Finally we showed that our model can be used for the purpose of network tomography. Developing learning-based network tomography schemes is a promising direction for future research.
Chapter 5

Topology Discovery Using Path Interference

Knowing the topology of the underlying network can provide several advantages to the communicating hosts. For example, the topology can be used to improve the throughput and robustness of the network [3, 38], and it can be a necessary part of identifying bottlenecks and critical links in the network [42]. It can also be used to monitor the network or to simply get a picture of the underlying system. However, often the owners of the network keep the topology information hidden due to privacy and security concerns [29]. This has led to a significant amount of research on topology discovery. We develop a new method that can be used to identify general network topologies. This method only requires the interference pattern of the paths in the network which can be inferred from the data available at the end nodes.

Prior work on topology discovery can be divided into two main categories: algorithms that require cooperation from the internal nodes and the algorithms that do not. Many algorithms for topology discovery, usually designed for the purpose of mapping the Internet, use ICMP commands like traceroute [64, 21, 29]. These methods require some level of cooperation from the network providers. The other methods, that fall under the category of network tomography [69, 17], use data that can be measured directly at the end nodes. Our method falls under this category as we do not seek any information from the internal nodes.
In the network tomography literature significant attention has been given to the discovery of tree networks. Papers such as [18, 22, 54] use probing mechanisms to infer single source multiple destination trees. There is also some work on combining these single source trees to form a multi-source multi-destination network [19, 61]. In [41], the authors provide a method for identifying minimal trees with multiple sources and multiple destinations by using distance measurements.

In [2], the authors develop an algorithm called RGD1 that attempts to discover a general network topology. It uses a set for four nodes that share a link, called quatrets and uses them to build an approximation of the entrie network. The discovery of the quatret and placement of the nodes in the topology requires the shortest path distance between the nodes, which is inferred using packet delay. RGD1 algorithm is very close to our algorithm in terms the objective, hence we will compare its performance against ours via simulation.

In order to obtain the interference pattern, we provide a simple method based on linear regression. This method uses the number of in-flight packets in the paths and the delay experienced by the packets to determine whether a given pair of paths interfere with each other. Using the resulting interference information, we formulate the topology inference problem as an integer program. We develop polynomial time algorithms to solve it optimally for networks with special topologies, namely tree or ring topology. Both of these algorithms obtain the minimal version of the network, even when the original network is not minimal. We also develop a heuristic that attempts to recover any general topology in polynomial time.

The main contributions of this chapter can be summarized as follows:

- We use the interference pattern of the paths to formulate an integer linear program (ILP) that obtains the network that has the fewest number of links and supports the given interferences. The solution provides a new method to discover a general network topology.

- We design two polynomial time algorithms to recover tree and ring networks and show that if the network is in fact a tree or a ring, the algorithms solve the
ILP optimally.

- Building upon the tree and the ring algorithms, we develop a polynomial time heuristic to identify general networks. Using simulations we show that this method outperforms the RGD1 algorithm of [2].

5.1 Model

5.1.1 Network Model

We model the network as a graph $G = (N, E)$, where $N$ is the set of nodes and $E$ is the set of edges. We assume that all the links in the network are bidirectional and have unit capacity. Each bidirectional link $\{i, j\}$ is composed of two directed links $(i, j)$ and $(j, i)$. The network has two types of nodes: the overlay nodes, which represent hosts and can be controlled, and the underlay nodes, which represent routers that are uncontrollable and do not provide any direct feedback. We represent the set of overlay nodes by $\mathcal{O}$ and the set of underlay nodes by $U$, and $N = \mathcal{O} \cup U$. We further assume that each overlay node is connected to only one underlay node. Other than this, we do not have any knowledge of the structure of the network. The main goal of this chapter is to recover the graph $G$ from data measured at the overlay nodes.

All the overlay nodes are connected to each other by tunnels, which are paths that go through the underlay nodes. A tunnel $l = (l_1, l_2, ..., l_{|l|})$ consists of overlay nodes $l_1$ and $l_{|l|}$ and the rest of the nodes are underlay. Since, $l_1$ and $l_{|l|}$ are connected to only one underlay nodes, we will often refer to node $l_2$ as the parent of node $l_1$, $p(l_1)$, and node $l_{|l|-1}$ as the parent of node $l_{|l|}$, $p(l_{|l|})$. There are a total of $L = |\mathcal{O}| \times (|\mathcal{O}| - 1)$ tunnels in the network.

We also assume that each node $i \in N$ maintains a queue for each of it outgoing link $(i, j) \in E$. Packets from all the tunnels that uses the link $(i, j)$ gets enqueued in this queue when they reach node $i$ and are served on a first come first serve basis.
5.1.2 The Interference Matrix $\mathcal{F}$

Our algorithm for recovering the graph $G$ is based on whether or not any two tunnels between the overlay nodes intersect with each other. In order to identify this we propose a simple method based on linear regression. We note that depending on the measurements available, other methods such as the ones from [58, 2] can also be used to derive this information.

Let $d_l(t)$ represent the delay experienced by a packet that enters tunnel $l$ at time $t$. Tunnels in the network can intersect with each other, hence, the path traversed by a tunnel can have packets belonging to itself and packets from other tunnels. Let $h_l(t)$ represent the number of packets that belong to tunnel $l$ that are still in the tunnel at time $t$. We will refer to these packets as packets in flight of tunnel $l$. The delay experienced by a packet entering tunnel $l$ at time $t$ is affected by the number of packets in that tunnel and other tunnels that intersect with it. Considering only a pair of tunnels $k$ and $l$, we can model the relationship between the packets in flight and delay as a linear function:

$$d_l(t) = h_l(t) + \alpha_{kl} h_k(t) + \eta_l.$$ 

Here $\alpha_{kl}$ represents the fraction of packets of tunnel $k$ that are in the path traversed by tunnel $l$ and $\eta_l$ is random perturbation (noise).

By injecting randomly generated traffic into each pair of tunnels and measuring the packet delay and packets in flight, it is possible to determine if two tunnels intersect. In particular, using linear regression it is possible to calculate the optimal parameters $\alpha_{kl}$ that minimizes the noise for each pair of tunnel $(k,l)$. When tunnels $l$ and $k$ do not intersect, the number of packets in tunnel $k$ does not affect the delay of packets entering tunnel $l$, hence, $\alpha_{kl} \approx 0$. Otherwise, $\alpha_{kl}$ is closer to 1. We use these $\alpha_{kl}$ values to create the $L \times L$ binary interference matrix $\mathcal{F}$. If $\alpha_{kl} \approx 0$ then $\mathcal{F}_{kl} = 0$, and $\mathcal{F}_{kl} = 1$ otherwise. Moreover, $\mathcal{F}$ is symmetric, implying $\mathcal{F}_{kl} = \mathcal{F}_{lk}$.

We will use the graph representation of $\mathcal{F}$ in some of our results. We refer to such a graph as the interference graph of the network, $G_{\mathcal{F}}(N_{\mathcal{F}}, E_{\mathcal{F}})$. This graph is simply
the graph formed by using $\mathcal{F}$ as an adjacency matrix, where $N_\mathcal{F}$ consists of tunnels and an edge exists between tunnels that interfere with each other. An example of an interference matrix and its corresponding graph is given in Figure 5-1.

5.1.3 Minimal topology

There exist many networks that produce the same interference matrix $\mathcal{F}$, hence, these networks are indistinguishable by our method. For example, each tunnel in the two networks shown in Figure 5-1 face the same interference. E.g. the tunnel $(1, ..., 2)$ only interferes with tunnel $(1, ..., 3)$ in both the networks. Hence, they produce the same $\mathcal{F}$ matrix. We are interested the smallest network, in terms of the number of links, that produces the given $\mathcal{F}$ matrix. We will call such a topology the minimal network topology.

![Diagram of two networks](image)

(c) The interference matrix $\mathcal{F}$. (d) The interference graph $G_\mathcal{F}$. Each node $(i,j)$ represents a tunnel $(i, ..., j)$.

Figure 5-1: Two topologies in Figures 5-1(a) and 5-1(b) produce the same $\mathcal{F}$ matrix. The white nodes are overlay and gray nodes are underlay, and the network uses the shortest path routing.

A necessary condition for a network to be minimal was identified in [2]. Specifically, all underlay nodes must have at least three neighbors. If an underlay has only
one neighbor, we can simply remove it to obtain a smaller network that is indistinguishable from the original network by using only the measurements available at the overlay. If an underlay node has two neighbors, we can connect its two neighbors and remove the node in order to obtain a smaller network with the same properties. We note that this condition is not sufficient for minimality in general. E.g. in Figure 5-1(a), all the underlay nodes have 3 neighbors but the topology is not minimal. We will provide a sufficient condition for minimality, and show that the necessary condition from [2] is also sufficient for specific topologies, namely trees and rings.

In this chapter we assume that the \( F \) matrix for a network \( G(N, E) \) is given (i.e. obtained via measurements, as described earlier) and focus on obtaining the minimal network \( \hat{G}^*(\hat{N}^*, \hat{E}^*) \) that supports this interference pattern.

### 5.2 Integer Programming Formulation

We formulate the problem of finding the minimal network that supports the given path interference pattern as an integer linear problem (ILP). Although a solution for this ILP is computationally intractable for large networks, studying this formulation will provide us with useful insights into the problem. Also, when the network is small, we are able to solve it optimally.

#### 5.2.1 Integer program

Let us consider a network with \(|\hat{N}|\) nodes. Nodes \(1, \ldots, |\mathcal{O}|\) are overlay nodes and nodes \(|\mathcal{O}| + 1, \ldots, |\hat{N}|\) are underlay nodes. Note that the set \( \mathcal{O} \) is known a priori.

Let \( x_{ij}^l \in \{0, 1\} \) represent whether link \((i, j)\) is used by tunnel \(l\), for \(1 \leq i, j \leq |\hat{N}|\), \(1 \leq l \leq L\), and \( x_{ii}^l = 0 \forall l \). For notational simplicity, we define another variable \( x_{ij} \) which represents whether the link \(\{i, j\}\) is used by any tunnel in either direction. Hence,

\[
x_{ij} = \lor_i x_{ij}^l \lor x_{ji}^l, \quad \forall i, j
\]

(5.1)

Here "\(\lor\)" is a logical OR operator. Note that such logical constraints can easily
be transformed into a set of linear (integer) constraints [11]. The objective function

\[ \text{minimize } \sum_{ij} x_{ij}. \]

Our network model assumes that each overlay node is connected to only one underlay link. This can be enforced by using the following constraint:

\[ \sum_j x_{ij} = 1, \quad i = 1, \ldots, |O|. \] (5.2)

Again, to simplify the notation we define two new variables. Let \( s(l, j) \in \{0, 1\} \) represent whether tunnel \( l \) begins at node \( j \), and let \( d(l, j) \) represent whether tunnel \( l \) ends at node \( j \). These values are known a priori, so we can replace these variables with their respective values while formulating a specific problem. Now we can write the next set of constraints which are essentially the flow conservation constraints. These constraints guarantee that each tunnel has a set of connected links in the network, starting and ending at its respective overlay nodes.

\[ \sum_i x_{ij}^l + s(l, j) = \sum_i x_{ji}^l + d(l, j), \quad j = 1, \ldots, |\hat{N}|, \]

\[ l = 1, \ldots, L \] (5.3)

We can see that the flow conservation constraints above allows loops to be formed in the network. Unlike max-flow type problems where loops can be removed in the post processing without harming the feasibility, removing them in our case can change the interference pattern of the tunnels. Hence we need to add constraints to avoid formation of loops.

Similar problems arise in the ILP formulation of the Travelling Salesman Problem (TSP). We use the technique originally proposed by Miller-Tucker-Zemlink in [49] to resolve this issue in TSP and add the following constraints:

\[ u_i^l - u_j^l + |\hat{N}| x_{ij}^l \leq |\hat{N}| - 1, \quad \forall i \neq j, l = 1, \ldots, L. \] (5.4)
Here, the variables $u_i^l \geq 0$ is used to assign an order to each node $i$ in each tunnel $l$. If $x_{ij}^l = 1$, then $u_j^l \geq u_i^l + 1$, so the next node $j$ is assigned a higher value than node $i$. Otherwise, $u_i^l - u_j^l \leq |\hat{N}| - 1$. This ensures that there are enough values to assign to all the nodes that the tunnel might pass through.

Finally we consider the interference constraints. For each tunnel pair $(k, l)$ we add a set of constraints depending on whether tunnels $k$ and $l$ interfere with each other. If tunnels $k$ and $l$ do not interfere we have the following constraints:

$$x_{ij}^k + x_{ij}^l \leq 1, \quad \forall i, j, \text{ and } k, l : F(k, l) = 0. \quad (5.5)$$

This ensures that two tunnels that do not interfere with each are never assigned to the same link. If $F(k, l) = 1$, then both the tunnels $k$ and $l$ must appear together in at least one of the links. We enforce this with the following constraints

$$\sum_{i,j} x_{ij}^k \wedge x_{ij}^l \geq 1, \quad \forall i, j, \text{ and } k, l : F(k, l) = 1. \quad (5.6)$$

Here "\wedge" is the logical AND operator, and these constraints can also be transformed into a set of linear (integer) constraints.

The objective function along with the constraints (1) through (6) give the required ILP for identifying a minimal network. After solving the ILP, the graph can be recovered from the links $\{i, j\}$ for which $x_{ij} = 1$. A node that is not used by any of the tunnels can simply be removed from the recovered network.

### 5.2.2 Example

We consider a network where 6 underlay nodes are arranged to form a $3 \times 2$ grid, and an overlay node is attached to each underlay node. The network uses the shortest path routing. The $30 \times 30$ interference matrix $F$ is generated by determining whether two paths intersect with each other. We formulate the ILP with $|\hat{N}| = 12$ then solve it using the Gurobi solver [30].

The original and the recovered network are shown in Figure 5-2. The recovered
network has fewer nodes and edges than the original network. Link \( \{7, 8\} \) in the original network is used only by tunnels \((1, 8, 7, 4)\) and \((4, 7, 8, 1)\) in different directions. Hence there is no interference on this link, and it can be removed without changing the interference matrix. For the same reason, link \( \{11, 12\} \) can be removed to obtain the minimal network. Even after the removal of the links, we can see that the recovered network looks quite similar to the original.

### 5.2.3 Upper bound

We provide an upper bound on the solution to the ILP in the previous section by using a simple algorithm given in Algorithm 10. This algorithm produces a feasible solution to the ILP by assigning two interfering tunnels to a new link in \( \hat{G} \). This algorithm can be suboptimal because in an optimal solution many tunnels can interfere at the same link.

Algorithm 10 starts with a \( \hat{G} \) that is a line graph with \(|E_\mathcal{F}|\) edges, then maps each link in the interference graph \( G_\mathcal{F} \) to a link in \( \hat{G} \). Each edge in \( G_\mathcal{F} \) represents two tunnels that pass through the same edge in \( G \), so if there is an edge between tunnels \( k \) and \( l \) in \( G_\mathcal{F} \), then tunnels \( k \) and \( l \) are assigned to one of the links in \( \hat{G} \). When all the interferences are assigned, it is likely that the same tunnel gets assigned to links that are not attached to each other. In such a case, new links are added to create complete tunnels. An example of this process (Steps 1-3) is given in Figure 5-3. At the end of Step 3, all the interference constraints are satisfied. Steps 4-8 add the overlay nodes
Algorithm 10 FeasibleGraph(\(F, O\)) for obtaining a feasible network \(\hat{G}\)

1. Create a graph \(\hat{G}\) with \(|E_F|\) edges in a line.

2. For each link \(\{k, l\} \in E_F\) assign tunnel \(k\) and \(l\) to a unique edge in \(\hat{G}\). All the tunnels traverse the line graph in the same direction.

3. Connect links in \(\hat{G}\) that have the same tunnel, if they aren’t already connected, such that the tunnels form a loop free path. This can require either a single link, or a node and two links; see example in Figure 5-3.

4. Add nodes \(\hat{O} = \{\hat{o}_1, \hat{o}_2, ..., \hat{o}_{|O|}\}\) to \(\hat{G}\). Each node \(\hat{o}_i \in \hat{O}\) corresponds to an overlay node \(o_i \in O\).

5. For each node \(\hat{o}_i\) add a parent node \(p(\hat{o}_i)\) and edge \(\{\hat{o}_i, p(\hat{o}_i)\}\) to \(\hat{G}\).

6. For each tunnel \(l\) that starts at \(o_i\) assign tunnel \(l\) to link \((\hat{o}_i, p(\hat{o}_i))\).

7. For each tunnel \(l\) that ends at \(o_i\) assign tunnel \(l\) to link \((p(\hat{o}_i), \hat{o}_i)\).

8. Complete the tunnels by connecting \(p(\hat{o}_i)\) to the partial tunnels formed in Step 3.

and makes sure that each overlay node is connected to a single underlay node.

We give the following lemma to show that Algorithm 10 produces a feasible solution to the ILP. Then Theorem 7 establishes the upper bound on the number of links used by this algorithm.

Lemma 11. Algorithm 10 obtains a feasible solution to the ILP in Section 1.

Proof. Steps 5 satisfies Constraint 2. Constraints 3 and 4 are satisfied because the initial graph formed in Step 2 is a line, and the rest of the steps never create branches or rings. Constraints 5 and 6 are satisfied because each interaction in the interference graph, i.e. the \(F\) matrix, is represented in one of the links in \(\hat{G}\), and two tunnels that do not interfere are never assigned to the same link. \(\square\)

Theorem 7. The number of edges required for a feasible solution of the ILP, \(|\hat{E}| \leq |E_F| + 2L|E_F| + |O| + 2L\).

Proof. Lemma 10 shows that Algorithm 1 obtains a feasible solution. We need to establish the upper bound to prove the theorem. Step 1 of the algorithm adds \(|E_F|\)
edges to $\hat{G}$. The number of edges added by Step 3 in the worst case is upper bounded by $2L|E_{\mathcal{F}}|$ because each tunnel can require a maximum of $2|E_{\mathcal{F}}|$ extra edges. Step 5 adds $|O|$ edges, and Step 8 can add a maximum of $2L$ edges. Hence we get the required upper bound.

\begin{proof}
\end{proof}

\subsection{Lower bound}

We establish a lower bound on the number of edges in the minimal graph using the properties of the interference graph. In order to minimize the number of links, we want to assign as many interfering tunnels as possible to the same link. However, we cannot have two tunnels be assigned to the same link if they don’t interfere with each other. This property is nicely abstracted by the cliques in the interference graph $G_{\mathcal{F}}$. The tunnels, represented by the nodes in $G_{\mathcal{F}}$, that are in the same clique interfere with each other. So we can assign all of them to the same link. A lower bound is given by the minimum number of cliques required to cover all the links. For example, two
cliques are needed to cover all the edges of the interference graph in Figure 5-3(a), so we need at least two links in $\hat{G}$ to represent all the interferences. In graph theory the smallest such set is known as the \textit{minimum edge clique cover}\footnote{This is different from the minimum node clique cover which is the smallest set of cliques required to cover all the nodes.}, and the size of such set is known as the \textit{intersection number} of the graph\cite{59}. Computing the minimum edge clique cover of a graph is known to be NP hard so it might not be useful for the purpose of comparing our solutions. However, in the next subsection we will use it to derive conditions when a recovered graph achieves the lower bound and guarantee optimality.

The following lemma presents the lower bound result in terms of the number of directed links required to have a feasible solution. Theorem 8 extends this result to the case with undirected links, which is the setup in this this chapter.

\textbf{Lemma 12.} Let $|\hat{E}_D|$ be the number of directed links required for a feasible solution of the ILP. Let $C$ be the size of the minimum edge clique cover for the interference graph $G_F$. Then $|\hat{E}_D| \geq C$.

\textit{Proof.} A clique $q$ in the minimum edge clique cover of a graph has at least one unique edge, i.e. an edge that is not a part of any other cliques. If this was not the case, then we can obtain a cover with fewer cliques simply by removing clique $q$. Because each edge represents an interference, each unique edge must be assigned to a different link in $\hat{G}$.

If $|\hat{E}_D| < C$, then two unique edges of the interference graph have been assigned to the same edge of $\hat{G}$. This contains at least two tunnels that do not interfere with each other which violates the interference constraints in the ILP.

\textbf{Theorem 8.} Let $|\hat{E}|$ be the number of undirected links required for a feasible solution of the ILP. Let $C$ be the size of the minimum edge clique cover for the interference graph $G_F$. Then,

$$|\hat{E}| \geq \frac{C}{2}.$$

\textit{Proof.} Given a graph with directed edges, we consider the problem of assigning the same tunnels in an undirected network. If every edge in the directed network is used
by the tunnels in both direction, then \(|\hat{E}_D| = 2|\hat{E}|\). That is links \((a, b), (b, a) \in \hat{E}_D\) become a single link \(\{a, b\} \in \hat{E}\). However, in the directed network, some of the links can be used only in one direction. Hence, \(|\hat{E}_D| \leq 2|\hat{E}|\). The result follows directly from Lemma 12.

5.2.5 A sufficient condition for optimality

We give a condition under which a the recovered network has the same number edges as the original network. When this condition is satisfied, the interference pattern cannot be achieved in a smaller network, so this result also provides a sufficient condition for minimality of a network. We prove this result by showing that if the condition is satisfied, then the recovered network achieves the lower bound developed in the previous subsection. We use this result in the subsequent sections to show that our polynomial time algorithms optimally solve the ILP for special networks.

The main result of this subsection says that a given network is minimal if every directed edge in the network is associated with a unique interference (interfering pair of tunnels). Intuitively, this condition seems reasonable because if it is satisfied then each directed link in the graph creates a unique clique in the minimum edge clique cover of the interference graph.

Lemma 13. The size of the minimum edge clique cover of \(G_F\), \(C = 2|E|\) if and only if for each directed edge \((i, j)\) there exists a pair of tunnels \(k^{ij}\) and \(l^{ij}\) such that they intersect at link \((i, j)\) and nowhere else.

Proof. First we show that if there exists tunnel pairs \(k^{ij}\) and \(l^{ij}\) that intersect at link \((i, j)\) and nowhere else, then \(C = 2|E|\). We know that \(G\) provides a feasible solution to the ILP, hence from Theorem 8, \(C \leq 2|E|\). Also, the interference graph \(G_F\) has a clique corresponding to each directed edge in \(G\) as long as some flows intersect in this link. It is sufficient to show that if the condition is satisfied then each clique in \(G_F\) corresponds to a unique link \(G\).

Let \(Q_{ij}\) be the clique corresponding to the directed link \((i, j)\). \(Q_{ij}\) has a link
between the nodes \( k^{ij} \) and \( l^{ij} \), and this link is not part of any other clique. Hence, \( Q_{ij} \) must be a clique in the minimum edge clique cover. This shows that there is one to one correspondence between the cliques in the minimum edge clique cover of \( G_F \) and the links of \( G \).

Next we show that if \( C = 2|E| \) then there exist tunnel pairs \( k^{ij} \) and \( l^{ij} \) that intersect at link \((i, j)\) and nowhere else. Let \( L_{ij} \) be the set of all the tunnels that pass through at link \((i, j)\). Note that \( L_{ij} \) must have at least two tunnels because if \( L_{ij} \) has less than two tunnels then there is no clique corresponding to link \((i, j)\) giving \( C < 2|E| \).

For contradiction, assume that every pair of tunnels \((k, l) \in L_{ij}\) also intersects at some other link \((x^{kl}, y^{kl}) \in E\). Now we can consider a set of cliques corresponding to every link in the network other than link \((i, j)\) and cover all the edges in the interference graph giving \( C < |E| \).

### 5.3 Proof of Theorem 3

We know from Theorem 8 that \( C \leq 2|\hat{E}^*| \). We also know that the original network provides a feasible solution to the ILP, so \( |\hat{E}^*| \leq |E| \). Hence,

\[
C \leq 2|\hat{E}^*| \leq 2|E|.
\]

By Lemma 13, when the condition in the theorem statement is satisfied \( C = 2|E| \). Hence by a sandwiching argument \( |\hat{E}^*| = |E| \).

\[
\square
\]

**Theorem 9.** Let \( C \) be the size of the minimum edge clique cover for the interference graph \( G_F \). Let \( \hat{G}^*(\hat{N}^*, \hat{E}^*) \) be the optimal network obtained by solving the ILP. If every directed link \((i, j) \in E\) has a pair of tunnels \( k^{ij} \) and \( l^{ij} \) such that they intersect at link \((i, j)\) and nowhere else, then \( \hat{G}^* \) has the same number of edges as the original network, i.e. \( |\hat{E}^*| = |E| \).

**Proof.** The proof of this theorem is given in Appendix 5.3. \( \square \)
Note that this theorem provides a sufficient condition but it may not be necessary. That is, there may be graphs where $C < 2|E|$ but the ILP still produces a graph with $|E|$ edges. Also, if the number of edges in the optimal network obtained by solving the ILP is the same as the original network, then we know that the both the networks are minimal. Hence, we can use the condition in the theorem as a sufficient condition for minimality of a network.

**Corollary 4.** A network $G(N, E)$ is minimal if every directed link $(i, j) \in E$ has a pair of tunnels $k^{ij}$ and $l^{ij}$ such that they intersect at link $(i, j)$ and nowhere else.

### 5.4 Identifying Trees

We design a polynomial time algorithm to recover a tree network. If $G$ is a minimal tree, i.e. every non leaf nodes have at least three neighbors and all the leaf nodes are overlay, then this algorithm recovers the tree exactly. A similar result on recovering trees by using distance between the leaf nodes is given in [41], however, the algorithm of [41] requires the network to be minimal. In the situation when the network $G$ is a non-minimal tree, our algorithm produces a $\hat{G}$ that is a minimal tree corresponding to $G$ since both the networks have the same $F$ matrix. Note that there is a unique minimal tree corresponding to each non-minimal tree which can be obtained by using the process discussed in Section 5.1.3.

#### 5.4.1 Algorithm

The tree identification algorithm is given in Algorithm 11. The algorithm uses the interference matrix $F$ to obtain a tree graph $\hat{G}$ with the same $F$. It begins by initializing the graph $\hat{G}$ and checking for terminating conditions in Steps 1 to 3. In Step 4, the algorithm identifies a node $k^*_1$ such that when all its siblings along with itself are removed, its parent becomes a leaf node. This property will later help us compute a new $F$ matrix of the reduced graph. In Step 5, this algorithm finds a group of nodes $X_{k^*}$ that consists of all the sibling nodes of $k^*_1$. Procedure 12 is used
to identify such nodes; see Lemma 15 for proof. These nodes are then added to the recovered graph \( \hat{G} \) in Step 6 by assigning them a common parent node, \( p(X^*_k) \).

Steps 7 removes the sibling nodes in \( X^*_k \) from the original network \( G \). Since the graph \( G \) is not available, the removal is done indirectly by removing the corresponding tunnels from the \( F \) matrix. Note that node \( k^*_1 \) is not removed, instead it is renamed as the parent of the group \( p(X^*_k) \) in Step 8. This works because when all the siblings of \( k^*_i \) are removed, the interference of the tunnels that start or end at \( k^*_i \) is the same as the tunnels that start or end at its parent node. The algorithm is iteratively applied to the reduced \( F \) matrix until only one or two leaf nodes remain.

**Algorithm 11** IdentifyTree\((F, \mathcal{O})\) for recovering a tree network

1. Add the nodes in \( \mathcal{O} \) to \( \hat{G} \).
2. If \( |\mathcal{O}| = 1 \) return \( \hat{G} \).
3. If \( |\mathcal{O}| = 2 \), add an edge between the two nodes in \( \hat{G} \) and return \( \hat{G} \).
4. Identify the tunnel \( k^* \) that interferes with the largest number of other tunnels, \( k^* = \arg \max_k \sum_i F_{ki} \). Let \( k^*_1 \) be the first node of tunnel \( k^* \).
5. For each node \( i \in \mathcal{O} \), use Procedure 12 to decide whether it has the same parent as \( k^*_1 \). Let \( X_{k^*} \) be the set of nodes that successfully pass the test.
6. Add a new node \( p(X_{k^*}) \) to \( \hat{G} \). Connect \( p(X_{k^*}) \) to the nodes in \( X_{k^*} \) in \( \hat{G} \).
7. For each node \( i \in X_{k^*}, i \neq k^*_1 \):
   - Remove rows and columns corresponding to all the tunnels starting or ending at \( i \) from \( F \).
   - Remove node \( i \) from \( \mathcal{O} \).
8. Rename node \( k^*_1 \) to \( p(X_{k^*}) \) so that any tunnel in \( F \) starting or ending at \( k^*_1 \) starts or ends at \( p(X_{k^*}) \) respectively.

An example of the graphs created after the first and the second iterations of this algorithm are shown in Figure 5-4. In the first iteration, Step 4 identifies one of the tunnels that intersect with the most number of other tunnels, \((5,...,1)\). So
$X_k^* = \{5, 6, 7\}$ is obtained in Step 5. This avoids obtaining sibling groups such as $\{3, 4\}$, which when removed does not make their parent a leaf node. Step 6 produces the $\hat{G}$ shown in Figure 5-4(c), and Steps 7 and 8 result in the reduced tree shown in Figure 5-4(b). The $F$ matrix of the reduced tree is obtained by removing all the tunnels with nodes 6 and 7, then renaming node 5 to the parent node $p(5, 6, 7)$. Similarly the result of the second iteration is shown in Figures 5-4(d) and 5-4(e). Since there is only one group of siblings left in the graph after this iteration, the third iteration results in the $G$ with only one node. Also, the third iteration produces the $\hat{G}$ that is identical to the original graph in Figure 5-4(a).

Figure 5-4: First two iterations of the tree identification algorithm. The third iteration (not shown) recovers the complete graph in Figure 5-4(a).
5.4.2 Analysis

In order to prove that Algorithm 11 obtains the minimal tree, we first show that Step 4 identifies a node $k_1^*$ whose parent becomes a leaf node when we perform the node removal in Step 7. In Step 8 of the algorithm, this allows us to use the interference properties of the tunnels starting or ending at $k_1^*$ to obtain the interference of the tunnels of the parent node.

**Lemma 14.** Let $l = (l_1, l_2, ..., l_{|l|})$ be the tunnel that interferes with the largest number of other tunnels. When all the leaf nodes connected to $l_2$ are removed, $l_2$ becomes a leaf node in the resulting graph.

*Proof.* The proof is by contraction. Assume that $l$ interferes with the most number of other tunnels, but when all the siblings of $l_1$ are removed $l_2$ is not a leaf node. Because of this assumption, $l_2$ has at least one neighbor node $n \notin l$ such that $n$ is not a leaf node as shown in Figure 5-5. Since $G$ is a minimal tree the subtree of $n$, formed by removing the link $(n, l_2)$, has at least two leaf nodes $n_1$ and $n_2$.

![Figure 5-5](image)

Figure 5-5: If tunnel $l$ interferes with the most number of tunnels then all the nodes connected to $l_2$ must be leaf nodes. Otherwise, there exists a tunnel $(n_1, n, ..., l_{|l|})$ that interferes with more tunnels than tunnel $l$.

Consider a graph $G'$ formed by removing the neighbors of node $n$ other than $l_2$. In this graph, because of symmetry, a tunnel from $n$ to $l_{|l|}$ interferes with the same number of tunnels as $l$. Clearly, in graph $G$, the tunnel from $n_1$ to $l_{|l|}$ interferes with more tunnels because in addition to all the tunnels that the path from $n$ to $l_{|l|}$ interferes with it also interferes with the tunnel from $n_1$ to $n_2$. This leads to a contradiction.

146
The following lemma shows that Procedure 12 identifies the nodes that share the same parent. The main idea behind the proof is that a path between two nodes that share the same parent interferes with only the tunnels starting or ending at these nodes.

**Lemma 15.** Two leaf nodes of a tree $i$ and $j$ share the same parent if and only if the tunnel from $i$ to $j$ does not interfere with any tunnel $l$ such that $l_1 \neq i$ or $l_{|l|} \neq j$.

**Proof.** If $i$ and $j$ share the same parent, the path from $i$ to $j$ contains only two links $(i, p(i))$ and $(p(i), j)$. None of the tunnels that don’t start in $i$ or end in $j$ use these links, hence no such tunnels $l$ intersect with the tunnel from $i$ to $j$.

If $i$ and $j$ do not share the same parent, $p(i)$ must be connected to a leaf node $i'$ in the subgraph obtained by removing the link $\{l_2, l_3\}$. Similarly $p(j)$ must be connected to a leaf node $j'$ in the subgraph obtained by removing the link $\{l_{|l|-2}, l_{|l|-3}\}$. The tunnel connecting the nodes $i'$ and $j'$ intersects with the tunnel connecting $i$ and $j$.

We will need one more lemma before proving the main theorem. This lemma simply uses Lemma 15 to show that Step 5 identifies the correct set of nodes.

---

**Procedure 12** AreSiblings($\mathcal{F}, i, j$) for checking whether two nodes $i$ and $j$ share the same parent

1. Let $k$ be the tunnel going from node $i$ to $j$.
2. For each tunnel $l$ in the network:
   
   If $\mathcal{F}_{kl} = 1$ and $l_1 \neq i$ and $l_{|l|} \neq j$
   
   Nodes $i$ and $j$ don’t share the same parent.
   
   Return.
3. Let $k$ be the tunnel for $j$ to $i$ and repeat Step 2.
4. Nodes $i$ and $j$ share the same parent.
Lemma 16. Consider the set of nodes $X_{k^*}$ obtained in Step 5 of Algorithm 11. A leaf node $i$ is in $X_{k^*}$ if and only if $i$ shares the same parent as $k^*_i$.

Proof. By Lemma 15, node $i$ and $k^*_i$ pass the test of Algorithm 12 if and only if they share the same parent. Step 3 collects all the nodes that pass the test into $X_{k^*}$ and ignoring any node that doesn’t. Hence, we obtain the required set $X_{k^*}$.

Now we prove the following theorem that shows that the algorithm recovers the minimal tree network.

**Theorem 10.** If a given network $G$ is a minimal tree, then Algorithm 11 recovers the network.

Proof. By Lemmas 14 and 16 we can see that Steps 5 and 6 identify a group of sibling nodes such that removing them makes their parent a leaf node. Steps 7 and 8 produce a the $F$ matrix of a tree with the siblings of $k^*_i$ pruned. The new $F$ matrix so formed corresponds to the such a tree because interference of tunnels starting or ending on the node $p(k^*_i)$ is exactly the same the tunnels starting or ending at node $k^*_i$ when its siblings are removed. Meanwhile, the pruned portion tree is recreated in $\hat{G}$ at every iteration of Step 5. Hence, when all the nodes in $G$ are removed, the complete graph is created in $\hat{G}$.

Note that not only the recovered graph $\hat{G}$ is isomorphic to $G$, the relative positions of the overlay nodes are the same. That is if the overlay nodes $i$ and $j$ share the same parent in $G$, they also share the same parent in $\hat{G}$. Also, because of the fact that the $F$ matrix for a non minimal tree is the same as that of the minimal version of the tree, and the minimal tree is unique for any non-minimal tree, we get the following corollary.

**Corollary 5.** If a given network $G$ is a non-minimal tree, then the tree $\hat{G}$ recovered by Algorithm 11 is the unique minimal tree for $G$.

The following corollary states that the graph generated by the tree algorithm solves the ILP optimally. This is true simply because all minimal trees satisfy the condition of Theorem 9.
Corollary 6. If the interference pattern in a $F$ matrix can be represented in a tree, 
Algorithm 11 produces a $\hat{G}$ that solves the ILP optimally.

Note that even when $G$ is not a tree, Algorithm 11 can produce a tree as long as the interference can be represented by a tree. However if the interference pattern cannot be represented by a tree this algorithm will either fail Step 4, or the algorithm terminates but the recovered $\hat{G}$ has a different interference matrix than $F$.

5.5 Identifying Rings

We now consider a situation when the $F$ matrix cannot be represented in a tree. Specifically we consider a graph $G$ where the underlay nodes are arranged in a ring, and each underlay node is attached to exactly one overlay node. Also, we will assume that the network uses a shortest path routing algorithm, hence, the tunnels take the shortest paths between the overlay nodes. If the $F$ matrix can be represented in a ring, our algorithm identifies the order of the overlay nodes. Note that knowing the order of the nodes gives more information than just recovering isomorphic graphs, e.g. in [2]. Just like the tree discovery algorithm in the previous section, this algorithm can also be used to show that a particular network is not a ring.

5.5.1 Algorithm

The ring identification algorithm is given in Algorithm 13. This algorithm builds the ring in an incremental fashion. First, in Step 1 an overlay node $i$ and its parent node $p(i)$ are added to $\hat{G}$. The key idea behind the algorithm is in Step 2. It uses the $F$ matrix to identify two overlay nodes in the ring that are closest to node $i$, i.e. two nodes such that their parents are neighbors of $p(i)$. In Steps 3 to 5 we attach the two nodes to their parents, and connect the parents to $p(i)$. 

149
Algorithm 13 IdentifyRing($\mathcal{F}, \mathcal{O}$) for recovering a ring network
For each overlay node $i \in \mathcal{O}$:

1. If $i$ is not in $\hat{G}$, add two nodes $i$ and $p(i)$ to $\hat{G}$. Add an edge $\{i, p(i)\}$ to $\hat{G}$.

2. Identify two tunnels starting at node $i$ that interfere with the least number of other tunnels. Call these tunnels $k^*$ and $l^*$.

3. If node $k^*_{|k^*|}$ is not in $\hat{G}$, add two nodes $k^*_{|k^*|}$ and $p(k^*_{|k^*|})$. Add edge $\{k^*_{|k^*|}, p(k^*_{|k^*|})\}$.

4. Add edge $\{p(i), p(k^*_{|k^*|})\}$ to $\hat{G}$ if it doesn’t exist.

5. Repeat Steps 3 and 4 for node $l^*_{|l^*|}$.

5.5.2 Analysis

We will show that Algorithm 13 is guaranteed to recover the correct ring if $|\mathcal{O}| \geq 5$. For $|\mathcal{O}| = 3$, any ordering of the nodes is the same because the network links are bidirectional, so, using the algorithm is unnecessary. The algorithm might not produce the correct result for the a network with $|\mathcal{O}| = 4$ if the tunnels between the nodes in opposite sides pass through the same set of nodes. The networks in both of these situations with 3 or 4 overlay nodes in a ring are not minimal.

Lemma 17. Let $G$ be a graph where the underlay nodes are arranged in a ring, and each underlay node is connected to exactly one overlay node. Let $|\mathcal{O}| \geq 5$. Let $i$ and $j$ be two overlay nodes and $l$ be the tunnel from $i$ to $j$. Underlay nodes $p(i)$ and $p(j)$ are neighbors if and only if tunnel $l$ interferes with the fewest number of other tunnels.

Proof. Let $\mathcal{O} = \{1, 2, ..., n\}$ where $n = |\mathcal{O}|$. Assume that the correct ordering of the nodes in the ring is $p(1), p(2), ..., p(n)$. We want to show that tunnels from node 1 to 2 and 1 to $n$ intersect with fewer tunnels than any other tunnel that start at node 1.

We begin by showing that the tunnel $(1, ..., 2)$ intersects with fewer tunnels than tunnel $(1, ..., 3)$. These tunnels share the links $(1, p(1))$ and $(p(1), p(2))$. So any tunnel passing through these links intersect with both the tunnels. Also, because of symmetry, the number of tunnels intersecting with tunnel $(1, ..., 2)$ only at link $(p(2), 2)$ is equal to the number of tunnels intersecting with tunnel $(1, ..., 3)$ only at link $(p(3), 3)$.
link \((p(3), 3)\). The tunnel \((2, \ldots, 4)\) does not intersect with tunnel \((1, \ldots, 2)\) however it intersects with tunnel \((1, \ldots, 3)\) only at link \((p(2), p(3))\). Hence tunnel \((1, \ldots, 3)\) intersects with at least one more link than tunnel \((1, \ldots, 2)\). Clearly, any longer tunnel starting at node 1 must interfere with even more tunnels.

The ring has at least 5 nodes and the network is using the shortest path routing, so we can apply the same argument as above to show that to show that \((1, \ldots, n)\) also intersects with the fewest number of tunnels among the tunnels starting at node 1 and passing through link \((p(1), p(n))\). Since all tunnels that start at node 1 has to pass through either \(p(n)\) or \(p(1)\), these two tunnels must be the ones that intersect with the fewest other tunnels that start at node 1.

Because of symmetry this property holds for tunnels starting at every node in the network. This completes the proof.

Algorithm 13 identifies overlay nodes whose parent nodes are neighbors and pieces them together into a ring. Hence, Theorem 11 follows directly from the lemma above.

**Theorem 11.** If the given network \(G\) is a minimal ring, Algorithm 13 recovers the network.

Similar to the tree identification algorithm, this algorithm will produce a corresponding minimal ring if the original network is a non-minimal ring. This is true because both the rings have the same \(F\) matrix. Also, because a minimal ring satisfies the sufficient condition for minimality, this algorithm optimally solves the ILP for ring networks. Hence we get the following two corollaries.

**Corollary 7.** If a given network \(G\) is a non-minimal ring with \(|\mathcal{O}| \geq 5\), then the ring \(\hat{G}\) recovered by Algorithm 13 is the unique minimal ring for \(G\).

**Corollary 8.** If the interference pattern in the \(F\) matrix with \(|\mathcal{O}| \geq 5\) can be represented in a ring, Algorithm 13 produces a \(\hat{G}\) that solves the ILP optimally.
5.6 Identifying General networks

Inspired by the algorithms for identifying trees and rings in the previous sections, we develop a scheme for identifying general networks. A network can consist of trees and rings connected to each other. Our algorithm assumes that the network uses shortest path routing, and attempts to separate the trees from the rest of the graph, and identify these components separately. We will use Algorithm 11 for recovering the trees, and we will design a new algorithm inspired by Algorithm 13 for the non-tree components. Finally we will combine the discovered components to obtain the full network. This scheme is largely a heuristic, hence, we will compare its performance against another algorithm that also discovers general graphs.

5.6.1 Algorithm

We first present Algorithm 14 which is designed to recover a graph where every underlay node is part of one or more cycles and only one overlay node is attached to each underlay node. The algorithm works in similar fashion as the ring recovery algorithm from the previous section. The difference is that now each underlay node can have more than two underlay neighbors. So, for each overlay node \( i \), the algorithm attempts to find all the overlay nodes whose parents are neighbors of \( p(i) \). For clarity, we present this part of the algorithm separately in Procedure 15.

The main idea behind Procedure 15 is shown in an example in Figure 5-6. For Node 1, the procedure first identifies two neighbors of \( p(1) \) using the tunnels that start at 1 and intersects with the fewest number of other tunnels. The intuition behind this is the same as the ring algorithm from the previous section, however, when there are more than one rings, it is not guaranteed that the shortest tunnels have the fewest number interferences. It is possible that the tunnel \((1,...,5)\) intersects with the same number of tunnels as \((1,...,3)\). After identifying the two neighbors, the procedure avoids any tunnels that pass through these neighbors and identifies other shortest tunnels.

Finally, we present Algorithm 16 for identifying networks with multiple rings and
Figure 5-6: Example of Procedure 15 at work. Node \( p(1) \) has three neighbors \( p(2), p(3) \) and \( p(4) \). Procedure 15 first attempts to identify two nodes, e.g. 2 and 4, by minimizing the number of tunnel intersections. Then node 3 is identified by using the property that tunnel \((3,\ldots,1)\) doesn’t interfere with the tunnels \((2,\ldots,4)\) or \((4,\ldots,2)\).

Algorithm 14 IdentifyRings\((\mathcal{F}, \mathcal{O})\) for recovering a non-tree network

Initialize \( \hat{G} \) to empty graph.

For each overlay node \( i \in \mathcal{O} \):

1. Obtain the to neighboring nodes, \( R = \text{allNeighbors}(i) \).

2. For each \( r \in R \):

   (a) If node \( r \) is not in \( \hat{G} \) add nodes \( r \) and \( p(r) \) and edge \( \{p(i), p(r)\} \) to \( \hat{G} \).

   (b) Add edge \( \{p(i), p(r)\} \) to \( \hat{G} \) if it doesn’t exist.

trees. In Step 2, this algorithm identifies sets of overlay nodes that could be a part of a tree using Procedure 12. Step 2(i) identifies the siblings, \( X \), of node \( i \). Step 2(ii) obtains the siblings of all the nodes in \( X \). If \( j \) is a sibling of \( i \), then \( i \) must also be a sibling of \( j \). Using this property, Step 2(iii) attempts to reduce false positives. Step 2(iv) adds the nodes that are identified as part of a tree into the set of existing nodes. If some part of the tree containing the nodes in \( S \) have already been identified, then these nodes must have one node in common with \( S \), i.e. \( S' \) exists. In such a case, nodes in \( S \) is added to \( S' \), otherwise \( S \) is added as a new element \( \mathcal{C} \). The tunnels belonging to all but one node in \( S \) are removed from \( \mathcal{F} \), and Step 2 is repeated on this new interference matrix. The completion of Step 2 produces the set \( \mathcal{C} \) such that each element of \( \mathcal{C} \) is a set of nodes that belong to the same tree.
Procedure 15 allNeighbors(\( F, O, i \)) for finding all \( j \) such that \( p(i) \) and \( p(j) \) are neighbors

1. Identify two tunnels starting at node \( i \) that interfere with the least number of other tunnels. Add the end nodes of these tunnels to set \( R \).

2. For each \( n \in (O \setminus R) \), find a tunnel \( k = (1, \ldots, n) \) such that it interferes with the fewest number of tunnels and does not interfere with any tunnel \( l \) such that \( l_1, l_{|l|} \in R \).

3. If tunnel \( k \) exists, add \( k_{|k|} \) to \( R \) and goto Step 2.

4. Return \( R \).

Step 3 of the algorithm retrieves the original \( F \) matrix. Then in Step 4, Algorithm 11 is used on the elements of \( C \) to discover their corresponding trees. If the tree identification algorithm completes successfully, then all but one of the overlay nodes belonging to the tree are removed from the \( F \) matrix. The node that is not removed acts as an anchor node while combing the trees and the rest of the graph. In Step 5, the resulting \( F \) matrix is then used in Algorithm 5 to recover the non-tree part of the graph. In order to combine a tree with the non-tree graph, in Step 6, the anchor node corresponding to the tree is found in the graph. Then in Steps 6(ii) and 6(iii), attempts are made to connect the tree to the anchor node at different locations in tree. The algorithm keeps the connection that minimizes the difference between the interference matrix of the resulting graph \( \hat{G} \) and the original \( F \) matrix.

5.6.2 Simulation result

We compare the performance of Algorithm 16 against that of RGD1 algorithm from [2]. For the implementation of RGD1, we obtain the exact length of each path by using a shortest path algorithm. All links are assumed to have unit length. We choose the parameter \( Rg + \tau \) to be 4. We also tried the value of 3 and 5 for this parameter, however, the performance was not as good.

The graphs used to obtain the data for the simulation were generated to be similar to the random graphs considered in [2]. We first generate an Erdős-Rényi random
Algorithm 16 IdentifyGeneral(\(\mathcal{F}, \mathcal{O}\)) for recovering general networks

Initialize \(\hat{G}\) to empty graph.

1. Let \(\mathcal{C}\) be an empty set. Let \(\mathcal{F}' = \mathcal{F}\).

2. For each \(i \in \mathcal{O}\):
   i. Use Procedure 12 to find the set of nodes that share the same parent as \(i\).
      Let \(X\) be the set.
   ii. For each \(j \in X\) use Procedure 12 to find the set of nodes that share the
       same same parent as \(j\). Let \(X_j\) be the set.
   iii. Let \(S = X \cap (\cap_j X_j)\)
   iv. If \(|S| > 1,\)
      a) Let \(S' \in \mathcal{C}\) be a set of nodes such that \(S' \cap S \neq \emptyset\).
      b) If such \(S'\) exists, \(S' := S' \cup S\). Otherwise, \(\mathcal{C} := \mathcal{C} \cup \{S\}\)
      c) Let \(x\) be an arbitrary node in \(S\). Let \(S := S \setminus \{x\}\).
      d) Remove tunnels \(l\) from \(\mathcal{F}\) if \(l_1 \in S\) or \(l_{ij} \in S\). Let \(\mathcal{O} := \mathcal{O} \setminus S\).
      e) Restart Step 2.

3. Let \(\mathcal{F} := \mathcal{F}'\).

4. For each \(S \in \mathcal{C}\):
   i. Use Algorithm 11 on the nodes in \(S\). Let \(T\) be the corresponding tree.
   ii. If the algorithm fails to produce a tree, continue.
   iii. Remove tunnels \(l\) from \(\mathcal{F}\) if \(l_1 \in S\) or \(l_{ij} \in S\). Let \(\mathcal{O} := \mathcal{O} \setminus S\).

5. Use Algorithm 5 on the remaining \(\mathcal{F}\) to obtain \(\hat{G}\).

6. For each tree \(T \in \mathcal{T}\):
   i. Find the overlay node \(i\) that is common to \(T\) and \(\hat{G}\).
   ii. For each underlay node \(j\) of \(T\), add \(T\) to \(\hat{G}\) by replacing \(i\) in \(\hat{G}\) by node \(j\)
      of the tree. Calculate the interference matrix for each \(j\).
   iii. For each underlay node \(j\) of \(T\) add \(T\) to \(\hat{G}\) by replacing \(p(i)\) in \(\hat{G}\) by node
      \(j\) of the tree. Calculate the interference matrix for each \(j\).
   iv. Keep the \(\hat{G}\) that produces the interference matrix closest to \(\mathcal{F}\) in Steps ii
       and iii.

Graph with parameters \(\mathcal{G}(n, 2/n)\). Then we find the largest connected component of
the graph, and remove all the other nodes that do not belong to this component.
We then attach overlay nodes to 80% of the remaining nodes uniformly at random. Finally, we remove any underlay nodes that have degree less than 3 by using the process discussed in Section 5.1.3. We generate 100 networks for each value of $n$, where $n = 10, 20, ..., 50$ and obtain the measurements required for both algorithms: distances for RGD1 and the $F$ matrix for our algorithm. Finally, we use the measurements to recover the graphs.

The performance of the two algorithms was measured by computing the edit distance between the original graph $G$ and the recovered graph $\hat{G}$. Edit distance measures the number of links in $\hat{G}$ that needs to be added or removed in order to make it isomorphic to $G$. This metric is similar to the metric used in [2] to obtain the asymptotic bounds of RGD1. Unfortunately, calculating the graph edit distance is an NP-hard problem, so we use an open source tool called GEDEVO [34] to approximate it.

The results of the simulations are given in Figure 5-7. Figures 5-7(a) and 5-7(b) show the performance of the two algorithms for each of the 100 graphs that were generated. We can see that in most of the cases, our algorithm outperforms RDG1. Figure 5-7(c) shows the average performance of the two algorithms across different values of $n$. Again, we can see that our algorithm outperforms RGD1.

5.7 Conclusion

We developed a new method for discovering the topology of a network. It uses the path interference information, which can be obtained by using the measurements available at the end nodes. Using the path interference, we formulated an integer linear program that finds a minimal graph that can contain all the interferences. We then developed polynomial time algorithms that solve the ILP for the special cases when the network is a tree or a cycle. Finally, we developed a heuristic for identifying general networks and compared its performance to a well known algorithm. Future research in the area will focus on developing better heuristics for general networks and providing performance guarantees.
(a) Edit distances for all the iterations with $n=10$.

(b) Edit distances for all the iterations with $n=50$.

(c) Average edit distance for each value of $n$.

Figure 5-7: Comparison of Algorithm 16 and RGD1.
Chapter 6

Conclusion

In this thesis, we developed three algorithms, two on network control and one on network inference, all geared towards making the existing throughput optimal network control algorithms more practical. Our algorithms make it possible to build high throughput and robust computer networks in an incremental fashion, and they can obtain good delay performance while simultaneously achieving the maximum throughput supported by the network.

In Chapter 2, we observed that packets traversing loops in the network is a major cause of high delay in the backpressure routing algorithm. We solved this looping problem by restricting the routing to a directed acyclic graph (DAG). If the DAG doesn’t have enough capacity to support the traffic demand, some of the nodes become overloaded. To improve the capacity of the DAG, we developed a link-reversal algorithm that obtains a new DAG by reversing links that go from a non-overloaded node to an overloaded node. We showed that such a reversal process obtains an optimal DAG in a finite number of iterations. Furthermore, we designed a scheme so that the DAG can be maintained even when the network topology changes over time. We showed via simulations that our algorithm obtained much better delay performance compared to the traditional backpressure algorithm.

In Chapters 3 and 4, we tackled another problem with many optimal routing algorithms that arises from the fact that in order for such an algorithm to achieve optimality, every node in the network much execute the same algorithm. However,
almost none of the existing device implement throughput optimal algorithms such as backpressure. So we proposed designing an overlay network of controllable nodes on top of the legacy network, and we provided a new throughput optimal algorithm for such an overlay network. In Chapter 3, we began by showing that the existing schemes for routing in overlay networks are suboptimal. Then we derived a new throughput optimal routing algorithm, called Optimal Overlay Routing Policy (OORP), using the dual subgradient descent method. This algorithm can be completely implemented at the overlay nodes; however, it requires the size of the backlog in the underlay paths as the feedback. Since the underlay might not be able to provide this information, in Chapter 4, we designed a scheme to estimate it. We modeled the backlog as a function of the number of packets in flights and we fit linear and piecewise linear functions using regression. We also showed that historical data for training such models can be obtained by using the delay. Simulation results showed the OORP obtains throughput optimality when used in conjunction with these estimators.

The knowledge of the topology of the underlay network is essential to design a high throughput and robust overlay network. However, this information might not be available to the public. Hence in Chapter 5, we designed an algorithm to infer the topology of underlay networks by using the measurements available at the overlay nodes. We described a simple scheme to decide whether any two paths interfere with each other by using the regression based method from Chapter 4. Then using the interference information, we formulated the topology discovery problem as an integer linear program. Since integer programs are computationally infeasible, we developed polynomial time algorithms for special networks, namely networks with tree and ring topology. Finally, inspired by these algorithms, we design a heuristic that can recover a general network topology.
Bibliography


