MEASURING THE INTERNET AS GRAPH AND ITS EVOLUTION

by

PETER MATTISON BOOTHE

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2009

University of Oregon Graduate School

Confirmation of Approval and Acceptance of Dissertation prepared by:

Peter Boothe

Title:

"Measuring the Internet AS Graph and its Evolution"

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer & Information Science by:

Andrzej Proskurowski, Chairperson, Computer & Information Science
Arthur Farley, Member, Computer & Information Science
Jun Li, Member, Computer & Information Science
Anne van den Nouweland, Outside Member, Economics

and Richard Linton, Vice President for Research and Graduate Studies/Dean of the Graduate School for the University of Oregon.

September 5, 2009

Original approval signatures are on file with the Graduate School and the University of Oregon Libraries.

An Abstract of the Thesis of

Peter Mattison Boothe for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken September 2009

Title: MEASURING THE INTERNET AS GRAPH AND ITS EVOLUTION

Approved: _____

Dr. Andrzej Proskurowski, Chair

As the Internet has evolved over time, the interconnection patterns of the members of this "network of networks" have changed. Can we characterize those changes? Have those changes been good or bad? What does "good" mean in this context? Has market power been centralizing or decentralizing? How certain can we be of our answer? What are the limitations of our data? These are the questions which motivate this dissertation. In this dissertation, we answer these questions and more by carefully taking a long-term quantitative study of the evolution of the topology of the Internet's AS graph. In order to do this study, we spend most of the dissertation developing methods of data processing and data analysis all informed by ideas from networking, data mining, graph theory, and statistics. The contributions are both theoretical and practical. The theoretical contributions include an in-depth analysis of the complexity of AS graph measurement as well as

of the difficulty of reconstructing the AS graph from available data. The practical contributions include the design of graph metrics to capture properties of interest, usable approximation algorithms for several AS graph analysis methods, and an analysis of the evolution of the AS graph over time.

It is our hope that these methods may prove useful in other domains, and that the conclusions about the evolution of the Internet topology prove useful for Internet operators, network researchers, policy makers, and others.

CURRICULUM VITAE

NAME OF AUTHOR: Peter Mattison Boothe

PLACE OF BIRTH: Ellsworth, ME, U.S.A.

DATE OF BIRTH: July 6, 1978

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon
Harvey Mudd College, Claremont, California

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2009,
University of Oregon
Bachelor of Science, Joint Major in Computer Science and
Mathematics, 2000, Harvey Mudd College

AREAS OF SPECIAL INTEREST:

Graph theory
Internet measurement
Algorithms
Computer science education

PROFESSIONAL EXPERIENCE:

Software Developer, Gordian
Teaching Assistant, University of Oregon CIS Department
Research Assistant, Beyond BGP Project
Teaching Assistant, University of Oregon CIS Department
Software Developer, University of Oregon Computing Center
Assistant Professor of Computer Science, Manhattan College

PUBLICATIONS:

Peter Boothe, Zdeněk Dvořák, Art Farley, and Andrzej Proskurowski. Graph Covering via Shortest Paths. *Congressus Numerantium*, 2007

James Hiebert, Peter Boothe, Randy Bush, and Lucy Lynch. Determining the Cause and Frequency of Routing Instability with Anycast. *Proceedings of the Asian Internet Engineering Conference (AINTEC)*, 2006, pp. 172–185

R. Hadas, J. Hartline, P. Boothe, G. Rae, and J. Swisher. On multicast algorithms for heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 61:1665–1679, 2001

ACKNOWLEDGMENTS

Dissertations take a long time to come into existence, and although only one person's name gets on the cover, in my case it took a village. Friends, family, coworkers and bosses all played a role, and the roles are inherently inextricable. Despite their inextricability, however, some are bigger than others. For example: Thank you to Tom and Betty Boothe for being my parents! Without you, I would not exist! For another example: Thank you to Tracy van Cort for supporting me throughout my graduate career! Without you, I would not have finished!

I entered the University of Oregon Computer Science Department with twelve others, and four of us are leaving with a PhD. That's a small enough number that I can just thank Julian Catchen, Kevin Huck, and Jeremy Ludwig right here. Along the way I was also rendered invaluable assistance by James Hiebert, Dave Hofer, John Lasseter, Dan Stutzbach, Eric Wills, and many others.

Prior to defending this thesis, I joined the Mathematics and Computer Science department at Manhattan College. Their forbearance as "the last few months" of my studies stretched into the last year of my studies has been wonderful, and I appreciate it very much. Also, while I of course thank my committee, I'd also like to thank Dr. Timur Friedman, who gave me fantastic feedback and was on my committee in spirit, but was not able to be officially included due to the twisty nature of the rules of the graduate school.

Finally, I'd like to thank Cheri Smith, Star Holmberg, and Jan Saunders. If the person reading this dissertation learns nothing else, then let me pass down some of the most important advice I ever received from my father (thanks Dad!): Always be kind to the office staff; they are the ones who actually run the place.

DEDICATION

Dedicated to Tracy van Cort.

Go team.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

In which we introduce the problem and map out the course of this dissertation

How has the Internet topology changed over time, and have those changes been good? This is the central question that drives this whole dissertation. To answer it, we delve into quite a few areas of computer science, including data mining, graph theory, algorithms, and networking. Along the way, we will use methods from economics and statistics as needed.

In order to answer this deceptively simple question, we must: pin down exactly what we mean by "Internet topology", find data from which can reconstruct that topology, and then analyze the data while taking into account any biases. The analysis step involves translating the idea of a "good" or 'bad" topology into more concrete goals of desirable and undesirable properties for a robust Internet topology, and then translating these properties into the language of graph theory and apply this new theory to the available data.

We break this dissertation down a similar structure which mirrors these steps. We define our terms, starting with "the Internet", in Chapter II. After suitable definitions have been discussed and the area is well established, we have two main

classes of problem — practical problems such as "Where can Internet topology data be found and how should it be processed and stored?", and theoretical problems such as "How complete is this data and how should it be analyzed?" We conduct a survey of the network topology at each of the network layers and choose the autonomous system graph as our primary object of study. This chapter serves as a survey to provide the necessary background for the rest of the dissertation.

Chapter III discusses practical issues of data availability and storage and eventually frames the problem using the language and terms of data mining. The solutions to the lower level problems of data acquisition and data parsing have long been treated as too trivial to write down, which has made approaching them very difficult for those who are not already deeply involved in both graph theory and computer networking. We lay out explicitly what the data means and how it may be turned into a usable form. The issue of data completeness has been dealt with before, but this chapter presents a synthesis of the best known approaches.

We then develop a graph-theoretic model of how these measurements were taken in Chapter IV. We take the best model for Internet paths, and present the idea of network measurement as a graph covering problem. We prove that most of the problems which naturally arise in this context are $\mathcal{NP}$-complete, but there exist many classes of graphs upon which they are efficiently solvable. We refine our measurement model to more accurately reflect the challenges of Internet measurement, and prove that with refinement, the problem remains $\mathcal{NP}$-complete. We survey other work that has been done in this area, but our formulation of the basic problem is new, and therefore so are all the results in this chapter.

In Chapters V we discuss methods of dealing with the inherent incompleteness of our data and assess the relative quality of these methods. It turns out that past efforts at Internet topology analysis have largely analyzed the available measurements, and assumed that the measurements were an accurate reflection of

the Internet topology. In this chapter, we propose an analysis pipeline that can take the existing data and analyze almost any property while still being robust to the fact that some data is missing. The analysis techniques are adapted from statistics, but have never been applied in this manner before, and are extremely useful when faced with the inherent problems contained in our data set. To date, no study has attempted to analyze the graph or family of graphs from which the measurements may have come. Researchers have been generally content with analyzing the measurements, and simply acknowledging their incompleteness. Our departure from this model represents a significant contribution towards ensuring the accuracy of Internet AS graph analysis.

Finally, in Chapter VI, we put our data reconstruction and analysis techniques to the test by measuring how a host of graph metrics have changed over time. We also develop a graph theoretic measurement of the power of an Internet traffic oligopoly, and then find how that measurement has changed over time. Taking the measurement turns out to be a difficult computational task, both theoretically and practically, and we deal with each of the difficulties in turn. We end up with evidence that traffic flow on the Internet has become increasingly centralized over time, which has implications for both protocol designers and policy makers. This chapter can be thought of as an example of what can be done with the solutions developed in the previous chapters. Many of the studies have been performed before for a snapshot of the Internet, but no longitudinal study has ever been performed. That makes the results of this chapter a significant contribution, particularly given that we perform our analysis using the methods of the previous chapters, which allow us to not only attempt to analyze historical data, but, for the first time, to bound the certainty of our results.

It is important to note that many of the developed methods have broader use than just analyzing properties of the Internet graph. The potential broader impacts

of each aspect of the solution are discussed at the end of each chapter. The chapter on definitions is required background reading for having the rest of the dissertation make sense, but after that it should be possible to pick and choose which chapters to read depending on interest.

CHAPTER II

BACKGROUND

Where we define the objects we are going to study

In order to have a well defined problem, the terms in the question must themselves be well-defined. To ask "What is the Internet topology?", we must first define "the Internet" and then define "the Internet topology". Due to an obvious dependency, we answer these questions in that order.

2.1 The Internet Versus an Internet

An internet is a collection of interconnected networks, much like an interstate highway system is a collection of roads between independent states. The Internet is the largest and most popular public internet in the world, much like the Interstate Highway System is the largest and most popular interstate road system in the United States. When discussing problems, we must be careful to state which problems are universal for all internets, and which problems are specific to the capital-I Internet.[1] The Internet is a very specific internet, and the problems it faces

[1]Like many things which only matter a very little bit, whether or not to capitalize the "I" in Internet has been the subject of heated debate. The academic community has firmly settled on capitalization, but others (most notably WIRED magazine and The Economist) have decided that

are often unique to it, and, because of the Internet's specific history and the frameworks within which it grew, conclusions about the Internet do not necessarily generalize to all internets. Confusion between the general ideas of internets and inter-networking and the specifics of the existing Internet lie at the root of many fruitless discussions.

Among Internet users there is a massive confusion about what the Internet is. When we wish to study the Internet, we must be very careful to state exactly what aspect of this multi-layered multi-application multi-national network is of interest, and we must be very careful in our definitions.

The Internet is a network of networks. Using the **Internet Protocol** (**IP**), a member of a local-area-network (LAN) can communicate to members of other LANs. These LANs are joined together by requiring a common protocol (IP), as well as through the inclusion of special network hardware named routers, which send traffic in different directions based on the IP address of the recipient. When the Internet was first designed, routers were called Internet Message Processors (IMPs) and the network topology was entered by hand into each router. The routers acted as gateways between the LAN and the rest of the Internet. The Internet started out small, which meant that the administrative burden of maintaining each router by hand was also relatively small. As the Internet grew, however, the amount of state that each router had to maintain grew as well. In an effort to manage this increase in complexity, a second layer of abstraction was required. To design this second layer and the protocols used to communicate along it, the designers took another look at the networks that made up the Internet at the time, and then looked at how the network was evolving:

---

that the capitalization is vestigial and should be eliminated. Here we will capitalize when talking about the Internet, and use lowercase to indicate a generic internet. *Caveat lector*, as the usage in other sources may differ without note.

> In the future, the internet is expected to evolve into a set of separate domains or "autonomous systems", each of which consists of a set of one or more relatively homogeneous gateways. [...]
>
> Ultimately, however, the internet may consist of a number of co-equal autonomous systems, any of which may be used (with certain restrictions which will be discussed later) as a transport medium for traffic originating in any system and destined for any system.[59]

In this quote, we can see that the original operators of the Internet were imagining a system much like the one we now have. Our original description that the Internet is the largest international network of networks was a correct one, but that statement only tells a tiny bit of the story. Although the Internet's ancestors, Arpanet, NSFnet, and Milnet, were originally government funded and centrally managed, the Internet now consists largely of entities freely choosing to associate and exchange traffic in accordance with contracts and policies that range from straightforward to truly byzantine. These entities exchanging traffic run the gamut from educational institutions to telecommunications companies to Internet service providers to national governments to local cooperatives.

The one attribute these entities have in common is that they each control their own network, so we call each one an **autonomous system** (AS). These autonomous systems then make agreements with each other in an effort to interconnect and exchange traffic in accordance with each AS's goals. When money exchanges hands in AS agreements, we call one network a **provider** and the other a **customer**. The provider agrees to provide **transit** to and through the provider's network for all traffic from the customer's network. Other times, the two autonomous systems agree to exchange traffic freely in a practice known as **peering**, and two ASes in such a relationship are called **peers**. Peering differs from transit, because transit allows for traffic to go "to and through" the provider's network to any location that provider can reach, while peering only allows the two peers to reach each other and through to each others' customers, but not through to

each others' providers or peers. No matter what, though, it is important to note that there is no central organizing authority — companies come and go, telecommunications regulations in various countries come and go, and agreements get made and broken, and almost none of these activities require the imprimatur of an authority other than the entities making the deals. While there do exist organizations which attempt to coordinate network activity (IANA[2], RIPE NCC[3], LACNIC[4], APNIC[5], etc.), they generally restrict themselves to handing out blocks of IP addresses and autonomous system numbers to the ASes that require them. As a general rule, two ASes may form or dissolve a link of any kind without interference or approval from any third party. The small number of exceptions that do require a stamp of approval generally involve telephone companies or other companies historically regulated by the United States Federal Communications Commission (FCC) or the national equivalent in another country.

There have been some high-profile attempts to lay out an explicit Internet governance policy by both the United States and the United Nations, culminating in the Working Group on Internet Governance and the World Summit for the Information Society[69], but the Internet currently remains a self-organizing system with markedly little outside regulation. The best description available of how the peering and interconnection process works from the point of view of a network operator is a constantly-evolving paper by William Norton, co-founder and chief technical liaison of the ISP Equinix[54].

In a very real sense, it is astounding that this radically hands-off approach could work at all for something as complex as the international Internet. Autonomous

[2]The Internet Assigned Numbers Authority

[3]Réseaux IP Européens Network Coordination Centre

[4]The Latin American and Caribbean Internet Addresses Registry

[5]Asia-Pacific Network Information Centre

systems peer and pay for transit until they can reach every other AS that they might care about, and in the end almost everyone can send packets to almost everyone else. Although there is no edict from on high for massive inter-network connectivity, almost every host can communicate with almost every other host[33]. The emergent behaviors of this system have taken people completely off guard, and predicting future behavior is very much an open problem[53]. One of the most surprising things has been the degree to which the Internet has become a powerful economic force in the world, and helping spread communication and culture across the globe. Unanswered in this is *how* has it been growing and changing, and whether the changes been in the direction that policymakers and the Internet-using public desire.

When the original design documents of the Internet were being written, there were specific design goals in mind[23]. These design goals were, in order of decreasing importance: the ability to connect together multiple networks, robustness to individual network failures, support for multiple communication services, support for a wide variety of network technologies, distributed management, cost effectiveness, ease of attachment, and, lastly, accountability. In recent years, after the commercialization of the Internet, economic concerns have become more and more important[4] as policy makers want the Internet to be not just cost effective, but also a level playing field with very few barriers to prevent competition. Decisions by the FCC have been predicated on the idea that the Internet is a fair and open market with low barriers to entry, but whether the Internet actually behaves like an open market has never been put to quantitative study.

More recently, the idea of network neutrality has gotten significant attention in the press, but there exist multiple competing definitions of the term. Not only that, but for each definition, the question of whether maintaining the current level of network neutrality that exists on today's Internet is an economic inevitability or requires legislative attention is still a subject of active debate. All of these network

issues and more are oft-discussed, but there is very little hard data and analysis on these subjects to guide the debate, and so most discussions end up generating far more heat than light.

2.2 The Layered Internet

In order to properly understand the Internet topology, we must decide what layer to look at, because there are different logical topologies at each layer, and these topologies are quite distinct. The most obvious example of this idea of different logical topologies for each layer is that there is no *a priori* reason to think that a website's links to other sites have anything to do with the underlying wire connections that make up the physical network infrastructure, yet both graphs have, at times, been referred to as "the Internet graph"[72]. Thus, in an effort do be unambiguous, we will eschew the term completely, and explicitly name each graph at each layer.

According to the OSI 7-layer model of networking, the layers of a network are, from top to bottom: application, presentation, session, transport, network, data-link, and physical[79]. We will describe the topology and past work for a few of these layers. In this dissertation, we primarily wish to analyze the evolution of the interconnection patterns of the infrastructure itself, primarily at a social level rather than a technical one.

At the very highest layer, the application layer, people have studied the graph of hyperlinks on the world wide web — the **web graph**[9, 10, 16, 22, 43, 56] — as well as the interconnection patterns of different peer-to-peer networks and social interconnection websites[48, 50, 57, 58, 64]. Application-level graphs, however, are usually too high a level for our purposes. We are primarily concerned with the Internet topology at an economic and infrastructure level, and not in the

characteristics of particular applications that use said infrastructure. Applications which create graphs are usually written in a way that largely ignores the underlying physical topology, which means that their graphs are largely independent of the underlying hardware. Some topology-aware applications do exist — but they make up a small minority of deployed applications.

The presentation, session, and transport layers have little to do with topology, and much more to do with data reassembly and data structure, so we pass over them to the next topological layer, the network layer. To communicate across the Internet, a device must use **the Internet Protocol** (**IP**). In IP, computers with an **IP address** send small **packets** of data to each other. The destination of each packet is carried along with the packet itself, and it is forwarded through the Internet from point to point via **routers**. The main job that a router performs is to choose which adjacent device should receive a given packet, i.e. what the next hop should be. Thus, a path from one IP address on the Internet to another consists of a path that begins and ends with either a computer or a router, but consists solely of routers in the middle. This layer of topology is therefore called the **router graph** or sometimes the **IP graph**, and even, yes, the Internet graph. This path is exactly what is measured via the `traceroute` application, which uses varying time-to-live values on packets in an effort to deduce, from router-originating error messages, the paths a packet takes from the measurement point to its destination.

For a sample of this program's output, in Figure 1 we show a traceroute from the University of Oregon to the Royal Institute of Technology in Stockholm taken on April 16, 2008. In this measurement, we can see that there are 15 hops taken from the computer on the University of Oregon network in the router graph, indicating the presence of at least 14 routers between the endpoints in the two universities. This layer, the router graph, is very close to what we might be looking for when we say "the Internet graph", but it is not quite right. In particular, notice

```
traceroute to lvs-vip-1.sys.kth.se (130.237.32.107), 64 hops max, 40 byte packets
 1  vl-60.uonet1-gw.uoregon.edu (128.223.60.2)  1 ms  0 ms  0 ms
 2  0.ge-0-1-0.uonet8-gw.uoregon.edu (128.223.3.8)  0 ms  0 ms  0 ms
 3  vl-105.ge-2-0-0.core0-gw.pdx.oregon-gigapop.net (198.32.165.89)  3 ms  3 ms  2 ms
 4  vl-101.xe.pdx-losa.oregon-gigapop.net (198.32.165.66)  26 ms  25 ms  24 ms
 5  so-0-0-0.0.rtr.hous.net.internet2.edu (64.57.28.45)  57 ms  56 ms  56 ms
 6  so-4-0-0.0.rtr.atla.net.internet2.edu (64.57.28.42)  80 ms  80 ms  80 ms
 7  ge-0-1-0.10.nycmng.abilene.ucaid.edu (64.57.28.7)  93 ms  93 ms  93 ms
 8  abilene-wash.rt1.fra.de.geant2.net (62.40.125.17)  197 ms  197 ms  197 ms
 9  so-6-0-0.rt2.cop.dk.geant2.net (62.40.112.50)  203 ms  203 ms  203 ms
10  nordunet-gw.rt2.cop.dk.geant2.net (62.40.124.46)  203 ms  203 ms  203 ms
11  se-fre.nordu.net (193.10.68.117)  213 ms  213 ms  213 ms
12  c1sth-so-4-1-0.sunet.se (193.10.252.146)  213 ms  213 ms  213 ms
13  a1sth-kth.sunet.se (193.11.0.194)  213 ms  217 ms  213 ms
14  cn6-a1g-p2p.gw.kth.se (130.237.0.2)  213 ms  217 ms  213 ms
15  lvs-vip-1.sys.kth.se (130.237.32.107)  217 ms  213 ms  217 ms
```

FIGURE 1. The output of `traceroute` from a computer on the University of Oregon network (`128.223.60.112`) to the Royal Institute of Technology in Stockholm (`www.kth.se`) taken on April 16, 2008

that the names of the routers suggest that the conversation traveled through quite a few autonomous systems. Going by the router names alone (column 2 of the figure) we find evidence of the following networks: `uoregon.edu`, `oregon-gigapop.net`, `internet2.edu`, `ucaid.edu`, `geant2.net`, `nordu.net`, `sunet.se`, and `kth.se`. If we investigate further, we find that `ucaid.edu` and `internet2.edu` are actually the same organization, so we will treat these two as a single network. Excitingly, despite the fact that the University of Oregon has no direct contract to exchange traffic with the Swedish Royal Institute, our conversation made its way across seven networks and arrived at its destination, and five of those networks were neither the message recipient for the message sender. Unfortunately, despite the fact that our path consisted of just eight domains of control, we can see that the measured path is fifteen hops long. Thus, if we want to concentrate as much as possible on the inter-network links, this measurement contains extraneous information regarding the internal topology of the networks under consideration.

The main priority of the designers of the Internet was inter-networking, and the router graph contains many intra-network edges. In doing so, the router graph

partially exposes how a given network is implemented, and analysis of this graph may miss the forest for the trees with respect to inter-networking. Too much data about intra-network connections can easily drown out the essential information about inter-network connections. Thus, the router graph is very close to what we would like to analyze, but is not quite right.

Moving further down the stack of networking protocols, we find that things become even less helpful. A complete data-link layer graph of the Internet makes very little sense, because a variety of data-link protocols are used to forward traffic, and the method (and utility) of combining such diverse links into any organized framework is entirely unknown. The layer beneath the data-link layer is the physical wires that packets traverse. A single hop on the IP layer might cross a virtual private network (VPN), ATM network, or any of a number of other technologies that may logically act as a single link in the network, despite consisting of multiple distinct physical links. In the router graph these network elements would be reflected as a single edge, when in actuality it could be an arbitrary number of hops. The most fundamental graph of the Internet could then be considered the **wire graph** — every network element capable of splitting, collating, switching, routing, generating, or receiving traffic would be considered a node, and two nodes are connected if there is a physical wire from one network element to the other. How then, does this graph relate to the router graph? What, if anything, can we say about this graph from available data? Almost no research has been done on the wire graph, and it is not clear that there exist any good methods of discovery at this layer, but it is important to note all layers for completeness. The physical layer initially seems promising, but a complete reckoning of every wire on the Internet is both a totally impossible request due to reasons of data unavailability, and this layer also entirely misses the forest for the trees with respect to inter-network connections. Thus, the router graph remains our best candidate for analysis.

Even better, however, is the application-level topology used by IP routers to perform the necessary path discovery to set up the router graph. This graph is called the **autonomous system graph** or **AS graph**. In the AS graph, each autonomous system is a single vertex, and connections between vertices indicate one or more connections between the two ASes along which traffic may flow. By collapsing all internal connections of a single network into a single vertex, the AS graph forms a **graph minor**[6] of the router graph, at least in theory. In practice, there exist a few network deployment technologies and techniques which can create a false picture of connectivity in the AS graph. Dealing with these special (and rare) cases is discussed in Chapter III.

The AS graph is set up by the **Border Gateway Protocol** (**BGP**), and, in the AS graph, the inter-AS connection methods are glossed over. If one AS connects to another in any way, over any link, then they are deemed to be connected in the AS graph. The AS graph is an expression of network relations, which makes it an economic and political graph of the Internet. The AS graph was famously studied by Faloutsos et al. [30], who concluded that the AS graph was a power law graph and was therefore vulnerable to specific kinds of targeted attack. Their work on the subject reverberated across the networking community, and was reinforced by a followup study four years later that reported very similar phenomena[62]. It is important to note, however, that there is a strong systemic bias to the data that they used, and later research by Doyle et al.[29] showed that treating the Internet as just a normal power law graph may be a fundamentally flawed approach. Indeed, much power-law research is coming under attack from different angles as people are accused of fitting popular theories to their data instead of looking for the correct theories[63], and the future of power law research is moving away from raw

[6] A graph minor takes a graph $G = (V, E)$ and makes a new smaller graph, by either removing an edge, or by taking two adjacent vertices $(u, v) \in E$ and combining them into a single vertex that is connected to any and all vertices that either v or u was connected to in the original graph.

observation, and towards validation of the observations via an underlying model[51]. The relationship between the router graph and the AS graph was briefly touched upon by Broido et al. in [17] where they note that the AS graph is a graph of economic relations, and not necessarily a graph of packet paths. The issue of how we may infer missing links in the router graph from real-time BGP data was tackled by Andersen et al.[5] and they came to the conclusion that BGP dynamics do reflect the underlying AS topology, and that hidden relationships in the AS graph may be deduced from the arrival times of BGP messages — an artifact of IP traffic and router-graph topology.

Recent work by Achlioptas et al.[1] suggests strong similarities between the fundamental problems in the AS-graph domain and router-graph domain. A nice example of these similarities is that our concern for completeness of the AS graph has been dealt with by Lakhina et al. for the IP graph[45]. Because BGP shortest-path routing determines, to a large extent, what route an IP packet will take, the underlying graphs at each layer are related at a very deep level. This inference process between layers should not be taken too far, however, as Hyun et al. compared traceroute results with AS path results and found that there were significant incongruities[40].

Several other studies have conflated the AS graph and the router graph, and assumed that each link in the AS graph represented a single link in the router graph[2]. The best study of the forces that drive the construction of the router graph is a series of papers by Li, Alderson, Willinger, and Doyle[47, 3] which show that issues of technology cost and capability have graph theoretic implications for the degree distribution of the router graph. They validate these conclusions by studying the router graph of a large ISP. We note, however, that technological concerns are unlikely to directly affect the AS graph — only the router graph.

If we take as our axiom that the key feature that distinguishes the Internet from other networks is the act of inter-networking, then the layer that represents each individual network as a vertex and inter-network links as edges seems the most directly relevant. Thus, our primary object of study in this dissertation will be the AS graph.

2.3 The Autonomous System (AS) Graph

In the autonomous system graph, every AS is a single vertex, and all intra-network topology is abstracted away. Instead, the AS graph reflects only the inter-network topology. Turning back to the example in Figure 1, we can create a new example where instead of looking at the routers involved, we look at the domains of control. In that case, we note that the conversation passed through seven domains of control over the course of 15 hops. In order, they were: the University of Oregon, Oregon Gigapop, Internet2, Geant2, NORDUnet, SUNET, and finally the Royal Institute of Technology in Stockholm, Sweden. Any of these autonomous systems had the capability of altering the data "in flight" or censoring the traceroute altogether. Therefore, while there were 15 hops from the point of view of an IP packet, from a policy point of view, the path contained only seven distinct entities. In the AS graph, this path actually is a path of length 6, just like we want!

In order to meet its design goals[23], the Internet uses the Internet Protocol (**IP**), and every publicly-accessible node of the network has its own globally unique address. In version 4 of IP (**IPv4**) the address takes the form of four numbers that can range from 0 through 255, separated by dots. For example, the computer in my office has the IPv4 address `128.223.6.141`. In version 6 of IP (**IPv6**) this address takes the form of eight numbers that can range from 0 to 65535, traditionally written in hexadecimal, and separated by colons. The same computer as mentioned

previously has the IPv6 address `2001:468:d01:6:250:fcff:fe9c:da44`. The Internet was built using IPv4, but the demand for addresses has outstripped the supply, and so IPv6 was designed and introduced so that the Internet could continue to grow[12].

When people connect to an internet host, they generally don't use these addresses explicitly, however. Usually people use a name like `www.yahoo.com` or `ix.cs.uoregon.edu`, which is then translated from a name into an IP address by a local computer running the domain name service (**DNS**). Once this translation takes place, the end host, now armed with an address instead of name, proceeds to use the address for all communication. Thus, the issue of mapping names to addresses, while crucial to Internet reliability, is largely orthogonal to topology issues.

When every host has a globally unique address, the whole of a router's job consists of receiving a packet, and then sending it out in the right direction. In an effort to allow routers to know what the right direction is, most routers use version 4 of the Border Gateway Protocol (**BGP4**). In BGP, each router informs all of its neighbor routers of two things: what contiguous chunks of addresses (**netblocks**) the announcing router controls, and all of the best-paths that it knows to all of the other netblocks it has heard of. It initially seems miraculous that, with just local incremental communication and no initial knowledge, every router could learn the best path to take. The crucial insight into understanding the system is that each router begins not with no information, but with one piece of information — it knows how to get to itself and to the netblocks that it controls[7]. It then announces that (trivial) route to all of its neighbors, and its neighbors do the same to it. In this process, it then learns all the routes to all the netblocks controlled by its neighbors. If we view the protocol as proceeding in lockstep, then in the next phase

[7]This insight was garnered from Radia Perlman at a workshop given by her at NANOG 34

each router learns all of the routes that are three hops long, then all the four hop routes, and so on.

When an AS propagates a path to one of its neighbors, it appends its own Autonomous System Number (**ASN**) to the path. In this way, the length of the path being announced grows as it propagates away from the original announcer. These announcements then propagate through the system until eventually every router knows the best available path to every other router's address space.

Or at least, that's the theory, but it is not entirely accurate. BGP is not designed to settle on the best path to every router, but instead to determine the best path to every autonomous system, and ASes often have many routers. The BGP-speaking routers are installed on the internet/intranet border of the AS's network, and they act as a gateway that allows traffic either into the network, or immediately passes it along an inter-AS link[8]. An AS may have many such routers, as an AS may have many connections from its internal network to the Internet. Furthermore, autonomous systems have an incentive to filter the information that they pass on to their neighbors in an effort to minimize their own expenses. Gao[35] used these incentives to develop a model of path propagation and traffic flow that reflects the incentives and likely resulting policies of the autonomous systems doing the propagating.

The **valley-free model** of internet routing is the most prominent and successful attempt to take these incentives into account. The **valley-free model** was first introduced by Gao[34], and that paper defined three kinds of relations:

1. **sibling-sibling**, where two autonomous systems freely exchange all traffic. This is a relatively rare form of interconnection, and generally implies that the two autonomous systems are actually operated by the same entity, and just happen to have different AS numbers for historical or operational reasons.

[8]Hence the name Border Gateway Protocol

2. **peering**, where two autonomous systems freely exchange traffic destined for each other or their customers.

3. **customer-provider**, where one autonomous system, the customer, pays the other autonomous system, the provider, to move the customers traffic to anywhere that the provider can reach. The provider, however, can only use the customer's link to reach the customer or one of the customer's customers or siblings.

In the valley-free model, the only paths that get used or propagated are those which conform to the following specification: zero or more links from customer to provider, followed by at most one peering link, followed by zero or more links from provider to customer. The sibling-sibling links may act as any kind of link and appear at any point in the path.

This model reflects the incentives of ASes coupled with the obligations that come from being a provider. In particular, a customer has no motivation to perform volunteer work for their providers by carrying traffic from one of their providers to another of their providers.

As an example, the University of Oregon has as its providers both Qwest Communications and The Oregon Gigapop. Despite the fact that the University of Oregon could carry traffic from one to the other, the university routers are set up to avoid forwarding any traffic from Qwest to Oregon Gigapop and vice-versa. If those two entities want to communicate with each other across University of Oregon equipment, then at least one of them must pay the University of Oregon or one of the university's customers. Customers of the University of Oregon, however, have access to both Oregon Gigapop and Qwest, because a customer-provider contract obligates the provider to provide service to all reachable networks. Peers of the

University of Oregon can access the university and all of the university's customers, but not Qwest or Oregon Gigapop, as they are the university's providers.

The model is called "valley-free" because, if we consider customer-provider links to be directional from customer to provider (i.e.. they follow the money), then we note that we start out going strictly up, across at most one peering link, and then go strictly down. Once a path has leveled-off (gone across a peering link) or started to go down (gone from provider to customer) it must go strictly down, against the flow of money.

Once a path has gone down, it can never go up again, and since the only way to make a valley is to go down then up, valleys are exactly what our model forbids. Therefore, we say that subject to valley-free constraints, every AS seeks to make its communication paths as short as possible, and seeks to find a path to every other AS on the the Internet. When we analyze the AS graph, we build it from available routing data. Because routers do not store edge type and direction, what we measure is the **undirected incomplete AS graph**. The object we would like to measure is the **complete directed AS graph**, which is the graph from which our measurements come. Techniques for determining edge direction and for assessing incompleteness are the subject of Chapter V.

With our understanding that the Internet is an international decentralized network of networks, and armed with a deeper knowledge of how traffic flows, we can now discuss how we might analyze our large graph of Internet AS-graph topology measurements.

2.4 Large Graphs

Large graphs which derive from natural processes are, in general, a different type of object than the graphs typically measured and analyzed by graph theoreticians.

In his survey on this subject[49], Lovász notes that, if we ask the simple question "Does the graph have an odd number of vertices?", we may still find ourselves in a pickle:

> This is a very basic property of a graph in the classical setting. For example, it is one of the first theorems or exercises in a graph theory course that every graph with an odd number of nodes has a node with even degree.
>
> But for the internet, this question is clearly nonsense. Not only does the number of nodes change all the time, with devices going online and offline, but even if we fix a specific time like 12:00am today, it is not well-defined: there will be computers just in the process of booting up, breaking down etc.[9]

Lovász describes three questions that might be asked of a large graph:

1. Does the graph have an odd number of edges?
2. What is the average degree of the graph?
3. Is the graph connected?

The first is, as we have discussed, somewhat ill-defined. The second, however, is more natural to the situation. The average degree we find may only be accurate to within a certain measurement error of the underlying graph, but, for a large graph, the addition of one or two edges to the graph will not significantly change the average degree. Thus, for a given set of measurements, we can be confident of our estimation of average degree even if our measurement are slightly inaccurate, as measurements often are.

[9]From this and other context in the paper, it seems he is talking about the IP graph of the Internet as "the internet". Note that his concerns also apply to the AS graph, as new ISPs are constantly coming online and falling offline. Also, the lack of capitalization is [sic].

The third question, "Is the graph connected?", is more tricky. This provides us with a good example of a kind of category error that can emerge when we look at large graphs. On the AS graph, the answer can be argued to be each of "yes", "no", and "What are you getting at?" If we were to simply look at our measurements, we would see that the resulting graph is connected. This, however, is an artifact of our measurements. The only way we can collect measurements is if the router is connected to the Internet, and the only way another AS would show up in the measurements is if they were also connected to the Internet. On the other hand, because the AS graph is constantly changing and ISPs are constantly replacing routers and having systems break, we can with confidence state that, at any given time, the AS graph is almost certainly not a connected graph — the odds are high that at any given time, at least one ISP is experiencing severe technical difficulties. This "no" answer is no more useful than the "yes" answer, however. The most useful response is "What are getting at?"

Because, at any given time, the AS graph is highly likely to be a disconnected graph, we might view the AS graph as some sort of failure! The goal of the Internet was, after all, interconnection, and now we see that its connectivity graph is almost certainly not a connected graph. But when network operators ask whether they are connected to the Internet, they often consider the network to be working well if, say, 99% of the Internet is reachable at any given time. In this case, the operator would almost certainly consider their network to be connected, and the Internet as a whole to be working well. The graph theoretic idea of connectedness does not map to a useful concept for Internet operators, but network operators do care greatly about connectivity and related concepts. If we want to investigate this property, therefore, we must devise a graph theoretic metric that better maps to the idea of connectedness as it is used in practice. Before we can devise analysis techniques, however, we will look more deeply at the range of graph analysis techniques.

2.5 Analysis Methods for Large, Dynamic Graphs

As we saw earlier, large dynamic graphs that are the result of evolutionary processes are different in kind from the graphs traditionally analyzed in graph theory. The fact that there is an ambiguity in something as fundamental as the number of vertices and edges means that any metric which can radically change with the introduction or removal of a single edge is almost certainly impossible to measure with any accuracy.

We will classify analysis techniques by the result of their analysis. The most general method of graph analysis is to consider a function f which takes as its input a graph G, and outputs some item. Graph analysis functions have, as their domain, some subset of the set of all graphs, but the type of objects they may return are extremely diverse. There are useful graph analysis functions that return numbers (integer and real), but other functions of active use and interest may return tuples, vectors, distributions, matrices, functions, graphs, or sets of any of these.

Before we classify analysis methods, we should answer a key question: Why analyze at all? The alternative to analysis is presentation of the raw data[10]. If the AS graph consisted of 4 ASes with 6 links between them, then analysis of this graph would be foolish — we should just show the data by drawing and labeling the graph. In this case, the AS graph would be small enough to "fit in a person's head". From this, it follows that analysis need only be performed in situations where the object of study is too large for a person to grasp the whole of it at once.

Thus, graph analysis should be understood as, at least in part, belonging to the domain of human computer interaction. The result of the analysis function should, in all cases, be "simpler" than the input graph, lest our analysis reduce

[10]Although if we add "tables" and "visualizations" to our output types, then data presentation arguably falls under our model as well, but we restrict our purview to functions which emit mathematical objects.

understanding rather than increase it. We leave "simpler" in quotes because it is not clear that this term has a rigorous definition that is usable for us. However, we should not lose sight of the fact that, even if we cannot quantify exactly how, graph analysis should explicate and explain some graph which is too large to look at and understand.

The other thing that graph analysis can do is to guide some other optimization or allocation process. This is a more traditional area for computer science, and many examples may be seen in Chapter IV. In these problems, we are given a graph and some function which we would like to either minimize or maximize. This is a form of graph analysis in which the goal is already known, e.g. given a graph, find a vertex cover of minimum size. In this case, what we might think of as the "goal" of the whole process is not so much an understanding of the graph, but instead an understanding of how to allocate a particular resource on this graph. We will consider some of these methods as well, but a full listing of them is beyond the scope of this work. In particular, we will limit ourselves to discussing only those optimization and allocation properties which have already been shown to be of interest on the AS graph. Now, finally, we begin our enumeration of analysis technique types.

2.5.1 Analysis Techniques which Result in a Single Number

When our analysis function results in a number, then we might consider ourselves lucky! Numbers can be subjected to statistical analysis and placed in a chart or table with relative ease. If the aspect of interest of the AS graph may be expressed as a single number for a given graph, then it is also easy to examine the evolution of this property over time by creating a chart with time on the x-axis and the property of interest on the y-axis. This ease of display and understanding means that when a graph property of interest can be expressed as a single number, it

should be expressed as a single number. We will use this principle in the design of new analysis techniques. If an analysis technique results in a number, then we will call it a **graph metric** or **graph statistic**.

Graph metrics may be extremely simple. Counting the number of vertices or the number of edges are both graph metrics, for example. Only slightly more complicated is the calculation of the average degree, calculated by dividing the number of edges by the number of vertices. There are also much more sophisticated techniques, such as measurements of maximum flows and cuts in a graph. We discuss more of these techniques later, but an important insight for now is that, whenever a good one is available, we prefer a graph statistic over every other analysis technique.

2.5.2 Analysis Techniques which Result in a Distribution

Other graph analysis techniques result in a distribution as their final output. In these analysis methods, the result is not a single number, but a distribution over some domain of numbers. The best example of this is the idea of a **degree distribution**. In a degree distribution, the number of vertices with a given degree (or out-degree or in-degree in a directed graphs) is counted for every degree which exists in the graph. Graph analyses that result in a distribution are useful because there are many tools from statistics and other fields which allow us to manipulate and further analyze distributions of numbers.

Furthermore, many distributions are known to frequently occur and may be characterized as a known function with constant that are specific to a given situation. The best example of this is the normal distribution, which is completely characterized by two numbers: the mean and the variance. In this case, when a distribution may be regarded as an instance of a distribution with a known function, we can reduce the whole of the distribution to a just a few numbers, as in

the example of the normal distribution. In this way, we can often turn an analysis method that results in a distribution into a graph statistic.

2.5.3 Analysis Techniques which Result in a Function

Many analysis techniques result in a labeling of vertices and/or edges. We can then think of the result of our analysis being a mapping, or function, between elements of the graph and items from the domain of all possible labels. If the labels are taken from the set of real numbers, then we can also construct a distribution of labels, and possibly even turn the whole thing into a graph statistic as described earlier.

A nice example of this is the idea of **centrality**. There are many methods which try to formalize the idea of a vertex being central to a structure. Many centrality measures attempt to discover how important a vertex is to the graph as a whole by assigning a calculated number to each vertex. This number corresponds to the centrality of the vertex, with, in general, higher numbers being "more central". If our goal is to analyze a single vertex, then this can be quite helpful. Impact factors for journals and conferences are a form of centrality measure, and they help researchers decide where to submit their work. If our goal is to analyze the graph as whole, however, then quite often the immediate next step is to look at the distribution of centrality over the whole graph. In this way, we convert the centrality function into a centrality distribution. If we are lucky, then we can subsequently convert the distribution into a graph metric.

2.5.4 Other Analysis Techniques

Other analysis techniques include investigating a graph's adjacency matrix by analyzing its eigenvalues and eigenvectors, using heuristics to find clusters in the

graph, or the creation of another graph which is more easily understood that still contains the structures and properties of interest. There are an almost infinite number of these techniques, so when we finally analyze the AS graph, we will not enumerate all possible analysis techniques. Instead, we will restrict ourselves to re-performing analyses from extant literature, and to the development of just a few new metrics in an effort to formalize aspects of the AS graph structure which may be of interest.

We talk more on this in Chapter VI, where we discuss some old metrics and develop new ones in an attempt to better match the AS graph properties of interest for Internet stakeholders. We end this chapter by discussing an example property of interest for Internet stakeholders: network neutrality and traffic-flow centrality.

2.6 What Is Network Neutrality?

Network neutrality is a tricky concept. Much like the term "Internet", "network neutrality" means many things to many people. Tim Berners Lee, credited with inventing the world wide web, defines it like so:

> If I pay to connect to the Net with a certain quality of service, and you pay to connect with that or greater quality of service, then we can communicate at that level.[8]

Many in the networking community view the term network neutrality as just another name for undifferentiated services, and there is broad confusion about the term among advocates on both sides of the issue. Pro-network-neutrality arguments often bring up older concepts of "common carriage" and complicated metaphors, coupled with doom-laden scenarios of what "they" might do to the Internet if the desired legislation is not forthcoming. The other side is rhetorically no better, as opponents trot out horror scenarios wherein misguided legislation destroys the Internet or prevents its continued technological evolution.

Let us attempt to slice this Gordian knot by, initially, avoiding any definition of network neutrality at all. Instead, we examine what the effects of non-neutral behavior might be. In particular, non-neutral behavior benefits the single party performing the actions, but degrades the Internet experience for others, leading to an outcome that is overall negative. If the Internet were a truly open, fair, free, and undistorted market, then we might imagine this as being impossible. If a single party attempts to degrade everyone else's connections, then that party would simply be routed around. Thus, we find that the fear of non-neutral behavior translates exactly to a fear of monopoly or oligopoly control of a shared resource. Instantly, the folly of attempting to enumerate what sort of behavior might be considered "bad" becomes apparent: There are many forms of bad behavior, and most of them are only dangerous if, for some reason, there do not exist alternate communication paths that an AS might use to avoid the non-neutral AS. We will say more about this in Chapter VI, but for now it is best to roughly draw an equality sign between the growing call for network neutrality legislation and a growing worry of monopolistic behavior by a large or well-placed AS, or a cabal of ASes. In this way, we can translate the language of one domain into that of another. When speaking of graph theory, we can refer to centrality measurements; When speaking of networking, we talk of the ability to manipulate traffic flows; When speaking of the economics of networking, we use the language of network neutrality and market power. A key insight is that, to a large degree, these terms from different disciplines and domains are all referring to the same underlying concepts.

2.7 Summary

The Internet is the biggest internet yet developed. It spans the globe and links billions of people. Yet there is lots of confusion about what it is. There is a vague

notion that it has a topology — certainly there are lots of wires and routers — but looking for "the Internet graph" turns out to be a snipe hunt. We looked at the various network layers and came to the conclusion that if we are concerned with economics and policy, rather than more traditional networking issues of fault-tolerance and bandwidth, then we should look at not the wire graph or even the router/IP graph, but instead the AS graph.

The AS graph is set up by routers exchanging information according to specific policies that are the results of the local application of the incentives of each AS. These local incentives result in a characterizable global structure called "valley-free routing". The AS graph is a large dynamic graph, and such structures pose special problems for more traditional graph-theoretic analysis. Many traditional graph theoretic questions become irrelevant in the face of even a small amount of dynamism, and so we are forced to "go back to the drawing board" and develop measurements and metrics that are robust to dynamism in various ways.

We bootstrap this development process by attempting to articulate a particular property of the AS graph that is of interest to many Internet stakeholders: monopoly and oligopoly control of the Internet. A monopoly control of the routing system could potentially enrich the monopolist at great cost to the utility of the system for everyone else, and worries about this happening are one of the root causes for the ongoing debate over network neutrality.

However, we are a long way from further formalization of the idea of network neutrality. Before we can analyze the AS graph for network neutrality concerns, we must find sources of data, process it into a usable form, vet the data, develop techniques for AS graph reconstruction, and formally define those concerns. In the next chapter, we discuss data provenance, processing, and quality.

CHAPTER III

DATA SOURCES AND PROCESSING

In which we discuss the sources of data, and deal with the problems of parsing the data into a usable format

We have said that we would like to measure the AS graph, and in this chapter we discuss where AS graph data may be found so that we can construct it. Before we go on, however, let us define some new terms. Most historic router data lacks edge-types, and is incomplete. Therefore, we will call the raw result of our data the **undirected measured AS graph**. If we can figure out the edge types, which we traditionally modeled as directed edges, the we get the **directed measured AS graph**. This, let us remind ourselves, is not what we want. We would like the **directed measured complete AS graph**, or just the **complete AS graph**, but unfortunately, in order to measure that graph, we would need to have every operator of every router on the Internet donate their data to us — a gross improbability going forwards, and a total impossibility with respect to past data. This chapter concerns itself with the problem of taking the available historic data and constructing the directed measured AS graph. We then spend the subsequent two chapters investigating the completeness of our measurements.

3.1 Data Sources

In 1997, in order to serve the needs of the network operations community, the Route Views project began publicly archiving a daily snapshot of their router's routing tables. This proved to be of great utility and, over time, more and more ASes decided to contribute data in an effort to debug ongoing Internet problems. It was so successful that other data-gathering efforts appeared on the scene in order to collect similar data from other sources who, for whatever reason, preferred to share with these newer repositories instead of Route Views[70]. The largest of these parallel efforts is run by RIPE NCC (Réseaux IP Européens Network Coordination Center)[68]. All of these data archives have been available online for, in some cases, up to a decade, but no longitudinal study has yet been performed. However, with an eye towards future analysis, much of the modern Route Views data set has been preprocessed by The Cooperative Association for Internet Data Analysis (CAIDA)[66] into a more manageable form. We deal with the implications of this preprocessing in Section 3.5. They did not preprocess all the data from all sources, however, so if we would like a more complete picture, then we must be able to parse the data ourselves.

3.2 Parsing Historical Routing Data

Once we know where the data is, we must parse it to get useful information out of it. In one of life's little ironies, everybody who knows the format of the data considers it too obvious to document more than cursorily, although it is often opaque to an outside observer. In this section we will unpack the three different styles of data that are available, and map out the "gotchas" of each format. In doing so, we come up with a convincing argument for structured plaintext for long-term data storage — it's a little less space-efficient, but it can always be parsed

and manipulated with available tools. Unfortunately, in the case of routing data, we have to choose between poorly-structured text and a binary format parseable using programs that have not been updated in many years.

Because the oldest data is in text format only, we deal with the text format first.

3.2.1 Parsing Textual Routing Data

The old textual routing data was generated by a script that would collect the result of telnetting to the Route Views router and typing the command `show ip bgp`, which is the command to show all BGP routes for all blocks of IP addresses. This script was brittle and the routers went down occasionally, but it was the only source of data for years.

This text data was designed to be read by humans, and the output of the script also contains extraneous information. Representative samples of the textual output can be seen in Figure 2 and Figure 3[1]. These tables indicate what each AS has chosen as the best path from itself to every other netblock on the Internet. In Figure 2, for example, we can see that to reach any network address that begins with a 3, i.e. is in the netblock 3.0.0.0/8, the router with IP 134.55.24.6 has settled on the path from AS 293 to AS 701 to AS 80 as the best path. Using just this path, we derive a measured AS topology as in Figure 5(A). This is in contrast to the router at 192.121.154.25 which is using the path AS 1755 – AS 701 – AS 80, a path that only differs in the first hop. These two paths taken together imply a network topology as shown in Figure 5(B). If we take all the paths to 3.0.0.0/8 in Figure 2, then we get the topology shown in Figure 5(C).

[1]Both figures come from `http://archive.routeviews.org/oix-route-views/2000.06/oix-full-snapshot-2000-06-11-0840.dat.bz2` — a randomly chosen file from the Route Views collection

```
##################################################################

route-views.oregon-ix.net -- Oregon Exchange BGP Route Viewer

This hardware is part of a grant from Cisco Systems.

This router has several views of the full routing table, including:

    AboveNet      (MAE-WEST)      207.126.96.1    through AS6461
    ANS           (Cleveland)     206.157.77.11   through AS1673
    [ Middle of the list elided due to length --PMB ]
    Zocalo        (Berkeley)      157.22.9.7      through AS715
    UlakNet       (Turkey)        193.140.0.1     through AS8517

This service relies on the fact that ISPs provide their views in a
collaborative spirit, to support one another in debugging
operational internet problems. Each view is the property of each
ISP, and any non-operational use must be coordinated through the
providers via the Route Views administrator. For details on this
process, please see http://www.antc.uoregon.edu/route-views/aup.html.

Please contact dmm@antc.uoregon.edu if you have questions or
comments about this service, its use, or if you might be able to
contribute your view.

##################################################################

route-views.oregon-ix.net>
route-views.oregon-ix.net>term len 0
route-views.oregon-ix.net>show ip bgp
BGP table version is 4502983, local router ID is 198.32.162.100
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  3.0.0.0          134.55.24.6                              0 293 701 80 i
*                   192.121.154.25                           0 1755 701 80 i
*                   204.42.253.253                           0 267 2914 701 80 i
*                   193.0.0.56                               0 3333 1103 6453 701 80 i
*                   129.250.0.1              6               0 2914 701 80 i
*                   4.0.0.2                2105              0 1 701 80 i
*                   205.158.2.126                            0 2828 701 80 i
*                   134.24.127.30           58               0 1740 701 80 i
*                   158.43.206.96                            0 1849 702 701 80 i
*                   129.250.0.3              8               0 2914 701 80 i
*                   193.140.0.1                              0 8517 701 80 i
*>                  12.127.0.249                             0 7018 701 80 i
*                   157.22.9.7                               0 715 701 80 i
*                   204.212.44.129                           0 234 2914 701 80 i
*                   206.220.240.223                          0 10764 5646 1 701 80 i
*                   202.232.1.8                              0 2497 701 80 i
*                   204.70.4.89                              0 3561 701 80 i
*                   144.228.240.93          10               0 1239 701 80 i
*  4.0.0.0          134.55.24.6                              0 293 1 i
*                   192.121.154.25                           0 1755 1 i
*                   204.42.253.253                           0 267 2914 1 i
*                   193.0.0.56                               0 3333 1103 6453 1 i
*>                  4.0.0.2                                  0 1 i
```

FIGURE 2. A representative leading section for textual BGP data

```
*                   158.43.206.96                            0 1849 702 701 816 6407 i
*                   129.250.0.3              8               0 2914 701 6401 6407 i
*                   193.140.0.1                              0 8517 701 816 6407 i
*>                  12.127.0.249                             0 7018 701 6401 6407 i
*                   157.22.9.7                               0 715 701 6401 6407 i
*                   204.212.44.129                           0 234 2914 701 6401 6407 i
*                   144.228.240.93          10               0 1239 701 6401 6407 i
*                   204.70.4.89                              0 3561 701 6401 6407 i
*                   202.232.1.8                              0 2497 701 6401 6407 i
route-views.oregon-ix.net>
```

FIGURE 3. A representative example of how the text data ends

```
*  38.104.1.0/24     147.28.7.2              20       0       0 3130 2914 174 11057 i
*  38.104.104.104/30
                     209.161.175.4            0       0       0 14608 i
*  38.105.1.0/24     81.209.156.1             0       0       0 13237 174 40785 i
```

(A) A netblock with too many characters to fit in the given space. Note the inserted newline.

```
*  20.137.0.0/21     62.72.136.2              0       0       0 5413 1299 3549 2187
7 {4237} i
*  24.209.0.0/18     216.218.252.164          0       0       0 6939 1299 6461 7843
 10796 {11060 12262} i
```

(B) AS path aggregation using AS_SETs

FIGURE 4. Some unexpected cases for parsing text data

This data format does not look too difficult to parse, but any implementor must be wary of several special cases. In particular, note that there are too few text columns in the "network" column to hold a netblock in which all digits are used. By example, this means that 3.0.0.0/8 is not a problem, but 111.222.333.444/24 will not fit in the space provided. In that case, the output continues in the proper column, but on the next line as Figure 4(A).

Further special cases include dealing with AS-paths which contain { and }. These paths indicate that the ASes between the braces are members of an AS_SET, and that they are declining to state what order they appear on the path. An example of this is in Figure 4(B). In the second line of this example, the AS path "6939 1299 6461 7843 10796 {11060 12262}" is attesting that the path is either 6939 1299 6461 7843 10796 11060 12262 or it is 6939 1299 6461 7843 10796 12262 11060, but without further information, we can not determine which it is. However, the first line of Figure 4(B) in unambiguous, despite the AS_SET in the path "5413 1299 3549 2187 7 {4237}", because the AS_SETs in the path are all of cardinality one. When dealing with AS_SETs, the implementor has to decide whether bad

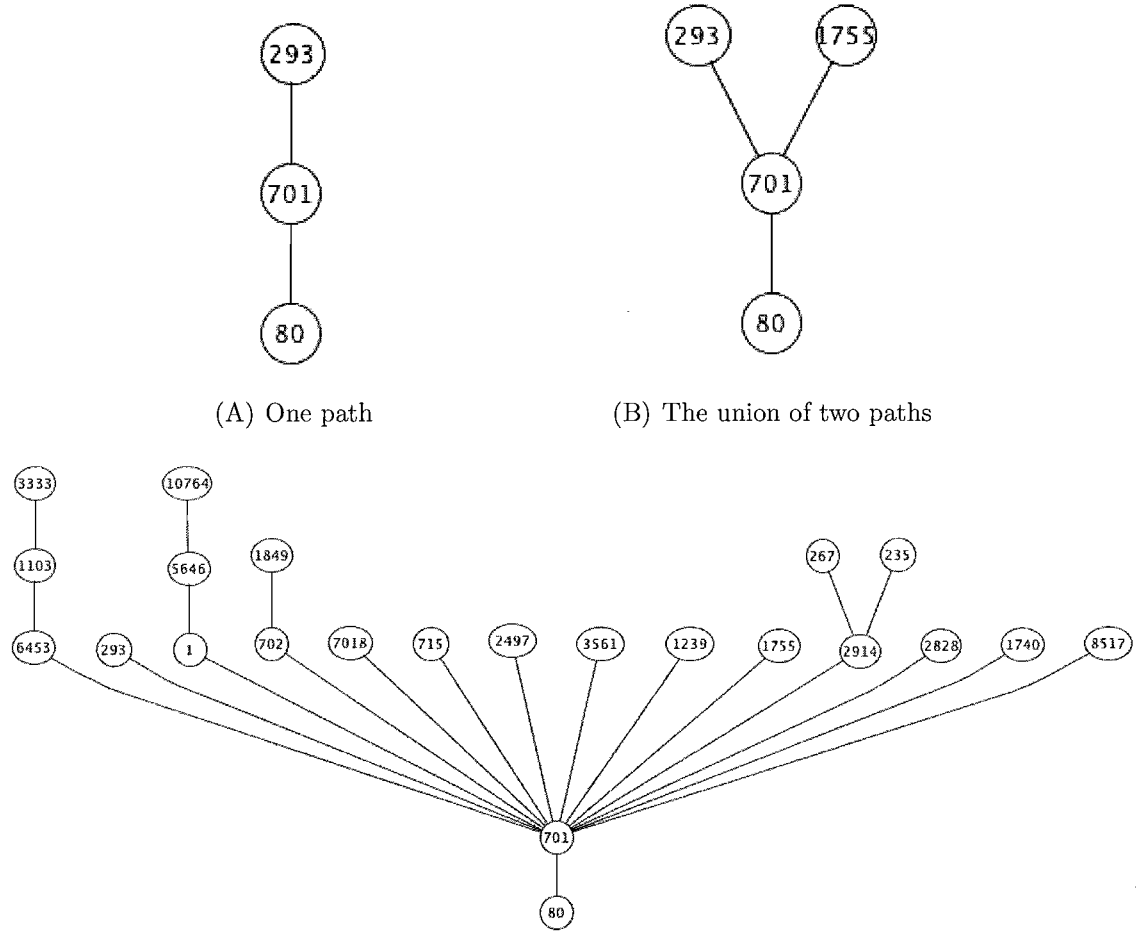
information is better than no information. For our study, as long as all AS_SETs are of cardinality one, then we will save the path. Otherwise the path will be discarded.

Our last special case occurs because the data is taken in a real-world operating environment, and occasionally things were broken. It is not impossible to find data files that end abruptly, due to a buffer in the collection script filling up, or sometimes due to the router containing the data going down as the data was being collected. In this case, we as researchers must make a choice: do we use the incomplete data, or do we throw out the entire day? In this dissertation, we use the incomplete data, with the exception of the last line (if it was a data line). No matter what the choice, however, any parser will need to be able to gracefully handle the data ending abruptly.

3.2.2 Parsing Binary Data

The middle-aged binary data is generated in Zebra format, and is meant to be parsed with the Multi-threaded Routing Toolkit (**MRT**). MRT has not been maintained in some years and has some warts, but it remains an excellent tool for turning routing table data into a convenient text format, suitable for subsequent parsing. It is possible to generate text formats from the middle aged binary blobs, but the most recent data is in a new format (specified by a draft RFC[67]) that currently only one tool — libbgpdump — can parse. C code that links against this library and takes a binary file as input and outputs a `(netblock, path)` pair can be found in Appendix A.

Using these parsing tools for both text and binary files, we now have the ability to extract data in the form of tuples `(date, netblock, path)` from the Route Views and RIPE data archives. How much data is available, and what exactly can we extract from that data?



(A) One path

(B) The union of two paths

(C) All paths to 3.0.0.0/8

FIGURE 5. The effects of gradually adding more paths to our measured AS graph.

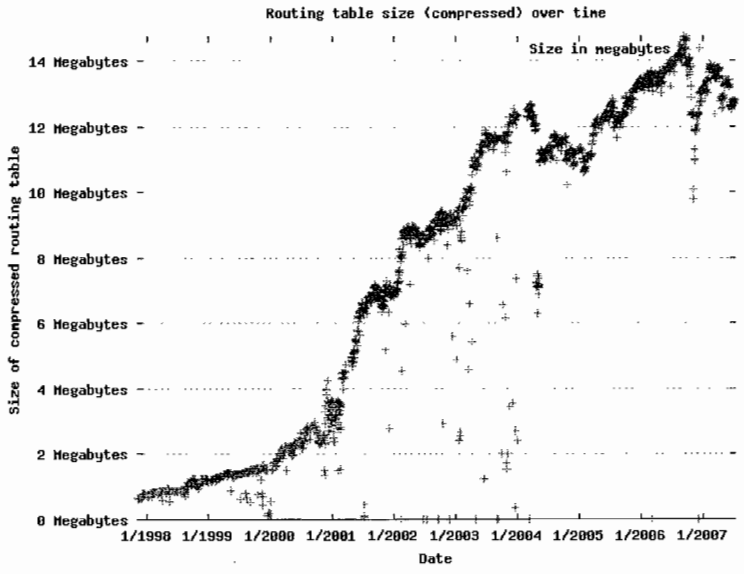


FIGURE 6. The size of the compressed routing table snapshots stored in Route Views

3.3 Data Quantity

Route Views and RIPE have been archiving data for a long time. Route Views was founded first, in 1997, and has been collecting data for over a decade. The growth of data in the Route Views repository can be seen in Figure 6, and that is just for one of several Route Views collection points (albeit the oldest one). In toto, Route Views has more than four terabytes of compressed data, any and all of which could be used in a longitudinal study of the AS graph. RIPE, while younger, has also seen impressive growth, and has a data archive which rivals Route Views in this regard.

All of these data repositories store unparsed compressed data, often in multiple formats. Depending on one's internet connection, CPU speed, etc., it can take weeks just to iterate over the data once. Therefore, locally compiling the available data into a cleaned-up form is essential if our analysis is going to be done more than

once. Since it is the nature of data analysis to lead to further data analysis, a data warehousing step is clearly necessary. Necessary, but not sufficient — in order to build a useful warehouse, we must know what aspects of the data to preserve, and what to throw out as irrelevant. To help analyze this problem, we take a cue from data mining and examine the data cube for AS graph data.

3.4 The Data Cube

A data cube for an n-tuple is a hypercube in which each vertex is labeled with the power-set of the tuple. Two vertices are connected whenever the disjunction of their labels is of size 1. In this way, we can get an overview of all possible combinations of data, and how we might roll-up or drill down into our data set.

We have upwards of 8 Tb of compressed text data, but not all aspects of the data are interesting in all combinations. To make our data warehouse we must break down the available data into its attributes. Every one of our measurements has four attributes: the AS path, the point in the network that is reporting the existence of that AS path, the netblock at the opposite end of the AS path, and the timestamp of the measurement. This naturally leads to a data cube with the dimensions: `(measurement AS, path, IP block, timestamp)`, as displayed in Figure 7. This cube also serves as a good illustration of how little of the potentially interesting data mining space has been explored in the literature.

In particular, there has been no time-series analysis of this data more complex than simply counting the number of elements in the dataset – either IP netblocks over time, or number of measured vertices and edges over time[39]. There have been more in depth analyses conducted on the AS graph at a specific point in time[62, 18], but none of these have grappled with the notion of missing data. So we have longitudinal studies which largely have neglected topology, single-day studies

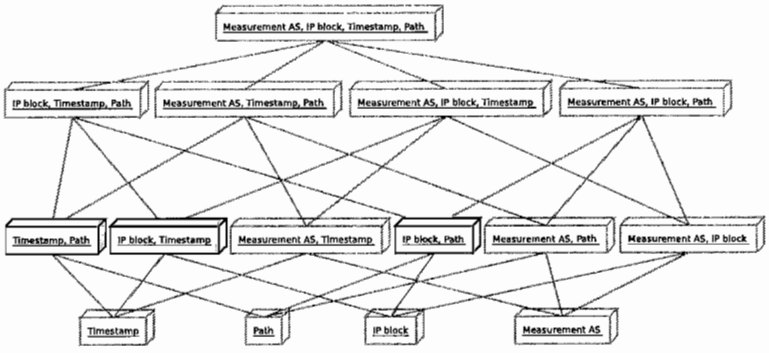


FIGURE 7. The AS Graph Data Cuboid — the three cuboids that have been discussed in other work are highlighted

which have largely neglected evolution, and an almost complete neglect of the problem of missing data.

In this dissertation we are concerned with the complete AS graph, and how it has evolved, so we will ignore all cuboids that ignore the interconnection patterns of the AS graph. In particular, all cuboids not containing the `path` we will leave as future work for others. We expect to be working with many snapshots and not combining multiple days into a single view, so retrieving a single day's topology should be fast if our warehouse is to be of use.

To achieve these goals, we make a directory for each day, and then inside the directory for each day, we put all of the data for a single measurement point in a single file. Doing this will allow us fast access to a given day's data, but will also allow us to add and remove sources one at a time. But before we start "adding sources" to the graph, we need a graph! To facilitate analyses such as this one, CAIDA has preprocessed much of the Route Views repository and used the best-known algorithms to assign direction and type to all of the measured edges.

Their technique uses multiple data stores (WHOIS, BGP data, and text heuristics) as well as an extremely valuable, but unprogrammable operational

knowledge. Rather than (poorly) duplicating their effort, we piggyback on their efforts. Therefore, we primarily concern ourselves with augmenting the derived graphs to include newfound data.

3.5 Preprocessed Route Views Data from CAIDA

With an eye towards future studies such as this one, CAIDA has developed a processed data archive of their own, in which all Route Views data for a given day is combined into a single graph, and the best-known methods for edge-type assignment are then used on the graph. Even better, they validate their use of these methods with actual surveying of AS operators[27]. We use their preprocessed dataset, which has the edge types included, (available online[74]) as a basis for our study.

Their dataset, however, only includes Route Views, so the first order of business is to extend each day's graph with that day's data from RIPE, as well as to double-check that all existing Route Views data has been incorporated. If we take their graph as correct, and also believe their conclusion that peering links made up a large portion of the missing links, then we should ensure that the augmenting links we add to the CAIDA AS graph will, as much as possible both be valley-free and contain a peering link (if possible).

To do this, we develop an algorithm for extending an existing known-correct AS graph. The pseudocode for the method is in Figure 8. In this method, we combine several observations. In particular, usually at most one link of a path is new information. Secondly, most paths are of length 4 or more. Therefore, most paths already have most of their edges already defined. So, although our heuristic seems dangerous at first — we simply add "up" edges until encountering a peering link or a "down" link, and then, if possible, we make the last new "up" edge a peering link

Extend-Directed-AS-Graph $((V, E, S), p)$
// V is the vertex set
// E is the edge set
// S is the set of sibling relationships - note that $S \subseteq E$
// p is the new path

$i \leftarrow 0$
// Make the path go "up" until we are one before a down edge or a peering edge
while $i <$length$(p) - 2$ and $(p_{i+2}, p_{i+1}) \notin E$
    if $(p_i, p_{i+1}) \notin E$
        $E \leftarrow E \cup \{(p_i, p_{i+1})\}$
    $i \leftarrow i + 1$

// Now we've either gone to the end of the path, are one before a down edge, or
// one before a peering edge.
if $(p_{i+2}, p_{i+1}) \in E$ and $(p_{i+1}, p_{i+2}) \notin E$
    // Make it a peering edge, if possible
    if $(p_i, p_{i+1}) \notin E$ and $(p_{i+1}, p_i) \notin E$
        $E \leftarrow E \cup \{(p_i, p_{i+1}), (p_{i+1}, p_i)\}$
elseif $(p_i, p_{i+1}) \notin E$ and (p_{i+1}, p_i)
    $E \leftarrow E \cup \{(p_i, p_{i+1})\}$
$i \leftarrow i + 1$ // Now all missing edges are "down" edges. **while** $i <$length$(p) - 1$
    $E \leftarrow E \cup \{(p_{i+1}, p_i)\}$

FIGURE 8. Our algorithm for extending an existing directed AS graph when a new (undirected) path is discovered.

— in practice this heuristic is highly likely to be correct, because so much of the path is already specified.

Before we go too far, however, let us verify our assumption that most paths are likely to only be missing a few edges. This assertion does mesh well with the intuition developed by Beerliova et al. who found that most graphs, when measured via shortest path trees, have most of their edges discovered with just a few trees rooted at only a few points[7]. It is only the last few edges that get missed that require lots of extra work to find. If that is so, then as long as the number of sources

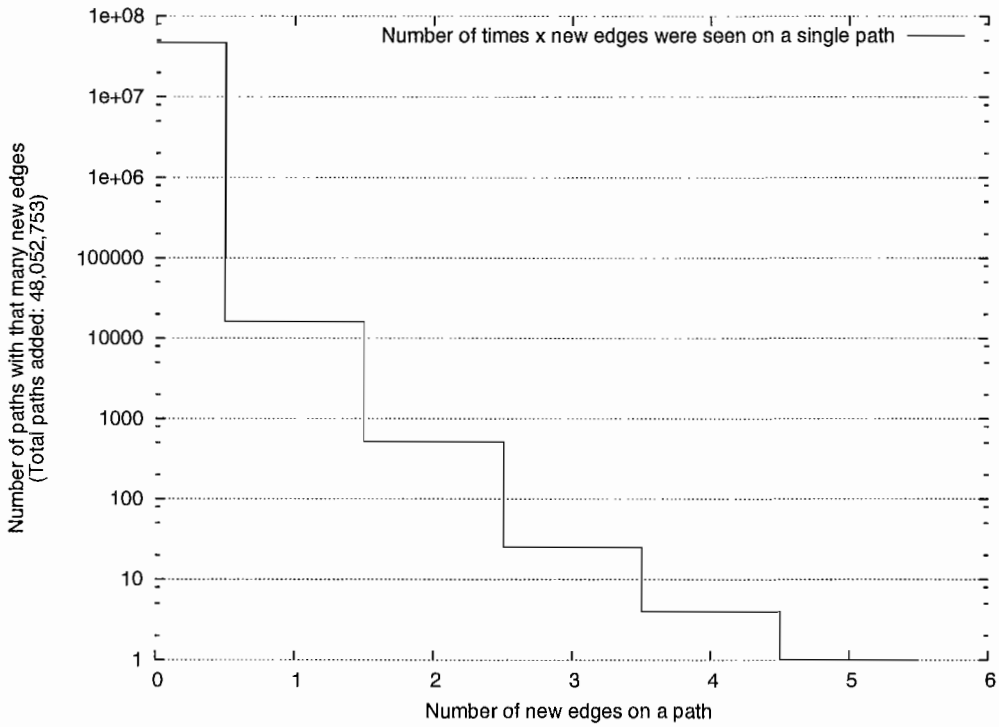


FIGURE 9. A histogram of the number of edges we found per-path that were not in CAIDA's graph but were in our more-complete BGP data.

is not too small, then we should expect to have covered a large percentage of edges already, and so each path shouldn't have too much new information. Or, at least, that's what we hope is the case. Let us choose a random day from the database, and keep track of how many new edges we discover, and how many we discover per path. In Figure 9 we see that, out of 48,052,753 total paths added, almost all of the paths contained no new information. Of those paths that did contain new information, the vast majority were paths containing only a single previously-unknown edge.

Under 1,000 edges are even potentially controversial, out of 137,139 edges in the graph. Using this heuristic is, therefore, unlikely to steer us awry. So, after many steps, we are finally done with data processing and reconstruction.

3.6 Summary

Parsing routing data is one of those things that seems like it should be easier than it turns out to be. There are several formats for the data, and the textual format is surprisingly tricky to parse, because it is not well-documented in a way that helps out graph reconstruction without a lot of domain knowledge. In this chapter we hope to have laid out the parsing process of the text files, and also given a cursory explanation why the standard tools fall down.

Along the way, we discovered that half of the job has already been done for us! Thanks to CAIDA, we can take advantage of pre-processed AS graphs without having to worry about the problem of assigning edge directions for most of the AS graph, thus saving us work later. We established a heuristic for edge direction when adding new paths to the graph, and justified it by seeing how often the assumptions of the algorithm may have become problematic.

Armed with a measured, directed, and partially completed AS graph, we take a brief interlude to consider the problem of AS graph measurement from a more theoretical angle. In particular, now that we have a graph in hand, we start to wonder: what might our graph be missing?

CHAPTER IV

GRAPH COVERING VIA SHORTEST PATHS

Where we devise graph covering problems using shortest paths, prove the $\mathcal{NP}$-completeness of all of them, and then attempt to approximate the answer[1]

Every AS knows, to a first approximation, a shortest path from itself to every other autonomous system. If this is the case, then a question leaps to mind: Given a complete AS graph, what can we know about the set of routers required to completely measure the graph? We define two related graph covering problems motivated by network monitoring. Both problems relate to the idea of graph measurement from a single vertex via shortest paths, and serve as lower and upper bounds on the amount of information that can be gained by measuring a graph via shortest paths. We show that the minimization problem for each model is $\mathcal{NP}$-hard, and we enumerate several classes of graphs for which the minimization problems are efficiently solvable.

For most of this chapter, we consider the easier-to-formulate problem of shortest path trees. Our results for valley-free shortest path trees can be found in Section 4.6.

[1]Early sections of this chapter were previously published as "Graph Covering via Shortest Paths" by Boothe, Dvořák, Farley, and Proskurowski in Congressus Numerantium[11]

4.1 Mathematical Preliminaries

We model a network by a simple graph $G = (V, E)$, where V is a set of vertices representing network devices and E is a set of undirected edges representing links. The edge (u, v) is said to be **incident** with the two vertices u and v. We will use n to signify the number of vertices $|V|$ in a graph G. The **degree** of a vertex v is the number of edges that contain v, i.e., the number of edges incident with v. A **path** of length $k > 0$ is a set of edges $p = \{(v_i, v_{i+1}) | 0 \leq i \leq k, (v_i, v_{i+1}) \in E\}$. We say that a path **connects** vertex v_0 and v_k. A graph G is **connected** if and only if there exists a path that connects every pair of vertices in G. A **connected component** of a graph G is a maximal subgraph of G that is connected; thus, a connected graph has a single connected component. If the removal of some subset of vertices $S \subset V$ renders the resulting graph disconnected, then we say that the set S is a **vertex separator** of G. The **distance** between a pair of vertices u and v, denoted $d(u, v)$, is equal to the minimum length of a path connecting u and v. A **shortest path** between two vertices u and v, is a path connecting u and v having length equal to $d(u, v)$. Given vertex v, for each distinct vertex u, let $P_{v,u}$ be the set of shortest paths that connect v and u. A **cycle** of length $k > 2$ in a graph $G = (V, E)$ is a set of k edges $\{(v_1, v_2), (v_2, v_3) \ldots (v_{k-1}, v_k), (v_k, v_1)\} \subset E$ in which all v_i are distinct. When talking about cycles, unless otherwise specified, we are referring to chord-free cycles, which are cycles in which $(v_i, v_j) \notin E$ unless $j = i + 1 \mod k$. If $G = (V, E)$ has $E = V \times V \setminus \{v \times v | v \in V\}$, then we say that G is a **complete graph**. If G is a complete graph with n vertices, we denote it as K_n, which we also call a **clique** of size n.

In an IP network, we assume each site knows one or more shortest paths to every other site, based upon routing information for disseminating messages in the network. Given the links on the paths, each site knows about the existence of a subset of the links in the network. In the network discovery problem, we establish

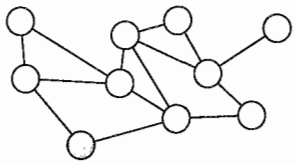
query sites (each query site is a vertex in V) that can probe the network via shortest paths and we then attempt to determine the exact set of links in a network. Beerliova et al. [7] discuss online algorithms to determine the minimum number of query sites required to discover all edges in a network. In the network monitoring, or covering, problem we want to determine the status of all lines that are known to be in a network based upon a set of query sites. In the **shortest path cover problem** we want to cover every edge of a known graph representing a network by the shortest paths from a subset of vertices in the graph. We consider two different models of graph covering via shortest paths, our goal being to minimize the number of required query sites.

We define two query models that respectively characterize the worst and best case sets of edges that can be learned from a single query. Consider a query site v and, for each other vertex u, let $I_{v,u}$ be the set of edges that are in every shortest path of $P_{v,u}$ ($I_{v,u} = \bigcap_{p \in P_{v,u}} p$). Let I_v be the union of edges in $I_{v,u}$ over all other vertices u in G. For an example, Figure 10(B) highlights the edges in I_v for a given graph G. This model represents the minimal amount information that is available in any shortest path query from a single vertex.

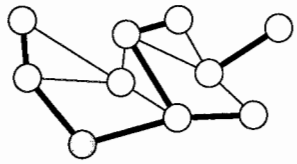
Our second model is defined more generously. Let $U_{v,u}$ be the set of edges that are on some path of $P_{v,u}$ ($U_{v,u} = \bigcup_{p \in P_{v,u}} p$). Let U_v be the union of all $U_{v,u}$ for all $u \in V$. This model represents that maximal amount of information that is available from a single vertex. An example U_v is illustrated in Figure 10(C).

Observation 4.1.1. *From the definitions, it immediately follows that $I_v \subseteq U_v$ for every vertex $v \in V$ in $G = (V, E)$*

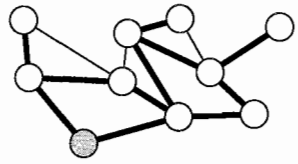
A set of query sites, a subset of V, is said to be a **cover** of $G = (V, E)$ if the union of the edges covered by the vertices in the set is exactly E. Vertex v is said to **intersection-cover** the edges in I_v and **union-cover** the edges in U_v. We have two types of queries, which leads to two models of shortest path covering. A **shortest**



(A) A graph G with query vertex v colored gray



(B) I_v of G with v colored gray



(C) U_v of G with v colored gray

FIGURE 10. Examples of each type of query for a given graph and vertex.

path intersection cover of graph G, $SPC_I(G)$, is a subset of vertices such that every edge in G is intersection-covered by at least one vertex in $SPC_I(G)$. A **shortest path union cover** is defined analogically and denoted $SPC_U(G)$.

A **minimal** cover is a cover such that removal of any vertex results in a subset of vertices that is no longer a cover, and a **minimum** cover is a minimal cover having the fewest vertices. The $\mathcal{NP}$-hardness of finding minimum covers is discussed in the Section 4.2. Minimal covers can be found efficiently through a hill-climbing strategy[2], because it is not difficult to verify that a set covers a graph with shortest paths.

Let v be a vertex of connected graph G such that the graph obtained by the removal of v (and the edges incident to v) has $k > 1$ connected components $G'_1 \ldots G'_k$. We say that v is an **articulation point** that **separates** G into subgraphs $G_1 \ldots G_k$, where $\{G_i | 1 \leq i \leq k\}$ is obtained from G'_i by adding back v and all the edges from v to any vertex in G'_i. We have the following result that is independent of query type.

[2]The hill-climbing algorithm is as follows: Take the set of all vertices V as our initial set of query sites. This set definitely covers the graph. Next, loop through the set of query sites and see if it is possible to remove a vertex from this set while keeping the entire graph covered. Repeat the loop until it is not possible to remove another vertex. The remaining set forms a minimal cover.

Theorem 4.1.2. *Given graph G with articulation point v that separates G into $G_1 \ldots G_k$, the size of the shortest path cover of G is bounded as follows:* $\sum_{i=0}^{k} |SPC(G_i)| - k \leq |SPC(G)| \leq \sum_{i=0}^{k} |SPC(G_i)|$.

Proof. Consider first the case where no minimum cover of the components includes v for any $1 \leq i \leq k$. Then, $SPC(G) = \sum_{i=0}^{k} SPC(G_i)$. Edges covered through v from one side of the articulation point to the other will therefore not reduce the size of the covering set in other components.

Now we consider the case where all minimum covers of G_i include v for all $1 \leq i \leq k$. Because all shortest paths from G_i to G_j $(i \neq j)$ must go through v, the edges covered by any query site of any vertex in G_i will necessarily include all edges that query query site v covers in G_j, and vice versa, for all distinct i and j in the range $1 \ldots k$. Therefore, if $|SPC(G_i)| > 1$ and $|SPC(G_j)| > 1$ for distinct i and j, then $|SPC(G)| = \sum_{i=0}^{k} |SPC(G_i)| - k$.

Finally, we consider the middle case, where some of the minimum covers of some G_i include v, but others do not. If we assume that j of the covers include v, then $|SPC(G)| = \sum i = 0^{k} |SPC(G_i)| - j$.

The number of G_i that have a minimum covering containing v is guaranteed to be between 0 and k (inclusive). Therefore the number of vertices required to cover G graph may range from $SPC(G) = \sum_{i=0}^{k} SPC(G_i) - k$ (as in the second case) to $\sum_{i=0}^{k} SPC(G_i)$ (as in the first case). □

This insight into the structure of solutions containing articulation points informs many of our results, and it also paves the way for later results involving 2-trees in Section 4.4. Prior to going any further, however, we prove that the shortest-path covering problem is computationally difficult in the general case.

4.2 $\mathcal{NP}$-Completeness of SPC

Before we discuss the computational complexity of the problems, it is useful to first define the corresponding decision problems for union-cover and intersection-cover. We begin with the intersection-cover decision problem and then define union-cover symmetrically:

Definition 4.2.1 (IntersectionCover).

INSTANCE: Graph $G = (V, E)$ and parameter k.

QUESTION: Is there a subset of vertices in V with cardinality no more than k, such that every edge in E is intersection-covered by at least one vertex in the set?

Definition 4.2.2 (UnionCover).

INSTANCE: Graph $G = (V, E)$ and parameter k.

QUESTION: Is there a subset of vertices in V with cardinality no more than k, such that every edge in E is union-covered by at least one vertex in the set?

In both cases, we will prove $\mathcal{NP}$-completeness of the problem by a reduction from the $\mathcal{NP}$-complete problem of VertexCover as defined by Garey and Johnson[36]. In the vertex cover problem, we wish to choose a set $S \subseteq V$ of vertices of a graph $G = (V, E)$ such that $\forall (u, v) \in E$, $|\{u, v\} \cap S| \geq 1$ and $|S| \leq k$.

Definition 4.2.3 (VertexCover).

INSTANCE: Graph $G = (V, E)$ and parameter k.

QUESTION: Is there a subset S of vertices in V with cardinality no more than k, such that at least one endpoint of every edge in E is also in S?

The reduction for SPC_I is more straightforward, so we begin with that one.

Theorem 4.2.4. *Given an arbitrary graph G, the* IntersectionCover *problem is $\mathcal{NP}$-complete.*

Proof. To prove that IntersectionCover is $\mathcal{NP}$-complete, we first show that it is in the class $\mathcal{NP}$, and then we prove completeness. To show membership, we

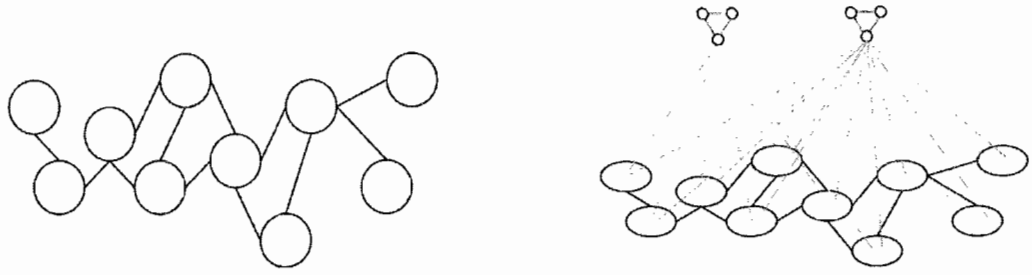
efficiently verify that a proposed solution actually intersection-covers the graph. Given a subset of vertices S such that $|S| \leq k$, we need to calculate the set of covered edges for each vertex in the set and then determine that the union of the edge sets is equal to E. To find the edges shortest-path covered by a single vertex v, we perform a breadth-first search from v to every other vertex in the graph. Then, for each other vertex u, if, for every $i \leq d(u, v)$ there is only a single edge (x, y) such that $d(u, x) + 1 + d(y, v) = d(u, v)$, then that edge is intersection-covered by v. This can be done in polynomial time, and therefore the problem is in $\mathcal{NP}$.

We show completeness by reduction from VERTEXCOVER. Take graph $G = (V, E)$ and add two new cycles of length (K_3) made up of a total of six new vertices, and call them T_1 and T_2. Now, choose one vertex v_1 from T_1 and one vertex v_2 from T_2. Now, add an edge from each of v_1 and v_2 to every vertex in V, forming $G' = (V', E')$ as in Figure 11. An intersection-cover of G' consists of a vertex in T_1 that is not v_1 and a vertex in T_2 that is not v_2 plus a solution to the vertex cover problem on G. Note that the I_u in G' for all $u \in V$ only includes edges of G that are neighbors of u in G; and for $v \in \{v_1, v_2\}$, I_v in G' consists only of the edges incident to v. Therefore, if there is an intersection cover of size $k + 2$ of G', there must exist a vertex cover of G of size k.

Because INTERSECTIONCOVER is in $\mathcal{NP}$ and VERTEXCOVER is reducible to it, INTERSECTIONCOVER is $\mathcal{NP}$-Complete. □

The proof for UNIONCOVER uses a similar graph transformation, but is more involved.

Theorem 4.2.5. *Given arbitrary graph G, the* UNIONCOVER *problem is $\mathcal{NP}$-Complete.*



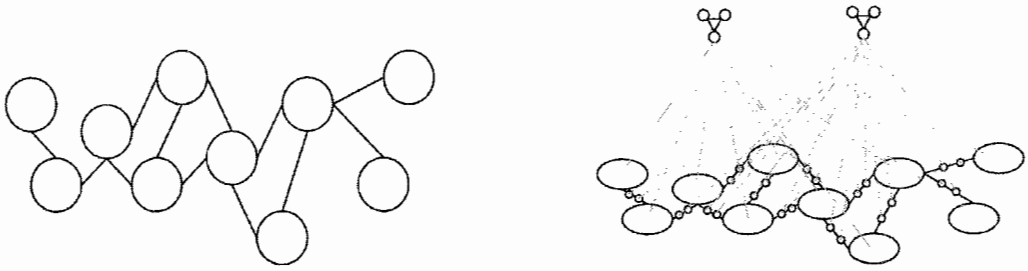
(A) Graph G input to the problem VertexCover

(B) Graph G' input to the problem IntersectionCover

FIGURE 11. An instance of the Vertex Cover problem and its translation into an Intersection Cover problem.

Proof. If we are given the subset of vertices that are in a proposed union cover of size k, then verification of this problem is easily accomplished in polynomial time simply by creating each shortest-path-union tree and verifying that every edge is present in at least one tree. The shortest-path union tree for a given vertex v consists of all edges (x, y) such that $d(v, x) + 1 = d(v, y)$. This can be found in polynomial time via breadth-first search, and therefore the problem is in $\mathcal{NP}$.

Our reduction from the vertex cover problem begins by transforming the input graph $G = (V, E)$. First, we subdivide every edge in E into a path that is three edges long. All new vertices that we create in this step, we will call **path vertices**. Now, we add to G two new cycles of length three (K_3) made up of a total of six new vertices, and call them T_1 and T_2. We choose one vertex from T_1 and add an edge from it to all vertices in V. We then choose one vertex from T_2 and add an edge from it to every path vertex. We call the transformed graph G'. An example G and G' may be seen in Figure 12. We now prove that there is a vertex cover of G of size k if and only if there is a union cover of G' of size $k + 2$.



(A) A graph G input to the problem VERTEXCOVER

(B) A graph G' input to the problem UNIONCOVER

FIGURE 12. An instance of the Vertex Cover problem and its translation into a Union Cover problem.

We prove the first direction by noting that the vertices in a vertex cover of G, together with a degree-2 vertex from each of T_1 and T_2, form a union-cover of G'. In particular, the degree-2 vertex from each of the trees will union-cover two out of the three edges in the K_3, as well as all of the edges which join a vertex in V with a path vertex. All that remains to be covered are the edges which join two path vertices, which we call the central edge of the path. To cover those, we note that a union-cover rooted at a particular vertex will cover the central edge of all of the adjacent paths. Therefore, a vertex-cover of G will exactly cover all of the central edges of G'. All that remains is to prove that a union-cover of G' can be transformed into a vertex cover of G.

To show this, we note that if the set of vertices in a union-cover of G' contains no path vertices, then that cover must consist solely of vertices from T_1 and T_2 and vertices from V, and the vertices from V form a vertex cover of G. Therefore, the only union-covers that are not already vertex covers are those containing one or more path vertices. To finish our proof, we show that if a union-cover contains a

path vertex, then it can be transformed into a union cover that does not contain that path vertex without increasing its size.

If there is a path vertex in the union-cover, we may replace it with the adjacent vertex in V, as the edges covered by the replacing vertex are a superset of the edges covered by the original vertex. By repeating this replacement procedure until there are no more path vertices in the union-cover, we create a union-cover of G' that is directly transformable to a vertex cover. The vertex cover of G is exactly the intersection of V and the vertices in the union-cover of G'.

Thus, UNIONCOVER is in $\mathcal{NP}$, and VERTEXCOVER may be reduced to it. Therefore, UNIONCOVER is $\mathcal{NP}$-complete. □

Now that we know that INTERSECTIONCOVER and UNIONCOVER are $\mathcal{NP}$-Complete in the general case, we begin our search for specific subclasses of graphs upon which these problems are tractable.

4.3 Easy Graph Classes

As defined, the edges that are intersection-covered by a single vertex v (I_v) necessarily include all edges incident with v, for all v in G, and by Observation 4.1.1, this is true for union-covers from v (U_v) as well. This implies the following:

Observation 4.3.1. *The size of a minimum shortest path cover (of either type) of a graph G is less than or equal to the size of a minimum vertex cover of G.*

This upper bound is tight for the complete graph K_n, where both vertex cover and all models of shortest path cover require $n-1$ vertices, as all shortest paths are of length 1. For intersection covers, this tight bound extends to complete bipartite graphs, $K_{s,t}$, which are graphs in which V can be partitioned into two sets S ($s = |S|$) and T ($t = |T|$), such that $E = S \times T$. In these graphs, if $s, t > 1$, every vertex at distance 2 from a given vertex v has multiple shortest paths. Thus, I_v is limited to edges incident to v in complete bipartite graphs, and $SPC_I(K_{s,t}) = \min(s, t)$.

4.3.1 Trees

A tree $T = (V, E)$, being an acyclic, connected graph with $n-1$ edges, contains a unique shortest path between every pair of vertices. Therefore, $I_v = E$ for every vertex v in T, implying that the size of $SPC_I(T) = SPC_U(T) = 1$ for any tree T.

For any tree with diameter greater than 2, a shortest path cover has fewer vertices than a vertex cover, which shows that our bound in Lemma 4.3.1 is not always tight. In a graph G, any vertex can only shortest path cover edges in its connected component of G. These observations are formalized as follows:

Observation 4.3.2. *For any graph G, the size of a minimum shortest path cover is bounded below by the number of connected components of G.*

Observation 4.3.3. *In any acyclic graph G, the size of a minimum shortest path cover is equal to the number of connected components of G.*

Thus, without loss of generality, we will discuss only connected graphs.

4.3.2 Unicyclic Graphs and Cacti

A unicyclic graph is a graph containing exactly one cycle, and a cactus is a generalization of this class. We begin with unicyclic graphs.

Lemma 4.3.4. *If G is unicyclic, a minimum $SPC_I(G)$ consists of two vertices.*

Proof. If G contains a cycle of odd length (an odd cycle), then there is a unique shortest path tree to each other vertex for every $v \in V$, but the edge opposite the query site on the cycle is not on any shortest path. To cover this one remaining edge, we require one more query site.

If G contains an even cycle, then for each $v \in V$ there is a unique path to every vertex but one. The vertex opposite our query site on the cycle has two shortest paths of equal length that will reach it: one proceeding clockwise around the cycle, and the other counterclockwise. Therefore, neither of the edges adjacent to this last vertex are covered by a query rooted at v. Thus, in order to cover all edges of the graph, we must use at least two non-adjacent query sites. □

In unicyclic graphs the increased knowledge available from a union-cover provides an advantage, so the result is slightly different for SPC_U.

Lemma 4.3.5. *If a unicyclic graph G contains a cycle of odd length (an odd cycle), a minimum $SPC_U(G)$ contains two vertices, but if G contains a cycle of even length (an even cycle), a minimum $SPC_U(g)$ contains only one vertex.*

Proof. In an odd cycle, then from every vertex there is a unique shortest path to every other vertex. However, there will always be one edge left out, namely the edge that is the greatest distance away from our query site. To cover that edge requires a second query site.

In an even cycle, then the union of all shortest paths rooted at a given vertex completely covers the graph. □

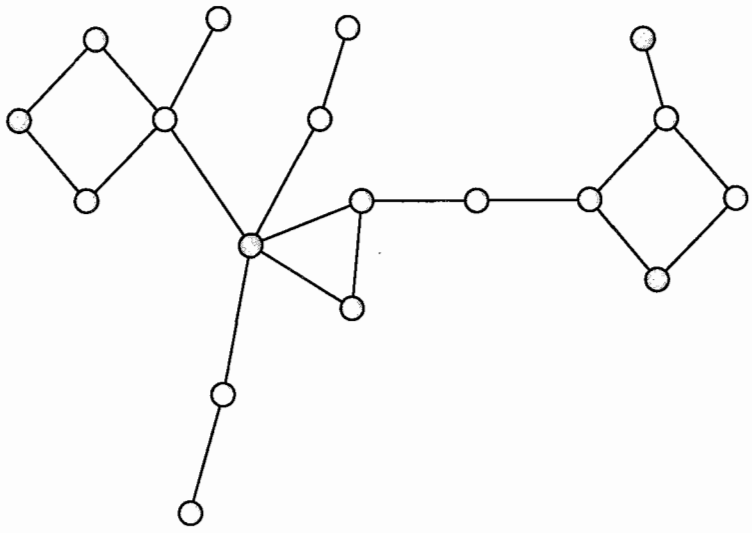


FIGURE 13. A cactus graph with three cycles, two of which are leaf cycles.

A **cactus** is either a tree, a unicyclic graph, or can be obtained from two cacti by joining them at a single vertex. An equivalent definition from Brandstädt et al. states that a cactus is a graph in which no two cycles share an edge [15]. A **leaf cycle** of a cactus is a cycle which is connected to all other cycles through a single articulation point. An example of a cactus with two leaf cycles may be seen in Figure 13.

Using the tree-like structure of a cactus, with special attention to the leaf cycles of the cactus, informs our algorithms for SPC_I and SPC_G. We begin with SPC_I.

Lemma 4.3.6. *To intersection-cover a cactus that is not unicyclic requires one query vertex in every leaf cycle, and one query vertex in every even cycle that is not a leaf cycle and is joined to the other cycles in the graph at only 2 adjacent vertices in the cycle.*

Proof. By lemma 4.3.4, we require two query points to intersection cover any cycle. For the leaf cycles, the articulation point that separates the leaf cycle from the rest

of the tree provides one query point for that cycle, but one more is required to cover the edge, or edges in the cycle that are opposite the articulation point. Thus, with a query point at each of the leaf cycles, we can cover all leaf cycles.

Now we consider the cycles that are not leaf cycles. These internal cycles are joined to other cycles by at least two distinct articulation points. By Lemma 4.3.4, all the odd cycles are covered, because any two query points on an odd cycle will intersection-cover the cycle. What remains are the internal cycles that are even. If an even cycle is connected to the rest of the graph by at least two non-adjacent articulation points, then that cycle will be completely covered by the query point in leaf cycles. If not, then one edge will not yet be covered. Therefore, we must place one more query point to completely cover that cycle.

Thus, we must place a query point in every leaf cycle, and in every non-leaf even cycle that is joined to the graph by just two adjacent articulation points. □

The union-cover problem on cacti retains the emphasis on leaf cycles, but the criteria are slightly different. By Lemma 4.3.5 even cycles can be union-covered with a single query point. We leverage this increased power in generating small union-covers.

To fully describe our algorithm for SPC_U on cacti we require several new lemmata and the concept of a **leaf vertex**, which is a vertex of degree 1.

Lemma 4.3.7. *If a connected graph $G = (V, E)$ may be union-covered with $k > 0$ query points, then G', which is created by adding leaf vertex v to create $G' = (V \cup v, E \cup (v, u))$, may also be union-covered with k query points.*

Proof. To prove this lemma, it suffices to note that every shortest path to leaf vertex v must contain the edge (v, u), where u is v's sole neighbor. Therefore, every query point in V will union-cover the edge (v, u). □

We extend this construction by considering the addition of not just leaf vertices, but of cycles as well.

Lemma 4.3.8. *Given graph $G = (V, E)$ and even cycle C_{2n}, we create G' by identifying one vertex in V with one vertex in C_{2n}. If G can be union-covered with $k > 0$ query points, then G' may be covered with those same k query points.*

Proof. By the second case of the proof of Theorem 4.3.5, C_{2n} may be covered with a single vertex. Furthermore, since all vertices are isomorphic in a cycle, it may be covered by any vertex in the cycle. Thus, C_{2n} is covered by v, the articulation point that joins C_{2n} to G.

Because our articulation point v serves as a minimum cover of C_{2n}, by Lemma 4.3.5, a minimum cover G' is exactly the same as a minimum cover of G. □

Lemma 4.3.9. *Given cactus G with more than one leaf cycle and with no leaf cycles of even length, a minimum union-cover of G consists of one query point in every leaf cycle.*

Proof. By Lemma 4.3.5 every odd cycle requires at least 2 query points, and the leaf cycles of G are connected to the graph through a single articulation point. Therefore, every leaf cycle requires at least one query point placed at a vertex other than its articulation point. Every cycle that is not a leaf cycle is connected to leaf cycles through at least two separate points, and the query points placed at each of the leaf cycles will serve to cover all such non-leaf cycles. Therefore, query points are required at every leaf cycle, and those query points serve to completely cover G. □

With the preliminaries out of the way, we are ready to minimally union-cover a cactus. The algorithm to generate a minimum union-cover is to take the given cactus, call it G', and repeatedly remove leaf vertices and even leaf cycles until the resultant graph G is either a tree, unicyclic, or contains only odd leaf cycles. We then union-cover G with a single vertex if it is a tree, as in Lemma 4.3.5 if it is unicyclic, and in all other cases by placing one query point in every odd leaf cycle as in Lemma 4.3.9.

Theorem 4.3.10. *This algorithm generates a minimum union-cover of any cactus*

Proof. By Lemma 4.3.7 and Lemma 4.3.8, any union-cover of G will also cover G', and by Lemma 4.3.9, placing query points in the odd cycles of G' is a minimum union-cover of G'.

Therefore, our algorithm results in a minimum union-cover of G', which is also a minimum union-cover of G by Lemma 4.1.2. □

Having moved from trees to unicyclic graphs to cacti, we go in a new direction and consider the class of grid graphs.

4.3.3 Grid Graphs

In this section, we consider a rectangular grid graph $G_{m,n}$ consisting of vertices labeled (i, j), for $1 \leq i \leq m$ and $1 \leq j \leq n$ with edges between (i, j) and $(i + 1, j)$ for $1 \leq i < m$ (row j) and between (i, j) and $(i, j + 1)$ for $1 \leq j < n$ (column i). Given a vertex (i, j), $I_{(i,j)}$ contains only the edges in row j and column i, as there are alternative shortest paths to all other vertices.

Lemma 4.3.11. *Any vertex of $G_{m,n}$ will union cover the whole graph.*

Proof. In a grid graph, the set of shortest paths between $v = (i_v, j_v)$ and $u = (i_u, j_u)$ constitutes the subgraph of $G_{m,n}$ with edge set

$$E = \{(k,l) | \min(i_v, i_u) \leq k \leq \max(i_v, i_u) \wedge \min(j_v, j_u) \leq l \leq \max(j_v, j_u)\}$$

Thus, any vertex suffices, as $P_{v,(0,0)} \cup P_{v,(n,0)} \cup P_{v,(0,m)} \cup P_{v,(n,m)}$ contains all edges in $G_{m,n}$, and a union cover rooted at a vertex v is exactly $\bigcup_{u \in V} P_{v,u}$. □

Lemma 4.3.12. *The size of $SPC_I(G_{m,n})$ is equal to the maximum of m and n.*

Proof. As we noted earlier, $I_{(i,j)}$ only contains the vertices in row i and column j. Thus, in order to cover every vertex, every row and column must contain at least one query point. □

Grid graphs provide a nice example of a graph class in which the union cover and the intersection cover may differ drastically. Only one vertex is required to union-cover a grid graph, but many more may be required to intersection-cover a grid graph. In the next section, we provide a class where this difference is taken to an extreme.

4.3.4 Bipartite Graphs

In this section, we extend the intuition and results from union covers on grid graphs and even cycles to a larger class of graphs: bipartite graphs. Bipartite graphs are graphs in which the vertices may be partitioned into two sets, S and T, and the edge set of the graph is a subset of $S \times T$. There are many other equivalent definitions of this class of graphs, but the most useful definition for our purposes is that bipartite graphs are exactly those graphs containing no odd cycles.

Theorem 4.3.13. *The minimum size union-cover of a connected graph G with at least one edge is 1 if and only if G is bipartite.*

Proof. Suppose first that G is bipartite, and let v be an arbitrary vertex of G. We show that v covers all edges of G. For contradiction, suppose that an edge $e = \{u, w\}$ is not covered by v, i.e., $d(v, u) = d(v, w)$. In this case, there must be two paths of equal length from v to each of u and w. Let x be the last vertex that these two paths have in common. Now we have a cycle of length $d(x, u) + d(x, w) + 1$. Because the two distances are equal, this cycle is of length $2k + 1$ for some integer k (is an odd cycle). Because bipartite graphs contain no odd cycles, we have reached a contradiction, and all edges of a bipartite graph must be union-covered by a single vertex.

On the other hand, suppose that all edges of a graph G are covered by a single vertex v. Therefore, for each edge $e = \{u, w\}$ of G, $|d(v, u) - d(v, w)| = 1$. Let us color the vertices an even distance from v by color 1 and the vertices in odd distance by color 2. This is a proper coloring of G by two colors, hence G is bipartite. □

This proof does not extend to intersection-covers. In the previous sections, we enumerated at least three separate classes of bipartite graphs with extremely different intersection-covers: Trees can be intersection-covered with a single vertex (Section 4.3.1), grid graphs require a number of query vertices equal to the largest length in any dimension (Section 4.3.3), and complete bipartite graphs require one side of the graph to be completely covered (Section 4.3). Intersection-covers are not as powerful a model as union-covers, and bipartite graphs serve as an excellent illustration of that fact.

4.4 Union Covers of 2-Trees

Through this chapter, we have been analyzing graphs that have a structure of which we can take advantage. We now turn our attention to graphs which are a generalization of trees. A **k-tree** is a type of graph that has, at its base level, a

tree-like structure that we will attempt to exploit. For example, a tree has treewidth 1, and a clique of size n has treewidth $n-1$. Our formal definition of a k-tree comes from Rose[28] via Kloks[44].

Definition 4.4.1 (k-tree). *A graph G is a k-tree iff:*

1. *G is connected,*
2. *The maximum clique contained in G is of size $k+1$,*
3. *every minimal vertex separator of G induces a clique of size k.*

This definition implies a method of k-tree construction. Namely, we note that a clique of size $k+1$ is a k-tree, and to combine two k-trees into a single k-tree, we take two subgraphs of each k-tree which each form a clique of size k, and join the two k-trees together by identifying each vertex of one subgraph with a unique vertex of the other subgraph. We will call the parameter k the **treewidth** of the graph, and we formally define it in another way below. This definition and construction method also leads to a method of deconstruction and analysis called **tree decomposition**.

Our definition of a tree decomposition comes from de Fluiter[25].

Definition 4.4.2 (tree decomposition). *Let $G = (V, E)$ be a graph. A tree decomposition of G is a pair (T, X), where $T = (I, F)$ is a tree, and $X = \{X_i | i \in I\}$ is a family of subsets of V, one for each vertex of T, such that:*

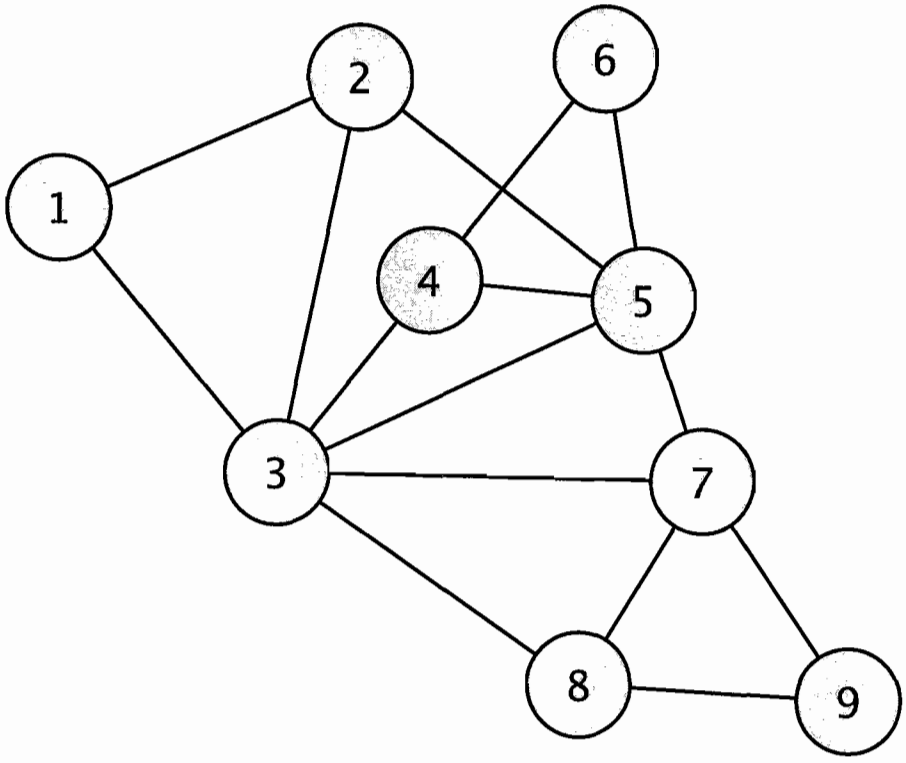
1. *$\cup_{i \in I} X_i = V$,*
2. *for every edge $\{v, w\} \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$,*
3. *for all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.*

We call each vertex in the tree decomposition a **bag**, so as not to confuse the vertices of the given graph G and the vertices of the tree in the tree decomposition. The **width** of a tree decomposition $((I, F), \{X_i | i \in I\})$ is equal to $\max_{i \in I} |X_i| - 1$, i.e. one less than the maximum size of any bag in the decomposition, and the treewidth of a graph is the minimum width over all possible tree decompositions of that graph. As a historical sidenote, the original definition of a k-tree was that it was an edge-maximal graph whose tree decomposition of minimum width was of width k, in which the addition of any edge to the k-tree would increase its treewidth. The tree decomposition of a k-tree sets each X_i to be the vertices of a distinct clique of size $k+1$ in the k-tree, and two bags $i, j \in I$ may be connected only if $|X_i \cap X_j| = k$. It will be an important fact later that any tree decomposition $((I, F), X)$ may be transformed into a decomposition of equal width $((I_b, F_b), X_b)$, where (I_b, F_b) is a rooted binary tree with $O(|V|)$ bags[25]. An example of a 2-tree, a minimum width tree decomposition of the 2-tree, and a rooted binary tree decomposition of minimum width, may all be seen in Figure 14. In this section, we restrict our attention to union covers of 2-trees and subgraphs of 2-trees, the latter of which we call **partial 2-trees**.

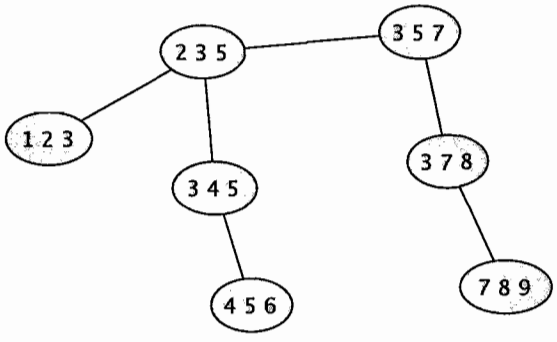
4.4.1 Union Covers Across a Clique of Size 2

The key feature of 2-trees that we will exploit is that, in the construction process of a 2-tree, we join the component 2-trees together by identifying cliques of size 2 (edges) in each graph. In the resulting graph, each edge then forms a separator through which only a limited amount of information may pass. In our next theorem, we discuss exactly how much information may travel through each of these connected 2-cuts of the graph.

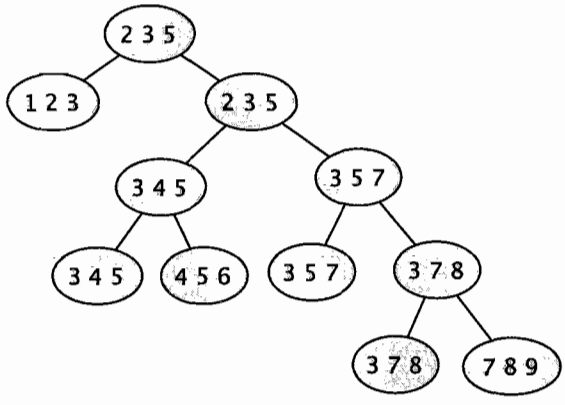
Lemma 4.4.3. *Let G be a connected graph $G = (V, E)$ and let u and v be two adjacent vertices ($\{u, v\} \in E$) such that the removal of u, v, and all incident edges*



(A) A 2-tree with labeled vertices



(B) A tree decomposition of width 2



(C) A rooted binary tree decomposition of width 2

FIGURE 14. A 2-tree and two decompositions of width 2, as defined in Definitions 4.4.1 and 4.4.2. Each bag in the decompositions is labeled with its corresponding vertex set.

from G makes G disconnected into distinct components $(V_1, E_1), \ldots (V_l, E_l)$. Furthermore, let G_i be the induced subgraph of G consisting of the vertices $V_i \cup \{u, v\}$. The set of edges of G_i union covered by a non-empty subset S of the vertices of G_j $(j \neq i)$ is completely determined by G_i and the following two integers:

1. *The maximum difference in distance between u and v over all $w \in S$ $(\max_{s \in S}(d(w,u) - d(w,v)))$*

2. *The minimum difference in distance between u and v over all $w \in S$ $(\min_{s \in S}(d(w,u) - d(w,v)))$*

Proof. By way of contradiction, assume that there exist two sets $S_1 \subseteq V_j \cup \{u, v\}$, and $S_2 \subseteq V_j \cup \{u, v\}$, in some G_i $(i \neq j)$ that have the exact same properties listed above, but there exists some edge $e = (x, y)$ in G_i that is union covered by S_1 and not S_2.

Therefore, there is a vertex $s_1 \in S_1$ in which e is an element of a shortest (s_1, y)-path, but there is no such $s_2 \in S_2$ where that is true. Let us assume that a shortest path containing the edge e goes through the vertex v before it goes through u, if it goes through vertex u at all. Therefore, $d(s_1, v) \leq d(s_1, u)$, and thus $d(s_1, x) = d(s_1, v) + d(v, x)$ and $d(s_1, y) = d(s_1, v) + d(v, y)$, which implies that $d(v, x) + 1 = d(v, y)$.

Because e is not union covered by any $s_2 \in S_2$, it must be true that $d(s_2, x) = d(s_2, y)$ for all $s_2 \in S_2$. Since $d(v, x) + 1 = d(v, y)$, v can not be on any shortest paths from s_2 to y for any $s_2 \in S_2$. Therefore, for all $s_2 \in S_2$, $d(s_2, v) + d(v, y) > d(s_2, u) + d(u, y)$. Because u and v are connected, we also know that $0 \leq d(s_2, v) - d(s_2, u) \leq 1$ and $0 \leq d(v, y) - d(u, y) \leq 1$.

In this situation, at least one of two conditions holds. Either $d(s_2, v) > d(s_2, u)$ for all $s_2 \in S_2$, or there is at least one $s_2 \in S_2$ in which $d(s_2, v) = d(s_2, u)$, and therefore $d(v, y) - d(u, y) = 1$. We reason by cases to show that either situation leads to a contradiction.

In the first case, we assume that $d(s_2, v) > d(s_2, u)$ for all $s_2 \in S_2$. In particular, note that there is no vertex $s_2 \in S_2$ where $d(s_2, v) < d(s_2, u)$ and therefore $\max_{w \in S_2}(d(w, u) - d(w, v)) = -1$. However, there is such a vertex $w \in S_1$ where $d(w, u) - d(w, v) \geq 0$; the vertex with that property is precisely the vertex that union covers e. Therefore, $\max_{w \in S_1}(d(w, u) - d(w, v)) > -1$ and we have our contradiction.

In the second case, we know that there exists some $s_2 \in S_2$ where $d(s_2, v) = d(s_2, u)$ and also we know that $d(v, y) - d(u, y) = 1$. We can immediately conclude that $\max_{w \in S_2}(d(w, u) - d(w, v)) = 0$. If $d(v, y) - d(u, y) = 1$, then, recalling that $d(v, x) + 1 = d(v, y)$, we know that $d(v, x) + 1 - d(u, y) = 1$ and that $d(v, x) = d(u, y)$. In this case, in order for S_1 to union cover e, it must be true that for some $s_1 \in S_1$, $d(s_1, v) < d(s_1, u)$. Therefore, $\max_{w \in S_1}(d(w, u) - d(w, v)) = 1$, and we have reached the contradiction for our second case.

Our argument is entirely symmetric, so if we assume that u, rather than v, is the first vertex along the shortest path containing e, it immediately implies that the minimum difference in distance between u and v over all $w \in S$ ($\min_{s \in S}(d(w, u) - d(w, v))$) must also be the same for S_1 and S_2.

Therefore, if, for two sets of vertices S_1 and S_2 from some G_i, $\max_{w \in S_1}(d(w, u) - d(w, v)) = \max_{w \in S_2}(d(w, u) - d(w, v))$ and $\min_{w \in S_1}(d(w, u) - d(w, v)) = \min_{w \in S_2}(d(w, u) - d(w, v))$, then the two vertex sets union cover the exact same set of edges in all G_j $(i \neq j)$. □

We use this hard-fought insight to develop a dynamic program for 2-trees by noting that the maximum intersection between the elements of a union cover of one G_i with another G_j is exactly the vertices $\{u, v\}$.

4.4.2 A Dynamic Program for 2-trees

We define the concept of **support** that a union cover S of some G_i provides across the vertices of the edge (u, v) to G_j to be exactly those two properties in Lemma 4.4.3. The support that one union cover provides will cover some edges of G_j. It is quite likely that this support will actually decrease the number of vertices in G_j that are required to cover G_j.

In this way, we design our dynamic program to match up minimum union covers some G_i that provide a given support to and require a particular support support to be provided to them. We give a recursive description of the algorithm that is suitable for memoization.

Our algorithm begins by generating a rooted binary tree decomposition of minimum width. Such a decomposition may be found in linear time using the algorithm of Matoušek and Thomas[42], although the special case of 2-trees has a much simpler linear-time algorithm: repeatedly remove degree-2 vertices, making the removed vertex and its neighbors into a bag, until we arrive at an instance of K_3. We next define a recursive algorithm to find all combinations of required and provided support that will cover the entire graph. The pseudo-code for such an algorithm is detailed in Figure 15.

The algorithm is linear time (although the constants may be large), because the state space of possible arguments for a given bag of the tree decomposition is also constant. S_{in} and S_{out} may range from -1 to 1, and v_{flag} and u_{flag} are each either true or false, yielding a constant number of recursive calls at every level of the tree. Because our solution is amenable to memoization, each recursive call is called at

Union-Cover-2-Tree $(G, D = ((I, F), \{X_i | i \in I\}), S_{in}, S_{out}, v_{flag}, u_{flag}, bag)$
// G is a 2-tree
// D is a rooted binary tree decomposition of G.
// S_{in} is the provided support coming in to this subtree
// S_{out} is the requested support coming out of this subtree
// v_{flag} indicates whether the vertex v from the two-cut is already in the cover
// u_{flag} indicates whether the vertex u from the two-cut is already in the cover
// bag is the bag of the 2-tree we are analyzing

if bag is a leaf of the tree decomposition
    // Iterate over all subsets of the fixed-size bag
    return the size of the minimum union cover of this bag which provides the requested support and covers the bag with the provided support, and includes u and v (if those flags are set).
else
    $size \leftarrow \infty$
    for every possible value of S_{in}, S_{out}, v_{flag}, and u_{flag}
        $size \leftarrow \min(size,$
            Union-Cover-2-Tree $(G, D, S_{in}, S_{out}, v_{flag}, u_{flag},$left-child$(bag))$
            $+$
            Union-Cover-2-Tree $(G, D, S_{out}, S_{in}, v_{flag}, u_{flag},$right-child$(bag))$
    return size

FIGURE 15. A linear-time algorithm for union-covering 2-trees

most once, and then becomes a table lookup. The size of the table is $O(|T|)$, where $|T|$ is the size of the rooted binary tree decomposition, which, as previously discussed is $O(|V|)$.

We now extend our algorithm to partial 2-trees.

4.5 Union Covers of Partial 2-Trees

A **partial k-tree** is a subgraph of a k-tree. We induct on the underlying k-tree structure of the graph in order to build an efficient dynamic program for calculating the minimum union-cover of a partial 2-tree, and the structure we use is exactly the minimal separators of the graph, with a little help from Lemma 4.4.3.

Armed with the knowledge that only a very limited amount of information can propagate across a separator of size 2, which we call a 2-cut, we build a dynamic program very much like the algorithm for full 2-trees. The main difference is that, for partial 2-trees, the parameters $\min_{s \in S}(d(w,u) - d(w,v))$ and $\max_{s \in S}(d(w,u) - d(w,v))$, instead of being either -1, 0, or 1, can range from $-|V|$ to $|V|$, which complicates both our algorithm and its subsequent analysis.

We begin our algorithm by calculating a rooted binary tree decomposition of width 2 for this partial 2-tree using the algorithm of Matoušek and Thomas[42]. We then recursively process this tree in a way that enables memoization. Pseudocode for our algorithm may be found in Figure 16.

The key difference of the algorithms in Figures 15 and 16 is somewhat hidden by the generic way the two algorithms are expressed. The **for** loop, which needs to iterate over all possible combinations of S_{in}, S_{out}, v_{flag}, and u_{flag}, has to iterate over many more possibilities when we are analyzing partial 2-trees. In the former algorithm the support provided to and from the bag containing u and v was always at most 1 and at minimum -1, because the vertices were adjacent. In the case of

UC-Partial-2-Tree $(G, D = ((I, F), \{X_i | i \in I\}), S_{in}, S_{out}, v_{flag}, u_{flag}, bag)$
// G is a 2-tree
// D is a rooted binary tree decomposition of G.
// S_{in} is the provided support coming in to this subtree
// S_{out} is the requested support coming out of this subtree
// v_{flag} indicates whether the vertex v from the two-cut is already in the cover
// u_{flag} indicates whether the vertex u from the two-cut is already in the cover
// *bag* is the bag of the 2-tree we are analyzing

if *bag* is a leaf of the tree decomposition
    // Iterate over all subsets of the fixed-size bag
    return the size of the minimum union cover of this bag which provides the requested support and covers the bag with the provided support, and includes u and v (if those flags are set).
else
    $size \leftarrow \infty$
    for every possible value of S_{in}, S_{out}, v_{flag}, and u_{flag}
        // Note that S_{in} and S_{out} may be anywhere in the range $\pm d(u, v)$
        $size \leftarrow \min(size,$
            UC-Partial-2-Tree $(G, D, S_{in}, S_{out}, v_{flag}, u_{flag},$left-child$(bag))$
            + **UC-Partial-2-Tree** $(G, D, S_{out}, S_{in}, v_{flag}, u_{flag},$right-child$(bag))$
    return size

FIGURE 16. A linear-time algorithm for union-covering 2-trees

partial 2-trees the distance between the two vertices may be quite large, and so the corresponding support provided or support requested may be similarly large, and so our algorithm must search a larger space of possibilities at each bag.

The search component of our dynamic program involves determining the minimum size set from each component required so that the support received is the support required to cover the component, and the support given is the support required to cover the other component. The supports provided and requested, S_{in} and S_{out}, may each range from $-|V|$ to $|V|$, for a total search space at each bag of size $O(|V|^2)$. So, there are $O(|V|)$ bags, and at each bag, and for each of the $O(|V|^2)$ possible input parameters to our covering function at each bag, we want to find the minimum over a search space of size $O(|V|^2)$. So we know that we will end up calling the recursive function $O(|V|^5)$ times, and each call takes, potentially, $O(|E|)$ time to verify that all edges are covered. This yields a runtime of of $O(|V|^5 * E)$ overall, or, if we want it in terms of the order of the vertex set $|V| = n$, a runtime of $O(n^7)$.

Having taken this cased based analysis of union covers to an extreme, we now refine our model to be more like the model required for the AS graph.

4.6 Shortest Path Trees in the Valley-Free Model

The Internet AS graph is not measured using shortest paths; it is measured via shortest valley-free paths. Therefore, we must extend our analysis of shortest paths to shortest valley-free paths if we would like our analysis to be potentially of use on the AS graph. Recall that a path on a directed, edge labeled graph is valley-free if it consists of zero or more hops that follow the directions of the underlying edge (from customer to provider in the AS graph), followed by at most one hop across a bidirectional (peering) edge, followed by zero or more hops that go strictly against the direction of the underlying edge (from provider to customer). We define $VFP_{u,v}$

to be the set of valley-free shortest paths between u and v. We then define $VFU_{u,v}$ to be the union of all edges in $VFP_{u,v}$

$$VFU_{u,v} = \bigcup_{p \in VFP_{u,v}} p$$

and $VFI_{u,v}$ is defined symmetrically as

$$VFI_{u,v} = \bigcap p \in VFP_{u,v} p$$

.

Following the previous example again, we define

$$\begin{aligned} VFU_u &= \bigcup_{v \in V} VFU_{u,v} \\ VFI_u &= \bigcup_{v \in V} VFI_{u,v} \end{aligned}$$

For these definitions, we wish to find the size of the minimum set $S \subseteq V$ such that $E = \bigcup_{v \in S} VFU_v$ or $E = \bigcup_{v \in S} VFI_v$, respectively. We refer to each of these problems as **valley free union cover** ($VFUC$) and **valley free intersection cover** ($VFIC$). Unfortunately, much like SPC_I and SPC_U, the decision problem for each of these problems is $\mathcal{NP}$-complete, albeit via different reductions from before. For both of these reductions, we will require a variant of breadth-first search that will find not the shortest path, like breadth-first search normally does, but rather the shortest valley-free path. We begin by describing the algorithms to find $VFU_{u,v}$ and $VFI_{u,v}$.

To find the union of all shortest paths from a vertex u of $G = (V, E)$ to a vertex v, we perform a breadth-first search rooted at u and then mark every vertex of V

with its distance from u. Then, beginning at vertex v, we perform a second breadth-first search. This time, we propagate our search only across edges which link a vertex of distance k from u to a vertex of distance $k+1$. Every time we propagate our search across such an edge, we add that edge to the set of covered edges. This algorithm works for shortest paths, but valley-free shortest paths have an additional constraint. Our algorithm for finding $VFU_{u,v}$ on a directed graph $G = (V, E)$ is an extension of the described algorithm for shortest paths.

In particular, to find the set of edges in $VFU_{u,v}$, we require a breadth-first search rooted at u and a breadth-first search rooted at v. We then combine the results of these two searches and attempt to find the vertices which have the smallest sum of the distance to u and the distance to v. These vertices may serve as the "top" of a valley-free shortest path from u to v. Then, from each vertex that may serve as a top of a path, we perform two more breadth-first searches: one to u, and another to v. In each of these searches, when we are at a vertex y, we only propagate across a link $(z \rightarrow y)$ if the distance to our destination from z is strictly less than the distance from y. When we propagate our search across such a link, we also add it to our set $VFU_{u,v}$. The pseudocode for this algorithm may be found in Figure 18, with supporting pseudocode for breadth-first search and reverse breadth-first search in Figures 17(A) and 17(B).

The algorithm for VFI is similar, but we must be more careful about how we extend the edge set we return. In particular, we only extend the edge set if there is just a single edge between the vertices at distance k and the vertices at distance $k+1$. To do this, we extend the algorithm for finding $VFU_{u,v}$. The algorithm is detailed in Figure 19.

Problem 4.6.1 (ValleyFreeIntersectionCover).

INSTANCE: A graph $G = (V, E)$ and a parameter k.

QUESTION: Does there exist a set $S \subset V$, $|S| = k$, such that $E = \bigcup_{v \in S} VFIC_v$?

```
BFS (G = (V, E), s) // A Breadth-first search of G from s
  d_s ← [∞ ... ∞] // The distances from s
  d_s[s] ← 0
  Q ←CREATE-QUEUE()
  ENQUEUE(s)
  while NOT-EMPTY(Q)
      v ← DEQUEUE(Q)
      for u ∈ {u|(v → u) ∈ E}
          if d_s[u] ≠ ∞
              d_s[u] = d_s[v] + 1
              if (u → v) ∉ E
                  ENQUEUE(u)
  return d_s
```

(A) Breadth-first search of a directed G from vertex s

```
Reverse-BFS (G = (V, E), s) // A search of G from s going against the edges
  d_s ← [∞ ... ∞] // The distances from s
  d_s[s] ← 0
  Q ←CREATE-QUEUE()
  ENQUEUE(s)
  while NOT-EMPTY(Q)
      v ← DEQUEUE(Q)
      for u ∈ {u|(v ← u) ∈ E}
          if d_s[u] ≠ ∞
              d_s[u] = d_s[v] + 1
              if (u ← v) ∉ E
                  ENQUEUE(u)
  return d_s
```

(B) Reverse breadth-first search of a directed G from vertex s.

FIGURE 17. The supporting breadth-first search functions for $VFU_{s,t}$.

Valley-Free-Union-BFS $(G = (V, E), s, t)$
  $VFU \leftarrow \{\}$ // The edges on any of the shortest valley-free paths from s to t.
  $d_s \leftarrow$**BFS**(G, s)
  $d_t \leftarrow$**BFS**(G, t)
  $d \leftarrow \min_{v \in V}(d_t[v] + d_s[v])$ // The valley-free distance from s to t
  $T \leftarrow \{v \in V | d_t[v] + d_s[v] = d\}$ // Vertices which are the top of a shortest path
  for $v \in T$
    $d_v \leftarrow$**Reverse-BFS**(G, v)
    $VFU \leftarrow VFU \cup \{(u \rightarrow w) \in E \mid d_s[u] + 1 = d_s[w] \ \wedge \ d_v[w] + 1 = d_v[u]\}$
    $VFU \leftarrow VFU \cup \{(u \rightarrow w) \in E \mid d_t[u] + 1 = d_t[w] \ \wedge \ d_v[w] + 1 = d_v[u]\}$
  return VFU

FIGURE 18. A variant of breadth-first search which finds $VFU_{s,t}$ for a given s and t of a graph.

Valley-Free-Intersection-BFS $(G = (V, E), s, t)$
  $d_s \leftarrow$**BFS**(G, s)
  $d_t \leftarrow$**BFS**(G, t)
  $d \leftarrow \min_{v \in V}(d_t[v] + d_s[v])$ // The valley-free distance from s to t
  $T \leftarrow \{v \in V | d_t[v] + d_s[v] = d\}$ // Vertices which are the top of a shortest path
  $VFI \leftarrow E$ // Edges on the shortest valley-free paths from s to t.
  for $v \in T$
    $d_v \leftarrow$**Reverse-BFS**(G, v)
    $VFI_v \leftarrow \{\}$
    for $i = 0 \ldots d - 1$
      $level \leftarrow \{(u \rightarrow w) \in E | d_s[u] + 1 = d_s[w] \wedge d_v[w] + 1 = d_v[u] \wedge d_s[u] = k\}$
      if $|level| = 1$
        $VFI_v \leftarrow VFI_v \cup level$
      $level \leftarrow \{(u \rightarrow w) \in E | d_t[u] + 1 = d_t[w] \wedge d_v[w] + 1 = d_v[u] \wedge d_t[u] = k\}$
      if $|level| = 1$
        $VFI_v \leftarrow VFI_v \cup level$
    $VFI \leftarrow VFI \cap VFI_v$
  return VFI

FIGURE 19. An algorithm for finding $VFI_{s,t}$ on a given graph.

Theorem 4.6.2. VALLEYFREEINTERSECTIONCOVER *is* $\mathcal{NP}$*-Complete.*

Proof. We prove that VALLEYFREEINTERSECTIONCOVER is $\mathcal{NP}$-complete by a reduction from VERTEXCOVER. Consider graph $G = (V, E)$ and parameter k for which we would like to solve VERTEXCOVER. We transform $G = (V, E)$ into $G' = (V', E')$ using the following steps. First, we label all edges of E and make them bidirectional peering edges. Next, we add three vertices to V and use them to create a new K_3, and make each of its edges bidirectional. Finally, we choose one vertex u of the newly created K_3 and for each $v \in V$, we create a new directed edge from v to u. In the new graph $G' = (V', E')$, V' is the union of V and the vertices of the new K_3, and E' is the union of the now-labeled edges of E with the $v \rightarrow u$ edges and the edges of the new K_3.

We now prove that the existence vertex cover of G of size k directly implies the existence of a valley-free shortest path cover of G' of size $k + 1$, and that any valley-free shortest path cover of G' of size $k + 1$ can be converted into a vertex cover of G of size no more than k.

Converting a vertex cover of G into a valley-free shortest path cover of G' is easy — simply take the vertex cover of G and add to it one of the degree-2 vertices of the K_3. This new set of vertices forms a valley-free shortest path cover of G' of size $k + 1$. Therefore, the existence of a vertex cover of G of size k implies a valley-free shortest path cover of size $k + 1$.

Proving the opposite direction is almost as easy. Given a valley-free shortest path cover of G' of size $k + 1$, we note that no vertex in the K_3 can cover any of the peering links between vertices that were originally in V. Therefore, the peering links that were originally members of E must be covered by their immediate neighbors, and so those same vertices will serve as a vertex cover of G. In the cover of G', there is at least one vertex in the K_3, so the cover of G is of size at most k.

Therefore, we can in linear time transform a vertex cover problem into a valley-free shortest path cover problem, and a solution of the first implies a solution to the second, and a solution to the second implies a solution to the first. Noting that VALLEYFREEINTERSECTIONCOVER is in $\mathcal{NP}$, because a set of vertices can be verified to be a cover via the breadth-first search variant of Figure 19, completes our proof that VALLEYFREEINTERSECTIONCOVER is $\mathcal{NP}$-Complete. □

We define the corresponding VALLEYFREEUNIONCOVER in much the same way.

Problem 4.6.3 (VALLEYFREEUNIONCOVER).

INSTANCE: A graph $G = (V, E)$ and a parameter k.

QUESTION: Does there exist a set $S \subset V$, $|S| = k$, such that $E = \bigcup_{v \in S} VFUC_v$?

Theorem 4.6.4. VALLEYFREEUNIONCOVER *is $\mathcal{NP}$-complete.*

Proof. We begin by showing membership in $\mathcal{NP}$, and then we will prove completeness. If we are given a set S, then we can use the Valley-Free-Union-BFS algorithm of Figure 18 to calculate $VFUC_{u,v}$ for each $v \in S$. It is then trivial to check whether $\cup_{v \in S} VFUC_v = E$. Our proof is a direct corollary of our $\mathcal{NP}$-completeness reduction for VALLEYFREEINTERSECTIONCOVER.

In particular, we take the graph $G' = (V', E')$ defined in the previous reduction, and note that there is exactly one valley-free shortest path between any two vertices of V'! Therefore, $VFIC_v = VFUC_v$ for all $v \in V$, and all the results for valley-free intersection covers still hold for valley-free union-covers. □

This direct correspondence between the two models of graph covering with valley-free shortest paths means that the problem which most closely models the covering of the AS graph via router measurements, namely valley-free shortest-path tree-covers of the AS graph — is definitely $\mathcal{NP}$-complete.

In our given data set we are looking at what might be thought of as the dual of this problem. In our data, we are given spanning sets of valley-free shortest paths from multiple roots, and we would like to determine how much of the graph has been discovered by our methods. This is related to the graph discovery problem discussed by Beerliova et al.[7] and quickly leads into larger worries of data incompleteness.

4.7 Summary

Our every attempt to formalize the problem of AS graph measurement resulted in a graph covering problem that is $\mathcal{NP}$-complete. We showed that it is $\mathcal{NP}$-complete to find an optimal graph covering with intersection covers, tree covers, union covers, valley-free intersection covers, valley-free tree covers, and valley-free union covers. We did, however, find graph classes for which these formalizations were easy to calculate. We found that intersection covers and union covers differed radically in power.

After enumerating several classes of graphs which admitted polynomial-time solution algorithms for these problems, we turned our eye towards refinements of our query model. When we refined our query model to reflect the fact that we measure the AS graph with valley-free shortest paths, rather than just shortest paths, we found that the problem was still $\mathcal{NP}$-complete.

Fortunately, for studying the AS graph, we are concerned with a slightly different problem. Instead of being given a graph, we are given measurements of an unknown graph. In the next chapter we deal with what might be thought of as the dual of our problem, and take a look at issues of data incompleteness.

CHAPTER V

DEALING WITH DATA INCOMPLETENESS

Where we figure out what to do when confronted with both the incompleteness of our measurements and the impossibility of going back in time and getting better ones

How can we be sure that, when we analyze our data, we our analyzing the underlying object being measured, and not just analyzing our measurements? Because we do not have access to historical data feeds from every BGP speaking router on the Internet, our data set may be missing edges. Also, BGP does not record the edge type. Therefore, what our measurements give us is the undirected measured AS graph, as defined in Chapter II. Our data's incompleteness comes in two major forms: missing edges and missing edge directions. Both are big problems, and we will address them each in turn. We begin with the problem of missing edge directions.

5.1 Determining Edge Directions in the AS Graph

In the valley-free shortest path model of routing, a path must consist of zero or more hops that strictly follow the direction of the money (go from customer to provider or from sibling to sibling), followed by an optional peering link, followed by zero or more hops that strictly go against the flow of money (from provider to customer or from sibling to sibling). Unfortunately, our measurements provide us with undirected paths. We therefore require the ability to turn our undirected measurements into directed edges.

This is the Type-of-Relationship problem and it was introduced by Gao[34] in the same paper that introduced the shortest path model of Internet routing. There are two main ways of solving it — both techniques begin by translating the problem into an instance of 2-SAT, but the first then finds an assignment that makes the maximal number of clauses true using heuristics[6], and the second transforms the instance of 2-SAT into a semi-definite program[27], a generalization of a linear program which provides the best-known approximation for MAX 2-SAT[37]. Let us formally define the problem as follows:

Problem 5.1.1 (The Type-of-Relationship Problem).

INSTANCE: A set of undirected paths P, the union of which is undirected graph $G = (V, E)$.

QUESTION: Is there a directed graph $G' = (V, E')$, in which the paths of P are valley-free shortest paths? For every undirected edge $(u, v) \in E$, at least one of the directed edges $\{(u, v), (v, u)\}$ must be in E'.

In this model of shortest-path routing, we look at only customer-provider links and peering links. It is assumed (and this has been backed up by ISP surveys, most notably Dimitroupolos et al.[27]) that sibling-sibling relationships are so rare that

they need not be explicitly modeled. When attempts have been made to account for sibling-sibling edges, the solution used has always been to add sibling-sibling edges in during post-processing. Thus, to solve the Type-of-Relationship problem, we restrict ourselves to considering customer-provider links and peer-to-peer links. Customer-provider links are modeled as directed edges from customer to provider, and peer-to-peer links are modeled as two directed edges, a situation which we call a "bidirectional link".

Recall that a valley-free path $p = (v_1, v_2, \ldots v_k)$ consists of zero or more $v_i \rightarrow v_{i+1}$ links, followed by a single optional bidirectional link, followed by zero or more $v_i \leftarrow v_{i+1}$ links. This makes it different from the traditional view of a path in a directed graph, because parts of the path go against the direction of the underlying edges.

The conversion of an instance of the Type-of-Relationship problem into an instance of 2-SAT hinges on the fact that our measurements are of valley-free paths, and therefore once a path has started to go against edges or across a peering link (a bidirectional edge), it can never go with the edges. More explicitly, if we have an undirected measurement $A - B - C - D$ of a directed valley free path, then we also know that if $C - D$ is actually $C \rightarrow D$, then we must have $B \rightarrow C$ and $A \rightarrow B$. Otherwise, our path would not be valley free! Also, by the same logic, if $B - C$ were actually $B \leftarrow C$, then we must also have $C \leftarrow D$. We then transform the direction of an edge into a logical variable in an instance of 2-SAT. We will call this literal x_{AB} to indicate that it refers to the edge between A and B. If this literal is true, then $A - B$ is actually $A \rightarrow B$, and if the literal is false, then $A - B$ is actually $A \leftarrow B$. We can then formalize the statements we previously made about the path $A - B - C - D$ as $x_{BC} \Rightarrow x_{AB}$, $\neg x_{BC} \Rightarrow \neg x_{CD}$, and so on.

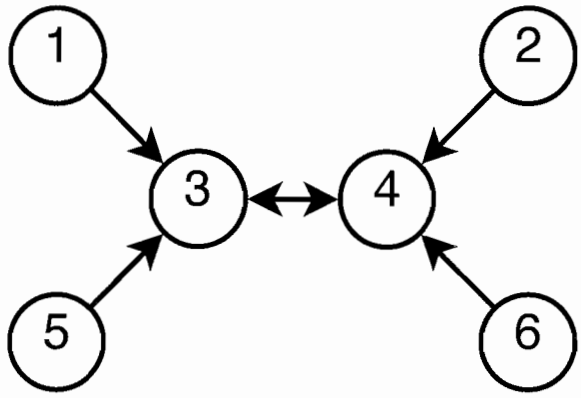


FIGURE 20. A simple example network for our direction-inference procedure

TABLE 1. All the shortest paths measured from all points of the network in Figure 20.

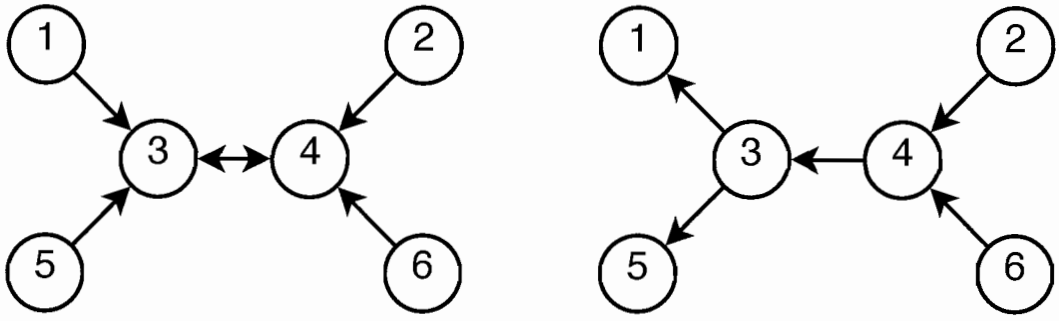
Measurement point	Paths Discovered				
1	1-3-4-2	1-3	1-3-4	1-3-5	1-3-4-6
2	2-4-3-1	2-4-3	2-4	2-4-3-5	2-4-6
3	3-1	3-4-2	3-4	3-5	3-4-6
4	4-3-1	4-2	4-3	4-3-5	4-6
5	5-3-1	5-3-4-2	5-3	5-3-4	5-3-4-6
6	6-4-3-1	6-4-2	6-4-3	6-4	6-4-3-5

Let us try a concrete example. Consider the "bowtie" network in Figure 20. If we took a measurement from every single vertex in the bowtie, we would see the paths of Table 1.

These paths represent the sum total of the information which it is possible to squeeze out of the graph, and with them, we set up an extremely large instance of 2-SAT as described. This instance of 2-SAT contains many, many clauses — on the order of the sum of the square of the length of each path. Because of this quadratic explosion, in Table 2 we only show the clauses resulting from the measurements at vertex 1 with the understanding that all of the other clauses are derived in the same fashion.

TABLE 2. The part of the 2-SAT instance resulting just from paths beginning with vertex 1 in Figure 20. The *Derived Clauses* cell for 1-3 is empty because no clauses may be derived from a path of two vertices.

Path	Derived Clauses
1-3-4-2	$(\neg x_{13} \Rightarrow \neg x_{34}) \wedge (\neg x_{13} \Rightarrow \neg x_{42}) \wedge (\neg x_{34} \Rightarrow \neg x_{42}) \wedge (x_{42} \Rightarrow x_{34}) \wedge (x_{42} \Rightarrow x_{13}) \wedge (x_{34} \Rightarrow x_{13})$
1-3	
1-3-4	$(\neg x_{13} \Rightarrow \neg x_{34}) \wedge (x_{34} \Rightarrow x_{13})$
1-3-5	$(\neg x_{13} \Rightarrow \neg x_{35}) \wedge (x_{35} \Rightarrow x_{13})$
1-3-4-6	$(\neg x_{13} \Rightarrow \neg x_{34}) \wedge (\neg x_{13} \Rightarrow \neg x_{46}) \wedge (\neg x_{34} \Rightarrow \neg x_{46}) \wedge (x_{46} \Rightarrow x_{34}) \wedge (x_{46} \Rightarrow x_{13}) \wedge (x_{34} \Rightarrow x_{13})$



(A) The original graph (B) One possible reconstruction

FIGURE 21. An example of a graph with multiple solutions for reconstruction. Note that all shortest paths in the reconstruction are still valley-free.

Once this 2-SAT instance is fully written out, we would solve it via traditional methods. Unfortunately, although this instance of 2-SAT is satisfiable, it is also under-specified: there are multiple satisfying assignments. In Figure 21 we can see both the original graph, as well as a satisfying assignment that differs from the original graph. In this regard, our simple example is a bit misleading. Due to the variation of contracts on the Internet, it is quite likely that, for the measured AS graph, the derived 2-SAT instance will be unsatisfiable. If the 2-SAT instance is unsatisfiable, however, we must try a slightly different approach. In the paper by Di Battista et al.[6], the authors prove the problem of maximizing the number of satisfied paths to be $\mathcal{NP}$-complete. They then advocate turning the problem into an instance of MAX 2-SAT (itself an $\mathcal{NP}$-Complete problem) and then applying heuristics as the best choice, while Subramanian et al. provide a heuristic that avoids 2-SAT altogether[65]. On the other hand, Dimitroupolos et al.[27] advocates turning the problem into a semi-definite program and then using a semi-definite program solver to discover the assignment of directions. Note that a semi-definite program solver should properly be called an optimizer rather than a solver, as its solutions are not guaranteed to be optimal.

These approaches are all compared in Dimitropoulos et al[27], where they use a broad survey of AS operators to discover both what adjacencies of those operators are missing from Route Views as well as what type of relationship each edge is. As it turns out, the "ground truth" of Internet connectivity is that there are operational requirements that, occasionally, require an Internet operator to hardcode in routes which are either not a shortest path, or are not valley-free[55]. Also, neither of the methods that involve creating an instance of 2-SAT will ever find an edge that must be a peering edge or must be a sibling edge. The sibling edges are simply eliminated from their models, and, because peering edges are more

restrictive than simply choosing a single direction for a given edge, peering edges will never arise in a MAX 2-SAT solution unless other heuristics are added.

Dimitropoulos et al. compared all of these solution methods and found, counterintuitively, that the semi-definite programming approach, while it contained more unsatisfied clauses in the corresponding 2-SAT instance, was generally more correct in the answers it found. They then went further and provided several heuristics, backed up with AS operator survey data for validation, that made the result of the semi-definite programming approach even more correct, and also added sibling edges to the graph. Therefore, we will use their methods for determining edge-type in the AS graph. Even better, a team at CAIDA created a repository of processed topology data[41] in which all edge types have been assigned by this best-of-breed method! When this repository is available for a given date, it is this topology we use as an initial graph.

All of these techniques help determine edge direction — especially the preprocessed graphs from CAIDA. That is only half the story, however. We may also be missing entire edges from our sample. In the next section we deal with the missing-edge problem.

5.2 Determining which Edges Might Have Been Missed

Our measurements of the AS graph, once the edge directions have been deduced, are still incomplete. As we saw in Chapter IV, it is quite possible for a set of shortest-path trees to fail in covering a graph. Thus, we must answer the question: What graph did our measurements come from? This question is unanswerable, however, as it requires us to reason beyond the bounds of our data. We instead ask the following questions: What is the largest graph from which our measurements

might have come? What is the smallest? What is the set of graphs from which our measurements might have come? What is the most likely graph from which our measurements have come? These questions will allow us to express our confidence in any conclusions that we draw from AS graph analysis. We may not have direct access to the complete AS graph, but we can develop methods that attempt to assess how close our measured AS graph is to the complete AS graph.

In our effort to tackle these problems, our fundamental insight is that each valley-free shortest path is an assertion of the existence of certain edges, but it is also an assertion about the nonexistence of other edges. For example, if we measure the path $1 \rightarrow 2 \rightarrow 3 \leftrightarrow 4 \leftarrow 5 \leftarrow 6$, then we can be confident that there exists no edge from 1 to 6, because the existence of such an edge would contradict our path being a shortest valley-free path. In this particular example, out of a possible complete graph on 6 vertices (30 possible edges), we can be certain of the existence of 6 edges (the links on the path) and we can be certain about the nonexistence of 22 more. Thus, with one path of length 6, we learn all about 6 existing edges and 22 nonexisting edges, for a total knowledge of 28 out of 30 possible edges. A full listing of these edges may be found in Table 3.

Note, however, that while our certainty about the existence of the 6 edges we measured on the path is 100%, our certainty about the impossible edges is conditional on the quality of our model. In particular, some of the impossible edges would make our purported shortest path even shorter, while others would violate the more subtle requirement that the path be valley-free, or that there be at most one peering link in a path, and that said peering link be at the very top. Thus, depending on one's confidence in the model, the interested researcher may relax the tightness of some assumptions, and place some of the so-called "impossible" edges back into the domain of possible edges. Moving forward in this study, however, we

TABLE 3. The conclusions we may draw from the path $1 \rightarrow 2 \rightarrow 3 \leftrightarrow 4 \leftarrow 5 \leftarrow 6$

Existing edges	*Possible edges*	*Impossible Edges*
Each of these edges is the measured path: $1 \rightarrow 2$, $2 \rightarrow 3$, $4 \rightarrow 3$, $3 \rightarrow 4$, $5 \rightarrow 4$, $6 \rightarrow 5$	The existence of these edges remains undetermined: $3 \rightarrow 1$, $4 \rightarrow 6$	The existence of any of these edges would make the path shorter: $1 \rightarrow 3$, $1 \rightarrow 4$, $1 \rightarrow 5$, $1 \rightarrow 6$, $2 \rightarrow 4$, $2 \rightarrow 5$, $2 \rightarrow 6$, $3 \rightarrow 5$, $3 \rightarrow 6$, $4 \rightarrow 1$, $4 \rightarrow 2$, $5 \rightarrow 1$, $5 \rightarrow 2$, $5 \rightarrow 3$, $6 \rightarrow 1$, $6 \rightarrow 2$, $6 \rightarrow 3$, $6 \rightarrow 4$ The existence of any of these edges would put multiple peering links on the path: $2 \rightarrow 1$, $3 \rightarrow 2$, $4 \rightarrow 5$, $5 \rightarrow 6$
6 edges	2 Edges	22 Edges

assume (as does most of the literature) that, in practice, our model is a good-enough match for the AS graph.

We formalize the idea of provable nonexistence in the following observations:

Observation 5.2.1. *If we have a valley-free shortest path p containing u followed by v, then, if there is at least one vertex between u and v in p, one or both of the following must be true: $u \not\rightarrow v$ or $u \not\leftarrow v$.*

Observation 5.2.2. *If we have a valley-free shortest path p that contains a bidirectional (peering) edge $a \leftrightarrow b$, then for all adjacent $u \rightarrow v$ in p such that $\{a, b\} \neq \{u, v\}$, $v \not\rightarrow u$.*

Observation 5.2.3. *Given a valley-free shortest path p without a measured bidirectional edge, then one of the following must be true:*

- *There exist sequential edges $u \rightarrow v \leftarrow w$, and the only possible bidirectional edge in the path is one of $u \leftrightarrow v$ and $v \leftrightarrow w$.*
- *The path consists entirely of $\rightarrow$ links and the only possible bidirectional link is the last edge in the path.*
- *The path consists entirely of $\leftarrow$ edges and the only possible bidirectional edge is the first edge in the path.*

All of these observations are direct consequences of p being a valley-free shortest path. If Observation 5.2.1 were violated, then there would be a shorter valley-free path than p, and if Observations 5.2.2 or 5.2.3 were violated, then p would not be a valley-free path. Observation 5.2.3 is the first rule of impossible edges that implies that all non-impossible edges may not be simultaneously present.

We can even do better than these three rules, as there is a dependence between AS paths that allows us to infer the nonexistence of edges between ASes that are not on the same path. In particular, consider two paths, both with the same root: $1 \rightarrow 2 \rightarrow 3 \leftrightarrow 4 \leftarrow 5 \leftarrow 6$, and $1 \rightarrow 7 \rightarrow 8 \leftrightarrow 9 \leftarrow 10 \leftarrow 11$. In this case, there are 110 possible directed edges in a graph of 11 vertices, and we have discovered that 12 of these edges definitely exist. The same logic that we previously used to rule out certain intra-path edges applies to each path, just like in Table 3. However, we can actually draw conclusions about the impossibility of some inter-path edges as well.

Theorem 5.2.4. *If vertex v is k hops from the measurement point p and vertex u is $k+i$ hops away from the measurement point, where $i \geq 2$, then one of the following is true:*

1. *The path containing v is of the form $p \rightarrow \ldots \rightarrow v \ldots$, the path containing u is of the form $p \rightarrow \ldots \rightarrow u \ldots$, and $v \not\rightarrow u$.*

2. *The path containing v is of the form $p \ldots \leftarrow v \ldots$, the path containing u is of the form $p \ldots \leftarrow u \ldots$, and $u \not\rightarrow v$.*

3. *The path containing v is of the form $p \rightarrow \ldots \rightarrow v \ldots$, the path containing u is of the form $p \ldots \leftarrow u \ldots$, and both $v \not\leftarrow u$ and $v \not\rightarrow u$ hold.*

4. *The path containing v is of the form $p \ldots \leftarrow v \ldots$, the path containing u is of the form $p \rightarrow \ldots \rightarrow u \ldots$, and no conclusion can be drawn about the nonexistence of any edges between u and v.*

Proof. Our proof has four cases. In our first case, the path from p containing v is of the form $p \rightarrow \ldots \rightarrow v \ldots$ and the path from p containing u is of the form $p \rightarrow \ldots \rightarrow u \ldots$, i.e. both vertices are "on the upslope" of the valley-free path. In this case, the existence of the edge $v \rightarrow u$ would make a path from p to u of $k+1$ hops, and reduce the overall length of the path with u in it by at least 1. Therefore, if both vertices are "on the upslope" of their respective paths, we can conclude that $v \not\rightarrow u$.

Symmetrically, if the path from p containing v is of the form $p \ldots \leftarrow v \ldots$ and the path from p containing u is of the form $p \ldots \leftarrow u \ldots$, (both vertices are "on the downslope") we can conclude that $v \not\leftarrow u$.

In our third case, if the path containing v has v on the upslope and the path containing u has u on the downslope, the we can conclude that $u \not\rightarrow v$, because the existence of any edge between u and v would create a shorter valley-free path from p through u to the endpoint of the measured path.

Our final case is the one in which we are stymied. If the path containing v has v on the downslope and the path containing u has u on the upslope, then we cannot conclude anything about the nonexistence of a relationship between u and v. No relationship between u and v in either direction would contradict any of our data. □

The final case of the theorem is perhaps surprising! It is intuitive to think that if the path containing v is of the form $p \ldots \leftarrow v \ldots$, the path containing u is of the form $p \rightarrow \ldots \rightarrow u \ldots$, and the distance from the measurement point to v is at least two less than the distance from the measurement point to u, then we might conclude that it must be true that $v \not\leftarrow u$. Unfortunately for our purposes, a sub-path of a shortest valley-free path is not guaranteed to itself be a shortest valley-free path.

We demonstrate this fact by considering the paths $1 \rightarrow 2 \leftarrow v \leftarrow 4 \leftarrow 5 \leftarrow 6$ and $1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow u \rightarrow 11 \rightarrow 12$. If we were to find an edge $v \rightarrow u$, we would not be able to use it to construct a shorter valley-free path from 1 to 12, because the path $1 \rightarrow 2 \leftarrow v \rightarrow u \rightarrow 11 \rightarrow 12$ is not valley-free. Similarly, we cannot use the edge $v \leftarrow u$ to create a contradiction because the path $1 \rightarrow 2 \leftarrow v \leftarrow u \rightarrow 11 \rightarrow 12$ is also not valley-free. Therefore, an edge between v and u is not forbidden by our data. The key insight in this is that, while there may exist a shorter valley-free path to u, there is no shorter way to get to vertex u through an uphill-only path, which is what is required if we are to use the latter part of the shortest valley-free path from 1 to 12. The counter-intuitive properties of valley-free shortest paths led to a paper by Curtis et al.[24] where they showed that it is possible to design topologies in which valley-free shortest paths require traffic to take tortuous paths which, when taken together, waste bandwidth in surprising ways as compared to ignoring edge type and direction and simply taking the shortest path.

Our eventual goal is to take the undirected measured AS graph, discover the directions of its edges to create the directed measured AS graph ($G = (V, E)$), and

then use all of the known paths to classify all edges into one of three categories: existing (X), impossible (I), and possible-but-unknown (U). We make two quick observations about these sets:

Observation 5.2.5. *None of the sets have any elements in common.*

$$X \cap I = X \cap U = I \cap U = \emptyset$$

Observation 5.2.6. *Each edge falls into one of the three categories, the union of all sets is the set of all edges.*

$$X \cup I \cup U = V \times V \setminus \{(v, v) | v \in V\}$$

We would obviously like $|U|$ to be as small as possible, because that is exactly the set of things which we do not know. In an ideal situation, as we added more and more measurements, $|X|$ and $|I|$ would grow and $|U|$ would shrink. Eventually (and ideally after a small amount of time), U would be the empty set, and we would know the entire graph.

Given a set of measurements, as well as the direction of the edges, or first step towards reconstructing the graph from which these measurements have come is to enumerate the sets X and U.

5.3 Enumerating X, U, and I

Enumerating the set of existing edges X from data is trivial. The edges contained in the data is exactly the set of measured edges X, and so one pass over

the data suffices to find the set X. If we know X, then thanks to Observation 5.2.6 we can find the remaining two sets by finding one of the sets and then subtracting the newly found set and X from the set of all possible edges. Let us attempt to therefore find the set of impossible edges I.

In the previous section we established a criteria for ruling certain intra-path edges impossible in Observations 5.2.1 and 5.2.2. The algorithm for finding all of these links is to exactly go through all the paths in our set of measurements and enumerate all the edges on each path which should be included in I. This algorithm is written out explicitly in Figure 22.

On each path p, the algorithm must perform a $O(|p|^2)$ process, including $O(|p|^2)$ lookups into the set of edges X. Therefore, the total time to find all impossible intra-path edges is proportional to the sum of the squares of all the path lengths. To find all impossible inter-path edges, we can use the algorithm in Figure 23.

Finding impossible inter-path edges is actually somewhat of a challenge! If we naïvely apply Theorem 5.2.4, we end up with an algorithm that runs in time proportional to, among other factors, the square of the number of measured paths. While the class $\mathcal{P}$ of polynomial-time algorithms is often used to represent the class of problems which are efficiently solvable, this is one instance where a quadratic algorithm is not efficient enough in practice. A given day can have 50 million different paths, and any computation that requires us to perform operations on more than 3 quadrillion pairs of paths will not allow us to analyze our data in a reasonable amount of time.[1]

[1] $(5 * 10^7)^2 = 2.5 * 10^{15}$ equals approximately 3 quadrillion pairs of paths. On a fast processor in 2009, a quadrillion operations will take many hours, but our analysis will require much more than one operation per path pair, and so the total expected running time of the naïve algorithm on a modern computer would be measured in weeks. That means that it would take weeks to process a single day's worth of data! Our rate of data analysis would be much slower than the rate of data generation.

```
Find-Missing-Intrapath-Edges (p, X) // path p and set of measured edges X
  I ← {}
  // This loop comes from Observation 5.2.1
  for i ← 0 ... length(p) − 1
      for j ← i + 2 ... length(p) − 1
          if (p[i], p[i + 1]) ∈ X and (p[j − 1], p[j]) ∈ X
              Add the edge (p[i], p[j]) to I //An "up" edge and an "up" edge
          elseif (p[i], p[i + 1]) ∈ X and (p[j − 1], p[j]) ∉ X
              //An "up" edge and a "down" edge
              Add the edge (p[i], p[j]) to I
              Add the edge (p[j], p[i]) to I
          elseif (p[i], p[i + 1]) ∉ X and (p[j − 1], p[j]) ∉ X
              // A "down" edge to a "down" edge
              Add the edge (p[j], p[i]) to I
  if there is a bidirectional link in the path p
      // Find all edges that would put two bidirectional edges in p (Obs. 5.2.2)
      for i ← 0 ... length(p) − 1
          if (p[i], p[i + 1]) ∈ X and (p[i + 1], p[i]) ∉ X
              Add the edge (p[i + 1], p[i]) to I
          if (p[i], p[i + 1]) ∉ X and (p[i + 1], p[i]) ∈ X
              Add the edge (p[i], p[i + 1]) to I
  else //No bidirectional link
      // Find all edges that would put a link at an illegal location (Obs. 5.2.3)
      if p consists only of up edges
          for i ← 0 ... length(path) − 2
              Add the edge (p[i + 1], p[i]) to I
      elseif p consists only of down edges
          for i ← 1 ... length(path) − 1
              Add the edge (p[i], p[i + 1]) to I
      else // the path has both up and down edges
          i ← 1
          while (p[i − 1], p[i]) ∈ X and (p[i], p[i + 1]) ∈ X
              Add the edge (p[i], p[i − 1]) to I
              i ← i + 1
          i ← i + 1
          while i < length(p)
              Add the edge (p[i], p[i + 1]) to I
              i ← i + 1
  return I
```

FIGURE 22. The algorithm to determine all forbidden/impossible intra-path edges on a shortest valley-free path

Find-Missing-Interpath-Edges (P, X, I, V, paths)
  for each path in the list of paths
    for each vertex on the path
      add to a list at that vertex the vertex's distance from the start, and
      whether all edges have been customer→provider
  for each vertex $u \in V$
    for each vertex $v \in V$
      for $i \leftarrow 0 \ldots |list[u]|$
        $(d_1, f_1) \leftarrow list[u][i]$
        $(d_2, f_2) \leftarrow list[v][i]$
        // Now we use Theorem 5.2.4
        if $f_1 \wedge f_2 \wedge abs(d_1 - d_2) \geq 2$
          $I \leftarrow I \cup \{(u, v)\}$
        elseif $f_1 \wedge \neg f_2 \wedge (d_2 - d_1) \geq 2$
          $I \leftarrow I \cup \{(u, v), (v, u)\}$
        elseif $\neg f_1 \wedge \neg f_2 \wedge abs(d_2 - d_1) \geq 2$
          $I \leftarrow I \cup \{(v, u)\}$

FIGURE 23. The algorithm to determine all forbidden/impossible inter-path edges among a set of shortest valley-free paths.

We exploit the fact that, for our data, the number of data sources, s, will always be much less than the number of vertices in the graph $|V|$ ($s \ll |V|$). Using this as an insight, we can create an algorithm that runs in $O(s * |V|^2)$ after a single pass over the input data. This is of great help, because $|p|$ is 50,000,000, but $|V|$ is between 10,000 and 30,000, and s is less than 300. To do this, we set up a list of length s for each vertex in the graph. Then, for each data source, we add to the list each vertex's distance from that data source and whether the path to that vertex consisted only of customer to provider links. We can then tell whether two vertices may be connected only by comparing their respective lists! If there does not exist a measurement point for which those two vertices fall afoul of Theorem 5.2.4, and the proposed edge is not already in I, then the edge should be in U.

5.4 The Extremal AS Graphs

Because the AS graph must necessarily contain all edges that exist in the measured AS graph (X), the graph of minimum size from which our measurements may have come is exactly the graph consisting of only those edges which our measurements measured, and no other edges. Thus, the measured directed AS graph (V, E) is the smallest graph from which our measurements might have come. In the other direction, we note that, if we remove our condition that a path can only contain one bidirectional (peering) link, then the graph of maximum size from which our measurements might have come is exactly the directed AS graph with all possible edges included $(V, X \cup U)$.

Unfortunately, we cannot remove that requirement willy-nilly. With the requirement that each path can only contain a single bidirectional link, we find that the problem of discovering the maximum AS graph, which we call MaxASGraph, is $\mathcal{NP}$-Complete.

Problem 5.4.1 (MAXASGRAPH).

INSTANCE: A directed graph $G = (V, E)$, a number k, and P, a set of directed valley-free shortest paths of G.

QUESTION: Does there exist a graph $G' = (V, E')$, with $E \subset E'$ and $|E' \setminus E| = k$, in which every path in P is a valley-free shortest path in G'?

Theorem 5.4.2. MAXASGRAPH *is $\mathcal{NP}$-Complete*

Proof. To prove $\mathcal{NP}$-Completeness, we must first show the problem is in $\mathcal{NP}$. This is easily done, because any graph can be verified, in polynomial time, to consist only of vertices from paths in P, to have more than k edges, and, via the algorithms in Figures 22 and 23 to not contradict any of the evidence from any of the paths in P.

Next, we show completeness by reducing from 3-SAT. An instance of 3-SAT consists of a set of 3-element logical or-clauses C and a set of boolean variables X_{SAT}, and we ask the question of whether there is an assignment of true and false to each of the variables in X_{SAT} that makes all the clauses of C true. Without loss of generality, we will restrict ourselves to instances of 3-SAT where no single clause contains both x and $\neg x$, as those clauses are trivially true. We transform an instance of 3-SAT into an instance of MAXASGRAPH in the following way:

We begin with a single vertex u. Then, for each boolean variable $x \in X_{SAT}$ in the 3-SAT instance, we create two vertices (v_x and $v_{\neg x}$) and a path p_x, consisting of $v_x \rightarrow u \leftarrow v_{\neg x}$. Note that vertex u is universal and is the same u for all v_x. Then, for each of the three-variable clauses $c = (x \vee y \vee z) \in C$, we create three vertices and six paths.

The three vertices we create, we call $v_{c,x}$, $v_{c,y}$, and $v_{c,z}$. We then create three paths to ensure that, for the whole clause c, there is only one possible missing link. These paths are $v_{c,x} \rightarrow u \leftarrow v_{c,y}$, $v_{c,x} \rightarrow u \leftarrow v_{c,z}$, and $v_{c,z} \rightarrow u \leftarrow v_{c,y}$. Note that if there is also an edge $v_{c,x} \leftarrow u$, then none of $v_{c,y} \leftarrow u$ and $v_{c,z} \leftarrow u$ may exist due to

Observation 5.2.2. Our logic is symmetric, and holds no matter which edge we consider adding along these paths.

We also create three paths to link each of our variables with the clause c containing it. If x is a literal in clause c then we create the path $v_{\neg x} \rightarrow u \leftarrow v_{c,x}$, and if $\neg x$ is a literal in clause c, then we create the path $v_x \rightarrow u \leftarrow v_{c,\neg x}$. If we again keep Observation 5.2.2 in mind and take the two paths $v_x \rightarrow u \leftarrow v_{\neg x}$ and $v_{\neg x} \rightarrow u \leftarrow v_{c,x}$, we see that if the edge $v_{\neg x} \leftarrow u$ exists, then neither of $v_x \leftarrow u$ and $u \rightarrow v_{c,x}$ may exist. We call this set of paths P_{SAT}. The vertices along these paths will serve as the vertex set V_{SAT} for our MAXASGRAPH instance.

The set of edges E' is the union of all edges contained on all paths in P_{SAT} (which we call E_{SAT}) with all possible edges whose existence is not conditional on the nonexistence of another edge. To find the set of possible edges, we calculate X, I, and U using the algorithms from Figures 22 and 23. E' is then equal to $E_{SAT} \cup \{(w,v) \in U | (v,w) \notin E_{SAT}\}$. We have now constructed, in polynomial time, a set of paths P_{SAT}, and a graph $G' = (V_{SAT}, E')$ in which some edges may be missing, but the only edges missing are edges are of the form $u \rightarrow v$ with $v \rightarrow u \in E_{SAT}$. As a shorthand, we say that if both $u \rightarrow v$ and $v \rightarrow u$ exist in a graph, then u and v are connected by a bidirectional link.

As input to our putative MAXASGRAPH solver, we use our constructed P_{SAT}, $G' = (V_{SAT}, E')$, and k equal to the number of clauses plus the number of variables ($k = |C| + |X_{SAT}|$). Now we prove that there exists a satisfying 3-SAT assignment if and only if there may be at least k edges added to E', with all paths in P_{SAT} remaining shortest valley-free paths.

A satisfying assignment to 3-SAT implies the existence of $k = |X_{SAT}| + |C|$ edges by direct construction. We know by our construction method that the only edges which may be missing are those edges which may make a link to u into a bidirectional link. We assign bidirectionality from the satisfying assignment in the

following way: If $x \in X_{SAT}$ is set to true, then add the edge $v_x \leftarrow u$ to the set of possible edges. If $x \in X$ is set to false, then add the edge $v_{\neg x} \leftarrow u$ to the set of possible edges. We know that each clause c has at least one variable x (or $\neg x$) which evaluates to make the clause true. Arbitrarily choose one of them and add the edge $u \rightarrow v_{c,x}$ (or $u \rightarrow v_{c,\neg x}$) to our set of possible edges. At the end of this, we have added one edge per clause, and one edge per variable. Furthermore, we know that, thanks to our construction method, we have not added more than one bidirectional edge to any path. Thus, we have exactly discovered $|C| + |X_{SAT}|$ edges which may all exist simultaneously in our graph without contradicting any of the paths in P_{SAT}.

The possibility of adding $k = |C| + |X_{SAT}|$ edges to E' implies a satisfying assignment to 3-SAT by much the same logic. First we note that, by construction, only the paths in P_{SAT} may be missing any edges. Therefore, the newfound edges must come solely from paths in P_{SAT}. Furthermore, we note that the path in P_{SAT} for each variable $x \in X_{SAT}$ may contain only one missing edge, so there can be no more than $|X_{SAT}|$ edges on paths corresponding to variables. Next we note that the three paths containing only $\{v_{c,x} | x \in c\}$ for each clause $c \in C$ may, by construction, only be missing one bidirectional link among the three of them. Therefore, there is at most one missing edge for each clause c, for a maximal total contribution of edges from these paths of $|C|$. Finally, we note that, again by construction, it is impossible for both $v_{\neg x} \leftarrow u$ and $v_{c,x} \leftarrow u$ to be simultaneously present. Which implies that our clause's truth values (a clause is true if it contains a bidirectional link) and our variable assignments (x is true if $v_x \leftarrow u$ exists and false if $v_{\neg x} \leftarrow u$ exists) be consistent. Therefore, if there are $|C| + |X_{SAT}|$ missing edges, then $|X_{SAT}|$ of them come solely from paths belonging to only variables and $|C|$ of them come from paths corresponding only to clauses, and the satisfying assignment is derived from the set of $|C| + |X_{SAT}|$ noncontradictory edges in the following manner: For all $x \in X$, if $v_{\neg x} \leftarrow u$ is in the set then assign false to x, otherwise

$v_x \leftarrow u$ must be in the set and we should assign true to x. In this manner, we guarantee that every clause can be satisfied.

Now we know that MAXASGRAPH is in $\mathcal{NP}$, and 3-SAT is reducible to it in polynomial time, and therefore MAXASGRAPH is $\mathcal{NP}$-complete. □

The eventual consequence of this is that when adding peering links, we must be careful to not create inconsistencies. Thus, our set of unknown edges U is actually two disjoint sets $U = N \cup C$, where N is the set of links that do not conflict with anything else, while C is the set of links that may conflict with other edges. The size of the maximum AS graph is therefore somewhere between $|X| + |N|$ and $|X| + |N| + |C| = |X| + |U|$.

These extremal graphs are not necessarily very informative, however, as it is quite unlikely that our measurements contain all edges (the minimum AS graph), and it is also highly unlikely that all the edges that have not yet been ruled impossible actually exist (the maximum AS graph).

5.5 Counting the Number of Missing Edges

We can count the number of missing edges in two ways. In each way we use statistical estimation techniques, but our assumptions are subtly different. In the first, we note that one of the things which we are not unsure about is the degree of the vertices from which our measurements are taken. Therefore, if we assume that the degree of our measurement points well represents the degree distribution of the complete AS graph, we then find their average degree, multiply that by the number of vertices in the graph, and divide by two to find the total number of edges we should expect to see. The difference between the total expected and the total we do see is then the number of edges from the set P of possible edges that we should add to the graph. In this case, on 14 April 2008 we find that the average degree of our

229 measurement points is 220.432, and therefore the total number of edges on our 28,150 measured vertices is estimated to be 3,102,580. This, together with the 68,567 measured edges, implies that we are missing an astonishing 98% of the AS graph in our measurements. This estimate of the amount of missing edges is out of line with every other estimate found in the literature[60, 55], and so we are forced to conclude that either every other researcher who has studied this problem is very wrong, or that the degree distribution of our measurement points is not representative of the AS graph as a whole. The second hypothesis is actually rather likely, because Route Views has actively and successfully sought out participation from large, central ISPs. Large ISPs tend to have more customers, which would also increase their degree. If large ISPs were over-represented in our sample, it would lead to exactly the phenomenon we see. This is an example of a phenomenon studied by Lee et al. who note that different sampling methods on the same graph can lead to extremely different conclusions about the graph's structure[46].

Our second method of estimating the number of missing edges is more refined. Population biologists measure the size of an unknown population by capturing some subpopulation, tagging and releasing them, and then analyzing the recapture rate. The recaptured percentage, divided by the total captured number, yields the total expected population. This method is called capture/recapture, and we can do something quite like it with our data. This approach is advocated by Flaxman and Vera, where they first used it to good effect on the AS graph in an attempt to estimate the degree distribution of the complete AS graph[32].

Flaxman and Vera construct an **estimator** for graph degree distributions when the graph data is collected via the measurement of shortest paths. An estimator is a function which takes in the measured data and attempts to reconstruct the object from which the data has come. They investigate several estimators over several families of graphs, and use experimental techniques to discover the best estimator in

all cases. They neglect the valley-free aspect of the underlying graph, but their work is the most apposite available, and therefore it is the base upon which we will do our work.

To perform a naïve capture/recapture on the AS graph, we take one half of the measurement points, and note what edges can be seen from those points ($E_{control}$). We then take the other half of the measurement points, find what edges can be seen from them (E_{test}), and find the intersection between the two observed edge sets. Using this, we can infer the total number of edges we expect to see in the graph with the equation:

$$\text{estimate} = \frac{|E_{control}| * | E_{test}|}{|E_{control} \cap E_{test}|}$$

The capture/recapture method both puts us on a slightly firmer statistical ground than our first method, and can also provide confirmation of the validity of the first method.

Doing this process, we see that our measurements of the AS graph are, as expected, not complete. According to Figure 24, we seem to be consistently missing between 2% and 5% of the edges of the AS graph. This stands in stark contrast with the 98% estimate from the first method, and, in turn, provides evidence that the measurement points of RIPE and Route Views are not uniformly random.

Next we investigate the distribution of our answers. In Figure 25, we can see the results of running dozens of capture/recapture trials to count both the number of edges and the number of vertices. In particular, note the axes. The X-axis is the number of vertices a given trial tells us exist, and the Y-axis is the number of edges. As can be seen from the graph, we are extremely confident of the number of vertices — the X range is less than 1 vertex. Unfortunately, our conclusion about the number of edges can vary by over 500, or 1% of the total, indicating that perhaps we need a new, better method of estimating how many edges may be missing from a given sample. The measured AS graph from RIPE and Route Views was

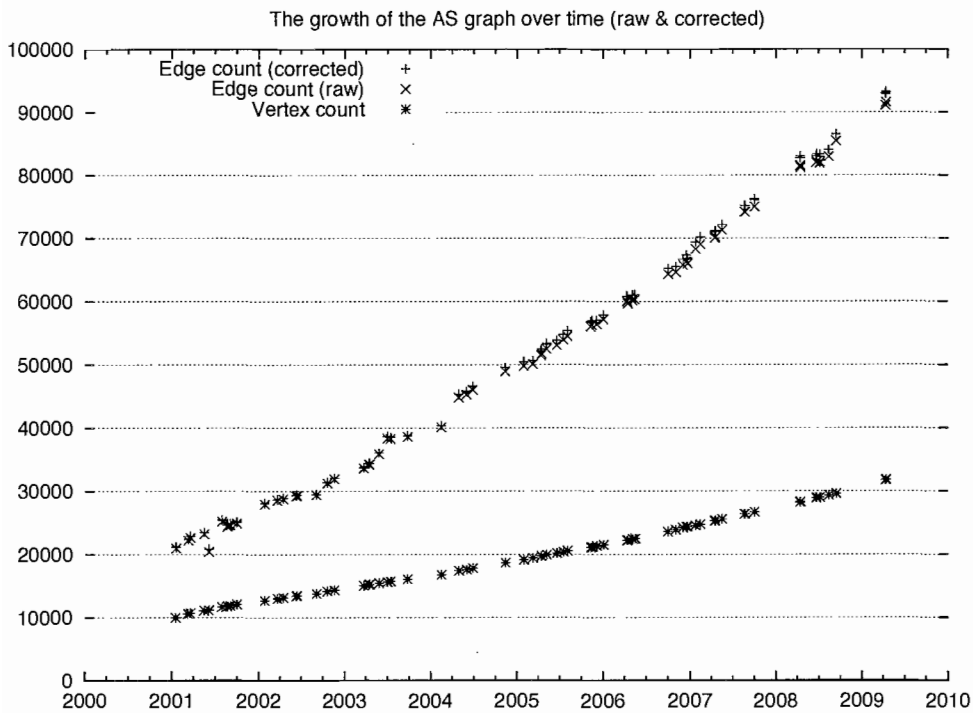


FIGURE 24. Our corrected edge count

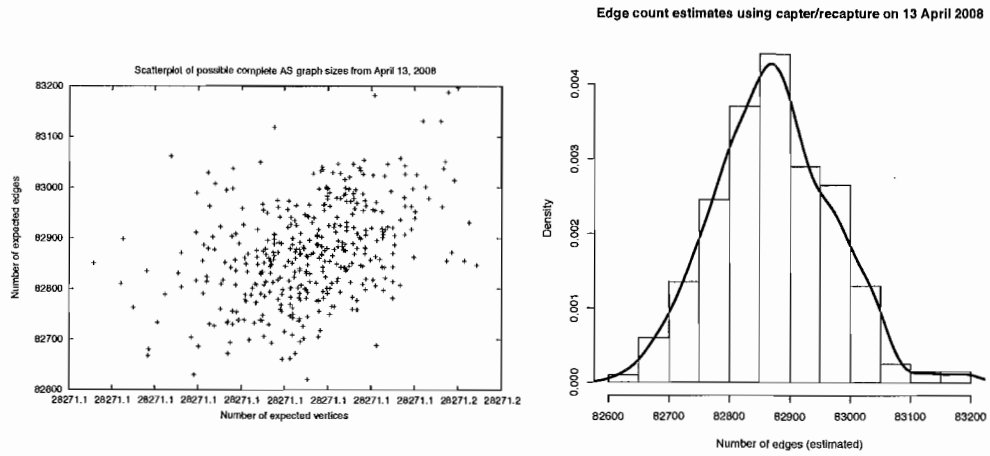


FIGURE 25. Scatter plot and distribution of the expected missing number of vertices and edges as proposed by naïve capture/recapture.

determined, via direct survey and through modeling and simulation, to be missing between 10% and 20% of the edges of the complete AS graph[27, 55, 60], so that is what we will be trying to achieve with our estimation techniques.

The process we must use to not be misled by measurement error is not as simple as our naïve method, as it is exactly the network structure which determines whether an edge will be sighted, and so our measurements of network structure are not independent — edges do not randomly mix with the population of edges, but remain where they are in the structure of the graph. Armed with the insight that capture/recapture may be of use, but that we must use it very carefully, we take another look at the results of Flaxman and Vera.

We now sketch out the estimation algorithm developed by Flaxman and Vera. We call the underlying graph from which measurements are taken G, and let G_s be the subgraph of G measured via shortest paths from vertex s. We use $N_s(u)$ to denote the set of neighbors of a vertex u in G_s, and analogically define G_t and $N_t(u)$ for vertex $t \neq s$. We now define our estimate, with respect to s and t, of node degree

for a vertex u of G to be

$$\widehat{deg}_{s,t}(u) = \begin{cases} \frac{|N_s(u)| \times |N_t(u)|}{|N_s(u) \cap N_t(u)|} & \text{if } |N_s(u) \cap N_t(u)| > 2 \\ \infty & \text{otherwise} \end{cases}$$

If we take the set of possible (s,t) pairs to be all distinct pairs of monitors $\{(s_1,t_1),\ldots(s_k,t_k)\}$, then we create the following as our overall degree estimator for $v \in V$:

$$\widehat{deg}(u) = \text{median}(\{\widehat{deg}_{s_i,t_i}(u) \neq \infty\})$$

This estimator was measured to work over a wide range of models, including models of the undirected AS graph, so we will be using this predictor in a domain where it has been measured to perform well. The only wrinkle is that a close examination of the predictor reveals that it only makes predictions on vertices with a degree greater than two — if an AS only has one or two neighbors, then this method ignores it. For comparison, our naïve algorithm and came to the conclusion that there were $82,881 \pm 87$ edges in the complete AS graph on 13 April 2008. We compare it to this newer, more refined method, which comes to the conclusion that there are $88,601$ edges on that same day (no confidence interval is provided with the second method). The measured AS graph on that day consisted of $28,272$ vertices and $81,554$ edges.

So our overall strategy is: we take AS graphs that already have their edge types annotated and extend them by the number of edges that were surveyed to be missing. We randomly choose edges from the sets N and C with the caveat that we attempt to minimize the conflicting edges we take from the set C. In doing so, we can generate a candidate complete AS graph that is not just possible, but also plausible. This predictor gives us even more information, however. Using the predictor, we not only have an estimate for the overall number of edges, we actually have an estimate of the expected degree in the undirected AS graph. Therefore, our

method is to generate random edges from N and C in an effort to ensure that each vertex has its predicted degree.

5.6 Sampling from the Set of Possible AS Graphs

Once we give up on worst-case analysis, we turn towards average-case analysis, using the mathematics of expectation. In this section we describe three approaches to solving our problem, which we broadly classify as lucky, frequentist, and bayesian.

We begin with the lucky method. In this method, when we analyze members of the family of possible AS graphs, we find that they are all so alike in result that no further statistical analysis of our output is required — all trends are immediately clear and unambiguous. Note that we have little reason to expect that this method will work, but its extreme ease of use dictates that we at least try it when performing analyses to ensure that any further refinement is actually necessary and of use. In the next chapter, we find that, surprisingly, the lucky method actually does work in practice for the problems we care about. Therefore, the following suggestions are largely of theoretical use, in case later analyses need greater precision.

For the frequentist method, our goal is to discover the most likely graph from which our data might have come, and to treat it as the graph under consideration. Of course, given the size of our search space in which we hope to discover the most probable graph, and given that we do not know of any broad strategy for searching that could enable a binary-search strategy, finding the most probable AS graph lies firmly in the domain of heuristic search.

To perform a heuristic search, we first require a heuristic. In the design of our heuristic, we will be guided by the insight that, while our graph may be missing some edges, no edges are missing between our measurement points and their

neighbors. Therefore, we have a known-and-correct degree distribution for our measurement points. If we assume that our measurement points are chosen independently (an assumption we will revisit later), then we can use statistical tests to measure the likelihood that our (presumably independent) sample came from the given graph. Using this measurement of likelihood, we use simulated annealing or some other high-dimensional search algorithm to discover a graph of high likelihood. Then, once we find the graph with the highest likelihood, we analyze that one graph.

The next approach, the bayesian method, seeks not the graph of highest likelihood, but instead of set of independently gathered graphs, each with an associated likelihood. We then perform any and all tests on each of the entire set of graphs. In this manner, we can discover both an expected value, as well as find a standard deviation away from that expected value.

Having loosely defined the landscape of methods, we now delve more deeply into the frequentist and bayesian methods — the two methods which actually require us to do something.

5.6.1 Frequentist

In the frequentist approach, we search for the most likely graph out of the set of possible graphs, and then we analyze just that one graph. In doing this, we treat a single graph as a stand-in for the set of possible graphs. To use this method, we need some way of finding possible graphs (Section 5.3), we need a high-dimensional search algorithm, and we need some way determining which graphs are more likely.

Both the frequentist and bayesian method depend on the idea that, while we may not have knowledge of the entire graph, we do have some global knowledge from our measurements. One example global property is that we know the degree of each of the measurement points — because each measurement point reports all of its shortest paths, we can be sure that the data contains all of the links between a

measurement point and its neighbors. Then, if we assume that the degree distribution of the sampled points is representative of the degree distribution as a whole, we can compare the two distributions using a statistical test. We may not want to use degree distribution, however, as we are constructing our candidate graph using a predictor for exactly that. As another example, we may be certain of the distribution of path shortest valley-free paths from each of our measurement points. We could then compare that distribution with the distribution of shortest valley-free path lengths in the candidate graph. This method generalizes to any global property that may be unambiguously deduced from these measurement points, but we always need to assume that our set of measurements is representative in some way.

We now have an example of why it is better to be lucky than good — both the frequentist and the bayesian methods will, when put in place, have an extra assumption, that we will not need if we are lucky. To determine which graphs might be more likely, we turn to the Kolmogorov-Smirnov test (K-S test), which is a non-parametric statistical test (meaning it does not assume an underlying normal distribution) designed to assess the likelihood that a given sample came from a given distribution. The K-S test simply calculates $\max |D_1 - D_2|$, where D_1 and D_2 are two distributions with the same domain. This test was used by Haddadi et al. to compare different AS topology generators[38], which means that is well-applies to this domain. We use the test to check the likelihood of the degree distribution of the measurement vertices (where the full degree is known) having come from the degree distribution of the full graph.

Our full method for finding the most likely graph is:

1. Generate an initial graph by estimating the number of missing edges and adding in that many edges from P.

2. Refine that graph to be more likely through repeated rounds of heuristic search (removing edges originally from P and/or adding in new edges from P) and the K-S test.

Note that this method does not have a well-defined end condition! This is intentional. Choosing when to stop an on-line heuristic search is an art, not a science. In our case, we choose to bound the number of rounds, but another equally valid choice might be to stop once little extra improvement is seen, or to stop after a certain amount of wall-clock time has passed.

Nonetheless, once our method has terminated is complete, we should have in hand one of the most likely AS graphs from which our data might have come, and then we pass this graph of for later input into our analysis routine.

5.6.2 Bayesian

Our bayesian method is easier to implement than our frequentist method, but it requires us to do more work farther down the road. In particular, the bayesian interpretation of the concept of expected value has to do with strength of belief. The frequentist worldview rejects as nonsensical the weatherman who proclaims a 40% chance of rain on a given day, as there can never be multiple independent trials of a given day, while the bayesian worldview interprets with weatherman as stating that they have a 40% belief in the premise "it will rain on this day".

We use this idea of "belief strength" in the following bayesian-reasoning based method:

1. Generate a graph by estimating the number of missing edges and adding in that many edges from P.

2. Estimate the likelihood of that graph using the K-S test.

3. Perform all subsequent analysis, weighting the results by the likelihood of the input graph, using the same estimates of likelihood that we used in the frequentist method.

4. Repeat until a large enough sample has been processed.

5. Report a weighted average of all analyzed samples.

Again, this method does not have a well-defined termination condition. Much like with the frequentist method, deciding when to stop sampling from a space is more an art than a science.

In this way, we both capture a larger fraction of the space of possible graphs, and we can derive both an expected value and a standard deviation from the repeated, weighted results of multiple analyses. In this sense, the bayesian algorithm gives us more data, because it provides not just with an answer, but also with a confidence interval. The downside is that it also requires us to only perform analyses which result in an output amenable finding a mean and standard deviation — which means that graph analysis techniques which produce something other than a scalar values are generally not suitable for this method.

5.7 Summary

Using an initial graph from CAIDA, and then the algorithm from Figure 8, we can assign direction to each of the measured edges. Then using techniques from statistics and machine learning, we attempt to derive an accurate picture not of the measured AS graph, but of the complete directed AS graph from which our measurements were taken.

Along the way to developing these techniques, we developed two algorithms for enumerating all possible edges which might have been missed by our measurements. We also proved that finding the maximum size AS graph is an $\mathcal{NP}$-complete problem. Fortunately for us, the complete directed AS graph is highly unlikely to be equal to the maximum AS graph. We formalized this knowledge that not all graphs are equally likely into two algorithms based on the two major schools of statistical reasoning[2].

By using these techniques together it becomes possible, for perhaps the first time, to accurately assess and describe the complete directed AS graph of the Internet. In the next chapter we use our these techniques to set up an analysis pipeline for AS graph data. We first try the "lucky" method, and only engage the greater sophistication of the frequentist and bayesian methods if we are not lucky.

[2]It is interesting to note that while bayesians form a majority in the machine learning community, they are the minority in most mathematical statistics programs. It turns out that bayesian analysis performs really well at certain machine learning problems such as, most famously, determining whether a message is spam or not[61], and so the bayesian viewpoint has gained much ground due to its utility. The debate between the two camps has gone on for decades.

CHAPTER VI

THE EVOLUTION OF THE AS GRAPH

In which we analyze a series of network measurements in an effort to understand how the AS graph has changed over time, and in doing so we apply the methods of the previous chapters

In this chapter, we can finally start discovering how the AS graph has evolved over time. To do this, we apply the pipeline developed in the previous chapter with metrics that have been of interest in the existing literature. In order to decide what to measure, and recalling the breakdown of graph analysis techniques in Section 2.5, we begin with a survey of what has been measured already.

6.1 Previous Analyses of the AS Graph

Previous analyses of the AS graph have measured the changing size of the measured AS graph, found the degree of the power-law degree-distribution of the measured AS graph for a given date, found the clustering coefficient for a given date, and have analyzed the eigenvalues of the graph matrix. These techniques have become part of the standard arsenal of algorithms to throw at a problem in the general area of what has come to be called **graph mining** in some communities or **network analysis** in others. We will only detail the techniques from this emerging

field that are immediately useful to us, and the reader interested in the generic field of graph mining is referred to the book by Brandes and Erlebach[14] or the survey article by Chakrabarti and Faloutsos[20]. One important thing to note is that, with only one or two exceptions, the graph analyzed has been the measured AS graph. Researchers almost exclusively have been analyzing their measurements (the measured AS graph, sometimes directed, sometimes undirected), rather than the object from which the measurements have come (the complete directed AS graph).

Furthermore, many of the analysis techniques were taken from analysis methods designed for undirected graphs and blithely used on the undirected AS graph. Previous analyses have come to the conclusion that the AS graph is growing (indeed, the growth study by Huston[39] was one of the main drivers for the introduction of 32-bit AS numbers), that the AS graph is a small world graph, and that the AS graph is a power-law graph. We re-examine the power law claim in a subsequent section.

6.2 Network Size

The first, and easiest, question to answer about the AS graph is "How many vertices does it have?" In Section 5.5, and Figure 25, we found experimental validation for our previous assumption that our measurements do contain all the vertices of the AS graph. Therefore, to measure the number of vertexes over time, it suffices to merely count the measured vertices over time. Our measurements provide exactly the same curve as those of Huston[39], which provides further support for the basic correctness of our processing pipeline implementation.

As we can see in Figures 25 and 26, the number of vertices in the AS graph has been steadily growing over time, from an initial value of $10,000$ in the year 2000, to more than $30,000$ today. The growth is quite steady, despite the dot-com boom,

bust, and subsequent recovery. In this, we see that the Internet infrastructure seems to always be growing. It is possible that the state of the surrounding industry and economy does affect the growth rate of the AS graph, but it does not seem to have ever caused the AS graph to stop growing.

6.3 Size of the Network Core

When measuring the AS graph, we created an estimator for vertex degree. Recall that our predictor only worked on vertices with degree greater than two. If a vertex has a degree of two or less, then it is highly likely that the AS it represents does not play a central role in the Internet infrastructure. To measure the size of the core, we repeatedly remove all vertices of degree one until all remaining vertices have degree two or higher. We call this the core of the network. Because we repeatedly removed all vertices of degree one, we might also call this the 1-core of the network.

We also calculate the 2-core by repeatedly removing all vertices of degree two or less. Now we can compare the 1-core to the 2-core to the complete AS graph, and we can see the growth of each over time. In Figure 26, we can see that the AS graph, the 1-core, and the 2-core have all been growing over time, but the size of the 2-core is less than half of the size of the complete AS graph, which means that most ASes are not part of the 2-core.

6.4 Degree Distribution

The degree distribution of a graph is a discrete probability distribution X, where $X[i]$ is equal to the number of vertices in the graph with degree i. Past studies of this property have concluded that the AS graph degree distribution was a power-law degree distribution. Then, more recently, it was shown that graphs sampled via

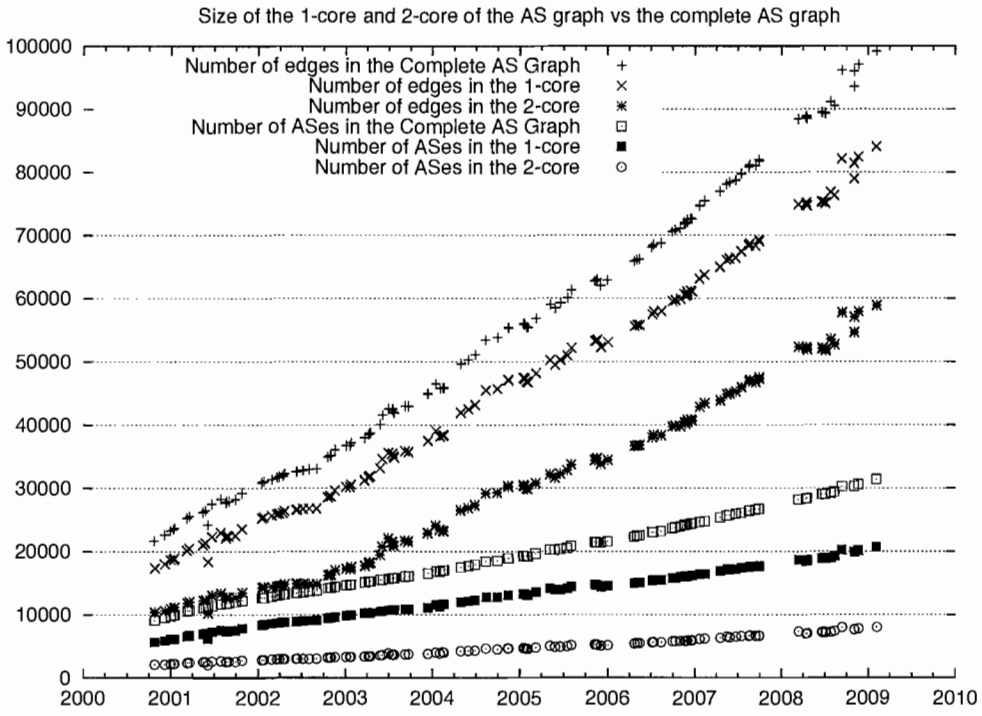


FIGURE 26. The size of the network core over time, for the both 1-core and the 2-core.

shortest paths may evince a degree distribution in their samples when no such distribution exists in the underlying graph[46]. So now things are a bit confused. To resolve this confusion, in Figures 27 through 31 we look at the degree distribution of a sampling of candidate complete AS graphs and find that our candidate completions (which attempt to remove that bias) still evince a distribution that is "heavy-tailed".

When a distribution is power law, its graph evinces an exponential decay. In general, all power law functions are in $\Theta(x^k)$, which means that, if we would like to turn our degree distribution into a graph statistic, we can find the best-fit k for each day and then graph k over time. Unfortunately, this would be, in a very real sense, overfitting our data to a perceived model. As Willinger et al. noted, measurements of the Internet are of extremely high variability, and while any and all reasonable models of Internet connectivity suggest a heavy-tailed distribution, they do not specifically suggest a power-law distribution.[73]. To see this principle in action, we need only look at the best-fit line of the graph to see that fitting the data to a straight line on a log-log plot is almost certainly the wrong thing to do. Modeling the AS graph as strictly a power-law graph systemically mis-represents its structure, and almost all generative models of the AS graph create a power-law degree distribution[77]. In particular, note the kink in the middle of the graph, around the degree of 64 in Figure 30[1]. No model of the AS graph yet known will produce such a degree distribution, and pretending that the kink doesn't exist because it contradicts a clean model is the very opposite of the scientific method. Therefore, we find that there are significant aspects of AS graph structure which are being ignored in current power-law models.

[1]It is suggested by Flaxman and Vera that the kink in the graph around the degree of 64 is a phenomenon associated with the availability of routers with 64 ports, but not 65 or more[32]. In this model, the marginal cost of adding a 65th neighbor is much greater than the marginal cost of adding a 64th, as a new hardware purchase is required. This idea is intriguing, but should be regarded as unproven.

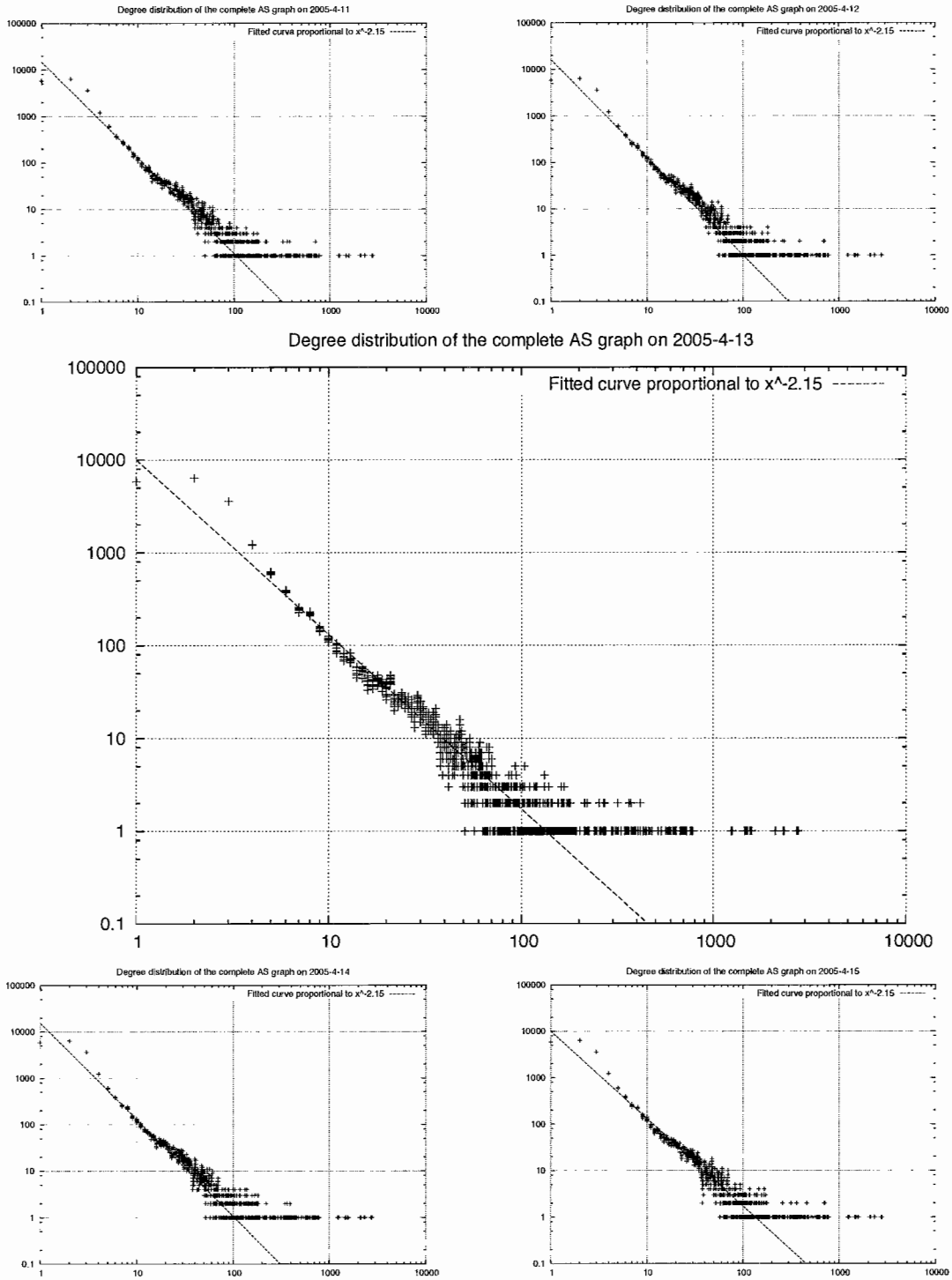


FIGURE 27. The degree distribution of different candidate complete AS graphs on 13 April 2005 (logarithmic scale), along with the same graph for the two days prior, and the two days afterwards.

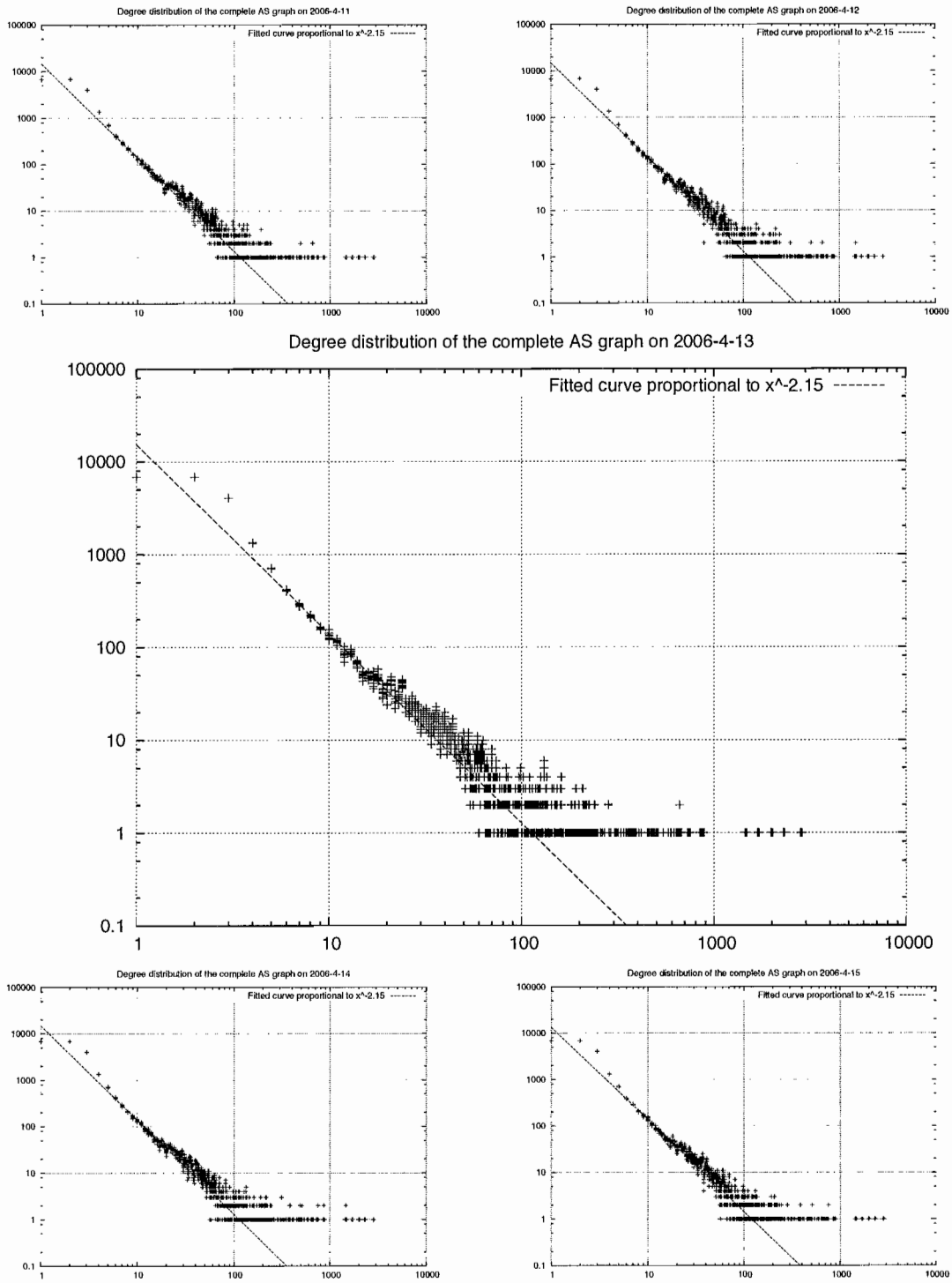


FIGURE 28. The degree distribution of different candidate complete AS graphs on 13 April 2006 (logarithmic scale), along with the same graph for the two days prior, and the two days afterwards.

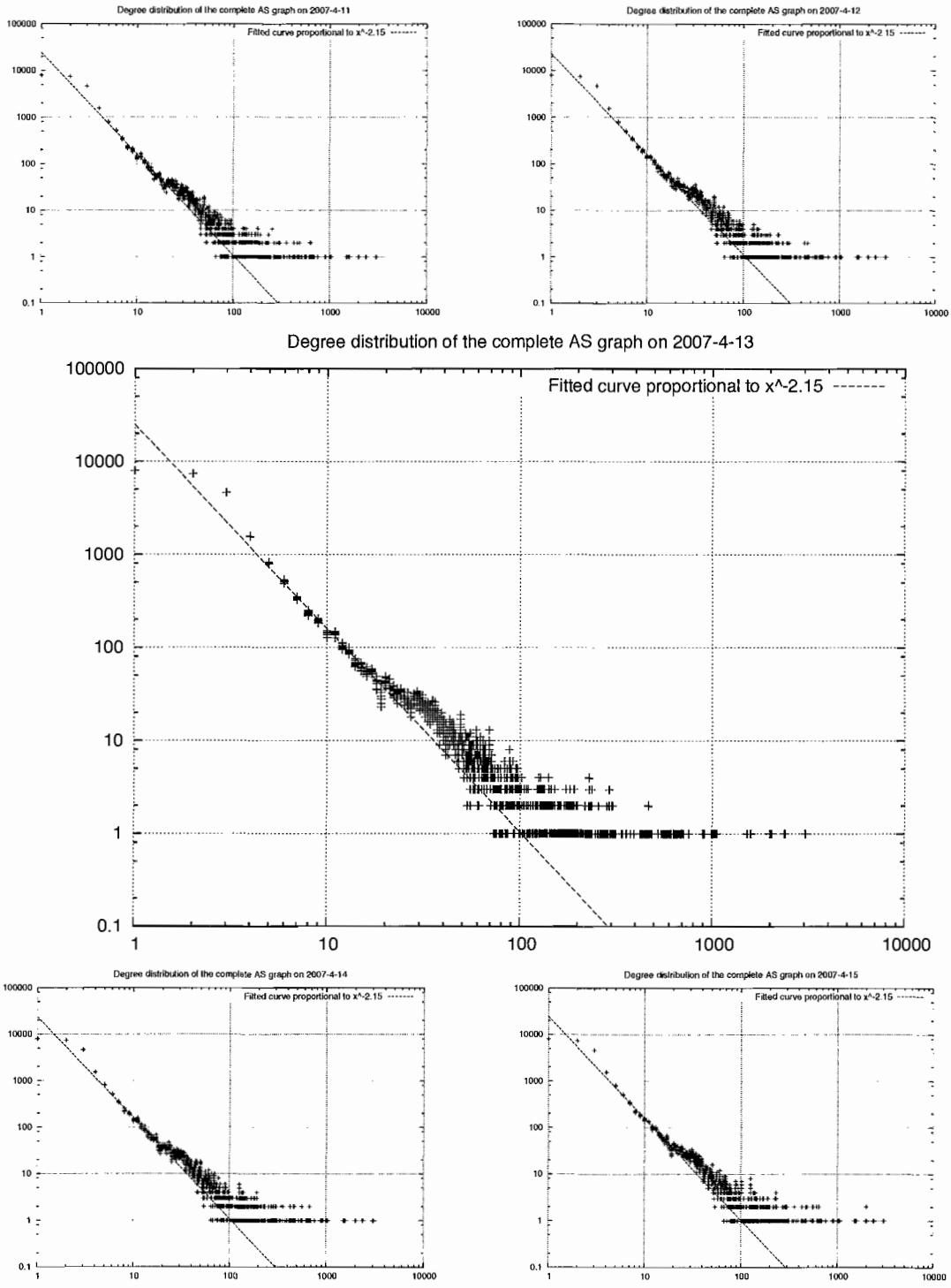


FIGURE 29. The degree distribution of different candidate complete AS graphs on 13 April 2007 (logarithmic scale), along with the same graph for the two days prior, and the two days afterwards.

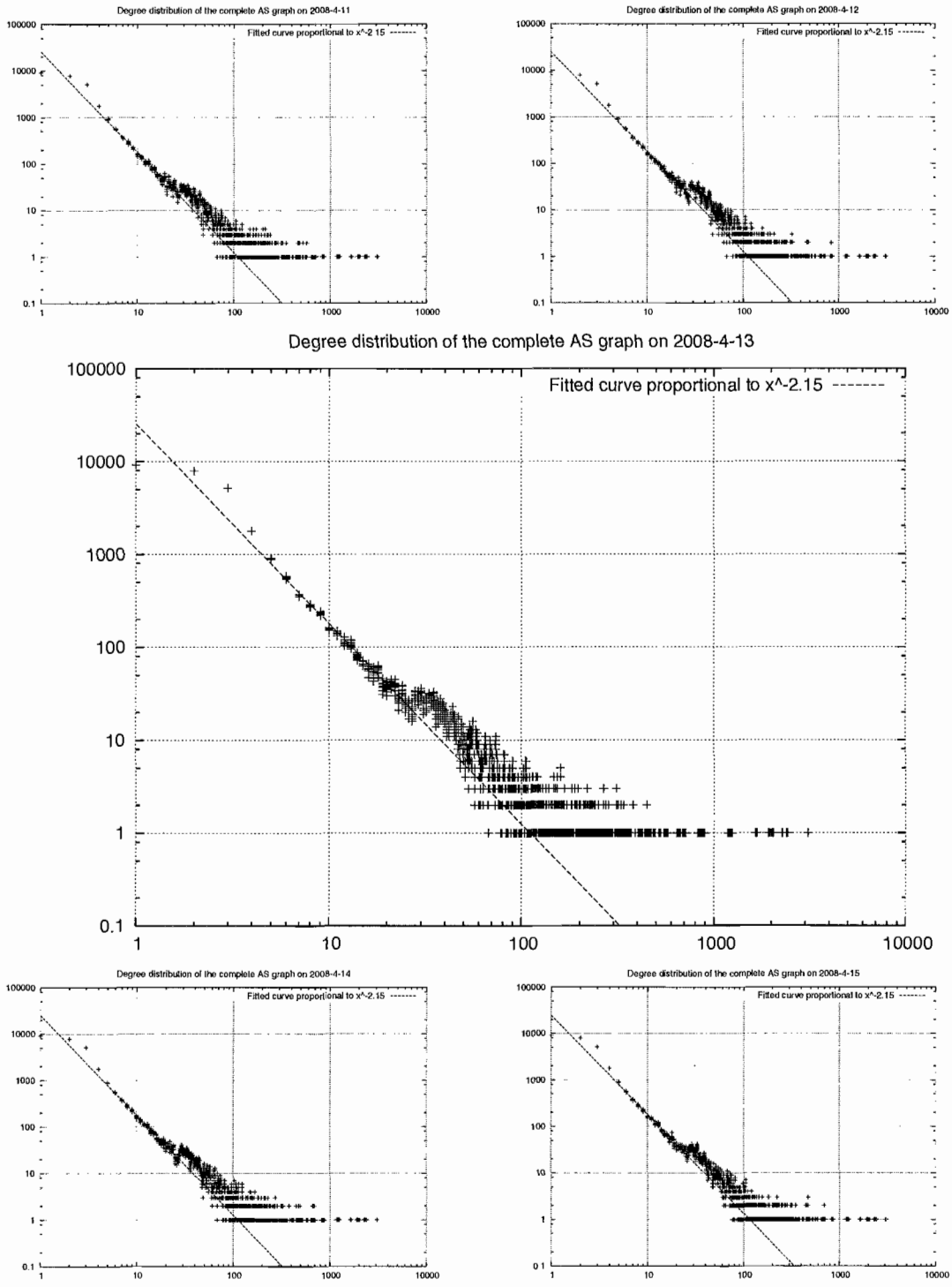


FIGURE 30. The degree distribution of different candidate complete AS graphs on 13 April 2008 (logarithmic scale), along with the same graph for the two days prior, and the two days afterwards.

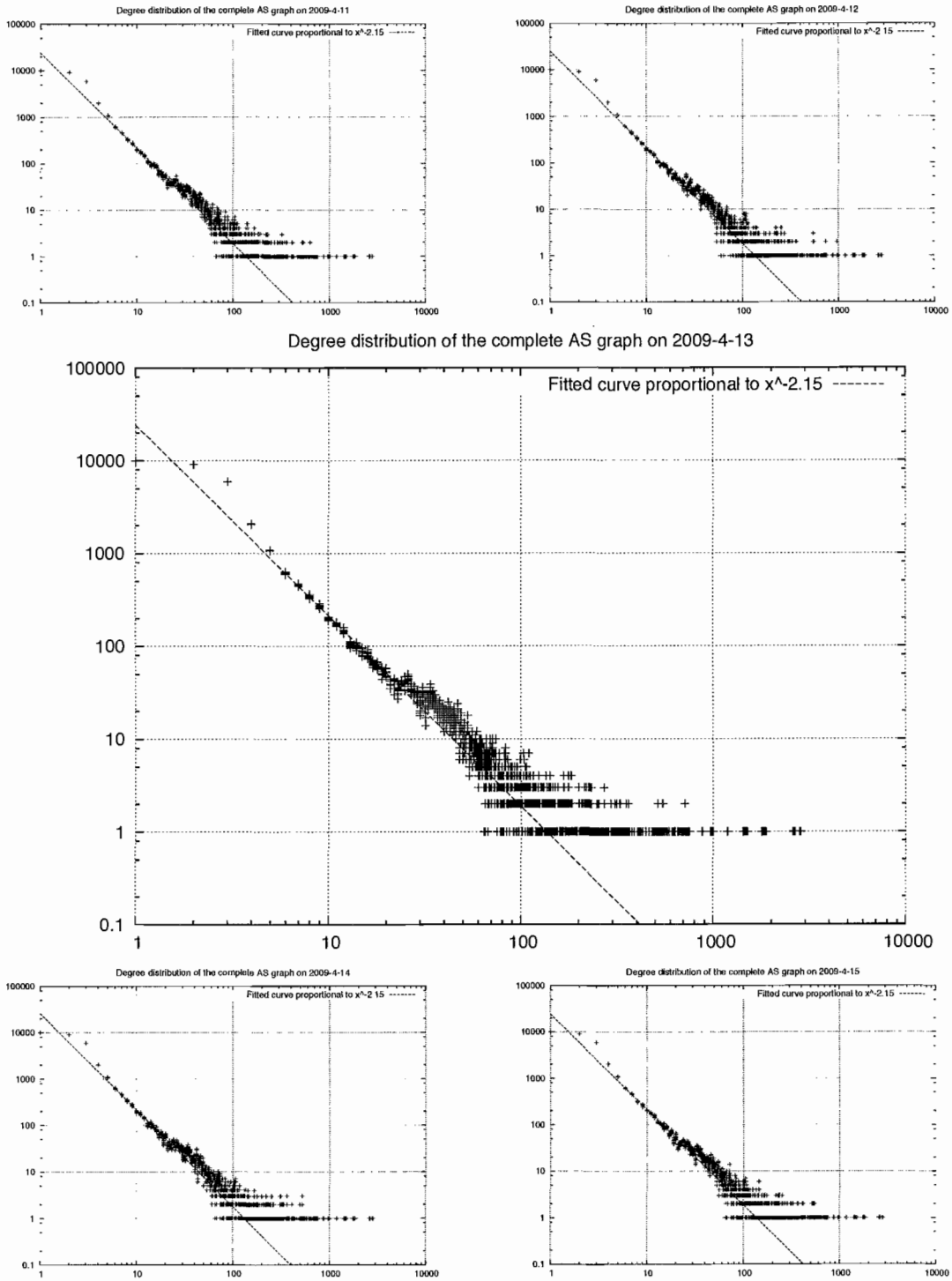


FIGURE 31. The degree distribution of different candidate complete AS graphs on 13 April 2009 (logarithmic scale), along with the same graph for the two days prior, and the two days afterwards.

6.5 Clustering Coefficient

The clustering coefficient is an attempt to find out how many of a vertexes neighbors are neighbors of each other. Graphs with a high clustering coefficient will tend to have a lot of triangles, as well as a high degree of fault tolerance. The clustering coefficient can be misleading, as all bipartite graphs have a clustering coefficient of 0, but for graphs which are not bipartite, the clustering coefficient serves as a common way of trying to asses the intuitive notion of how "clustered" or "clumpy" the vertices of a graph are arranged.

The clustering coefficient of a vertex is defined to be the number of neighbor-neighbor links which can be found around the given vertex, divided by the number of neighbor-neighbor links which could possibly exist. If we define $N(v)$ to be the set of vertices which are adjacent to v in some graph $G = (V, E)$, then the clustering coefficient is $\frac{|\{(u,w)|u,w \in N(v) \wedge (u,w) \in E\}|}{|N(v)|*(|N(v)|-1)}$ The clustering coefficient of a graph $G = (V, E)$ is the average clustering coefficient over all vertices in the graph, and may calculated as in Figure 32. This algorithm for calculating the clustering coefficient differs from the one introduced by Watts and Strogatz[71], because that original definition was ambiguous about how to treat vertices of degree 1 and 0. Instead, we use the definition of clustering coefficient that Bu and Towsley used to analyze the AS graph[19].

Our results, which can be seen in Figure 33 mirror the results of Bu and Towsley on the days in which our data overlaps, and demonstrate that, despite the rapid grown in size, the clustering coefficient of the AS graph has grown from 0.47 to .50 and then come back to a value of 0.47.

```
Vertex-Clustering-Coefficient (G = (V, E), v)
  if |neighbors(v)| ≤ 1
      return 0
  count ← 0
  for u ∈ neighbors(v)for w ∈ neighbors(v) \ {u}
          if w ∈ neighbors(u)
              count ← count + 1
  return count / (|neighbors(v)|*(|neighbors(v)|−1))

Graph-Clustering-Coefficient (G = (V, E))
  list ← 0
  for v ∈ V
      append(list, Vertex-Clustering-Coefficient(G, v))
  return average(list)
```

FIGURE 32. The algorithm for calculating the clustering coefficient of a graph, using the definition of Bu and Towsley.

6.6 Characteristic Path Length

Characteristic path length of a graph is the average shortest path length. It is meant to be a robust version of graph diameter that is less vulnerable to being thrown off by the existence of a single long path. Calculating characteristic path length is easy, albeit a bit time consuming on large graphs — simply find all the length of all shortest paths and take the average. Past analysis of the AS graph has not taken direction into account, and we reproduce that analysis in Figure 34.

From this graph, we can see that that despite the huge growth of the AS graph from under 10,000 ASes to almost 30,000 ASes, the average shortest path length has only increased from 3.25 to 3.5. This provides strong evidence that the AS graph is a small-world graph, which is exactly a graph in which the number of vertices is large, but the average shortest path length is small.

Unfortunately for the relevance of that analysis, traffic in the AS graph does not travel along shortest paths, it travels along valley-free shortest paths. Therefore, we

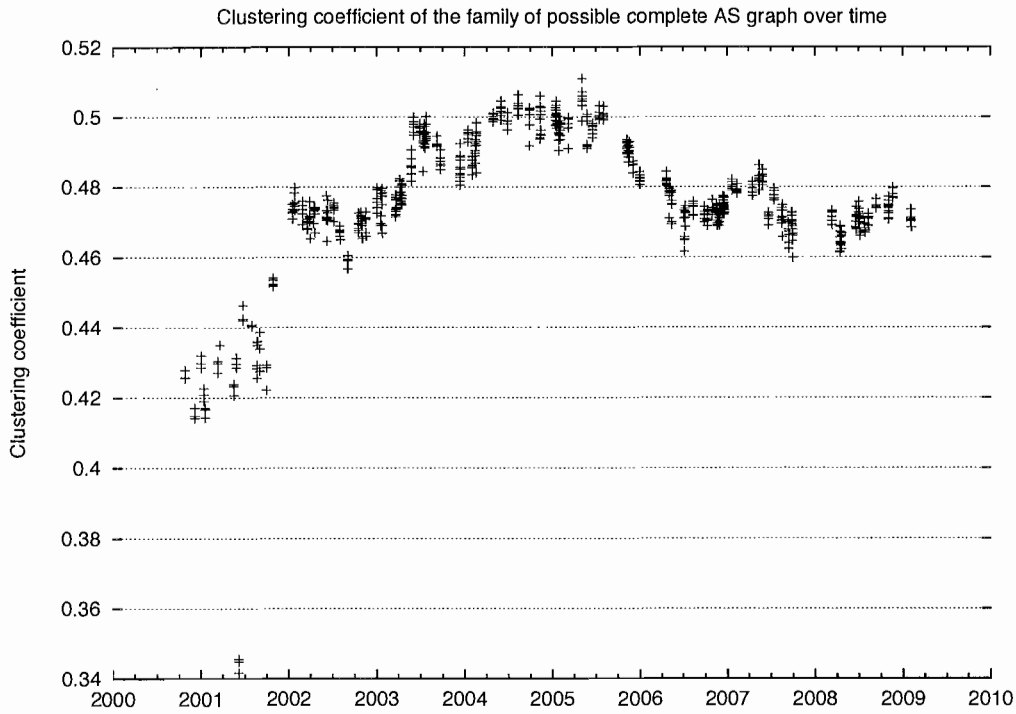


FIGURE 33. The clustering coefficient of the AS graph over time has grown and shrunk from a starting and ending value of around 0.47.

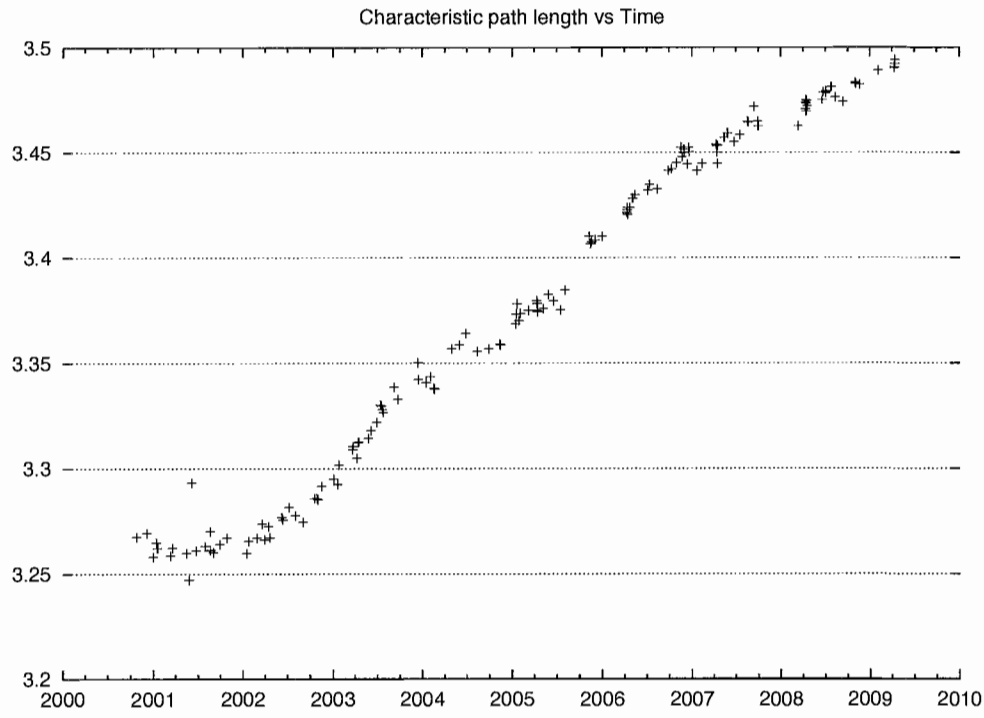


FIGURE 34. The characteristic path length of the AS graph over time

also calculate the characteristic valley-free path length over time, and present those results in Figure 35.

In this figure, we can see that the valley-free path length, after 2004, has grown very slowly. Prior to 2004, we do not have any pre-parsed data from CAIDA available, which means that our analysis of edge-directions prior to 2004 is more suspect, as the CAIDA graphs incorporate data which is not publicly available. Still, from 2004 onwards, we see an increase in average valley-free shortest path length from 3.38 to 3.51. This is only slightly greater than the characteristic path length that neglected edge direction, and implies that policy-compliant routing has not caused paths on the AS graph to be much larger than they would be without needing to conform to policy concerns.

Dhamdhere and Dovrolis note that because the graph has grown tremendously in size, but not in average path length, it must have become dense. This insight led an analysis of the growth rates of inter-AS links, which concluded that the AS graph has been through at least two distinct growth phases, each with distinctive patterns[26]. This result also serves to highlight the folly of trying to fit a simple model to the growth and connection patterns of the AS graph. Instead, we are forced to deal with the data itself, largely absent any generative model.

6.7 Developing Our Own Metrics

In this section, we examine the concerns of the AS graph stakeholders. Once we enumerate some ways in which the AS graph structure might contribute to these concerns, we can then attempt to devise some metric that measures the degree to which the structure is present.

The AS graph is, at a fundamental level, a graph of contracts over which traffic flows. When examining networks of contracts, the relevant concerns are in the

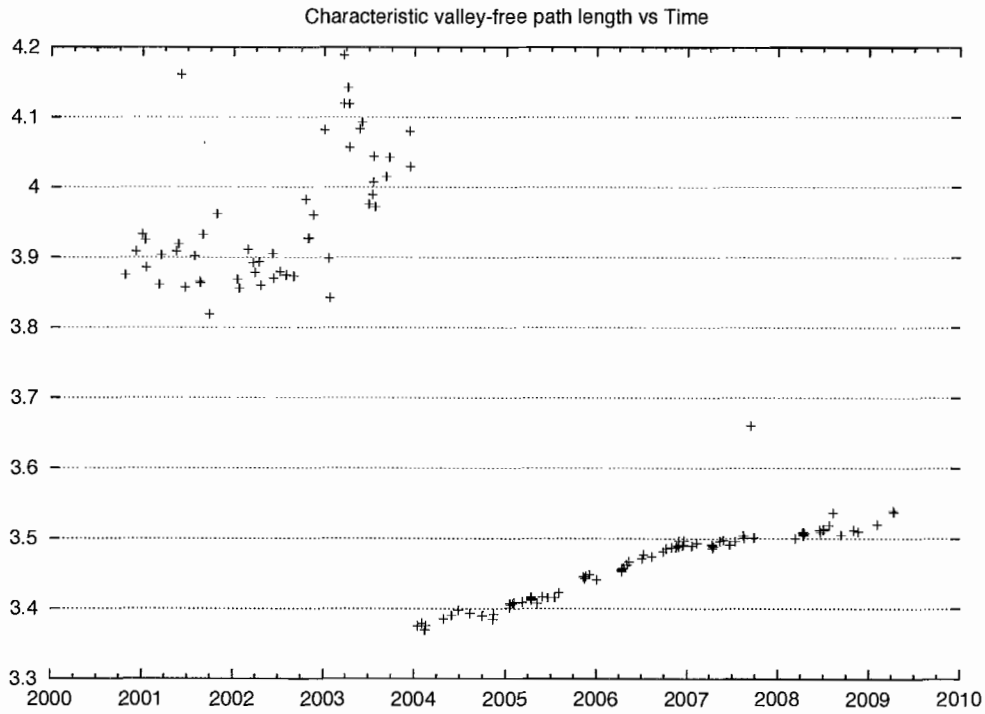


FIGURE 35. The characteristic valley-free path length of the AS graph over time. CAIDA has not provided edge directions prior to 2004, so this also shows how much of an improvement their method is over previously existing methods, which we are forced to use in the absence of a preprocessed graph.

domain of policy, and one of the leading policy questions on the Internet is that of network neutrality. There are several definitions of network neutrality, from the exceedingly general to very specifically technical. Some definitions have to do with blocking traffic and dropping some packets instead of other packets or the idea of differentiated services. Some refer to the motivations of the ISPs, while others make assumptions about core bandwidth versus edge bandwidth. Some researchers think the whole debate foolish and consider it another name for differentiated services.

Differentiated services has a long history in computer networking, and tuning a network for different classes of service and different service types could potentially yield big dividends in efficiency and reliability. But network neutrality is more a fear of blackmail than a fear of technical innovation. Of course, some proposed network neutrality rules also end up potentially prohibiting all forms of differentiated services — including those forms which are useful for network operators and network customers alike, such as dropping too-old VOIP packets. If voice-over-IP packets arrive too late, or are too old, then they will just be discarded by the end host, because playing them would be like pressing rewind on a conversation. So network operators can save their customers the bother of throwing away useless data and save themselves the expense of carrying that packet by inspecting VOIP traffic and discarding too-old packets. Unfortunately, this is in many respects indistinguishable from anti-VOIP behavior, where an ISP could purposely degrade third party VOIP streams in an effort to get people to purchase the ISP's voice-over-IP solution. Both of these situations are examples of differentiated services, and arguably violations of the ideal of network neutrality, but in one case an ISP interferes with customer communications in order to extract more money from their customers (an economic "rent") and in the other case, both the customers and the ISP are better off.

We should note that the technological determinists who believe that network neutrality legislation is inherently unnecessary have at least some leg to stand on.

At the moment, there's a good argument to be made that the cost of offering differentiated services is greater than the cost of simply building enough capacity into a neutral network to make all of the issues irrelevant. If this argument is true, and all networks in the Internet are rational independent actors, then network neutrality may arise naturally and need no legislative backing. Economics, rather than law, could ensure an open Internet.

All of the concerns about network neutrality generally boil down to a fear of monopolistic behavior. The fear that, if a particular network or group of networks was large and properly positioned inside the network, then the Internet would in some crucial way not be the Internet as we know it, but would be a network that was controlled, in all important respects, by that single network or group. To distinguish this group of ASes from other sets of ASes, we call the controlling powerful group of ASes a **cabal**.

Topology can amplify network neutrality concerns. An Internet service provider can exert power only over that traffic which it originates, receives, or passes along. Thus, an ISP can exert control over network traffic both by originating and receiving a lot of traffic (by being large), and by having a lot of traffic pass through it (by being well-positioned). It is possible to construct networks in which being well positioned can be even more important than being large. Consider the bow-tie network in Figure 36. The gray vertex is not large, but all communication between the two large networks must go through it. The gray vertex gains control over network communication through clever placement in the topology, and not through its own size.

Thus, if we wish to examine how much market power an ISP can exert, then we must examine both the ISP's size, as well as try to account for the amount of traffic which flows through it. Implicit in that is the need to account for how traffic flows on the network. Fortunately for us, this is a well-studied area.

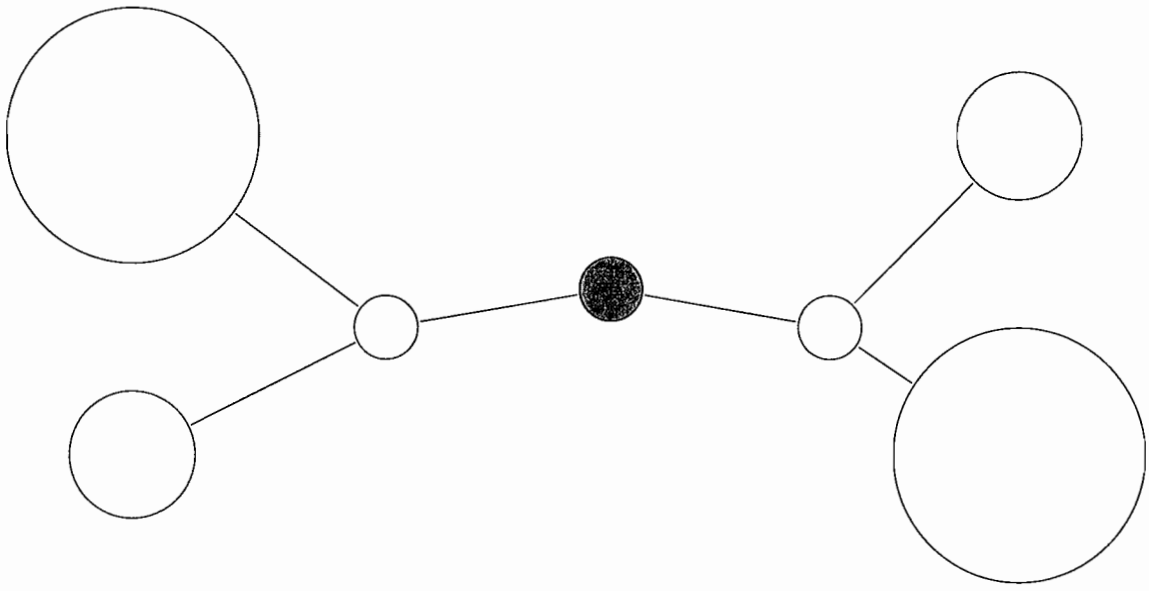


FIGURE 36. Despite its small size, the gray vertex can control a lot of traffic due to its position on the only path between two large ISPs

6.7.1 Modeling the Inter-AS Traffic Matrix

Through measurement, experiment, and backing theoretical model, the inter-AS traffic matrix has been verified to conform to a **gravity model**[21, 29, 76, 72]. In a gravity model, the amount of traffic flow between two entities (u, v) is proportional to the product of their two sizes divided by the square of the distance.

$$\text{flow}(u, v) \in \Theta\left(\frac{|\text{size}(u)| * |\text{size}(v)|}{d(u, v)^2}\right)$$

This equation is exactly analogous to the gravitational force between two bodies in space, and hence the name. With this method, we cannot derive absolute traffic flow quantities, as we would need actual traffic measurements in order to derive the constants involved, but we can derive exact relative flow quantities, which is more than enough in many situations.

6.8 Moving from Policy to Graph Theory

How is market power distributed among ISPs? This question is of interest to policy makers, and the AS graph is the ideal object of study to answer the question. Every vertex of the AS graph is an ISP, and potentially geographically distributed, so failure of a single vertex doesn't make a lot of sense, unless the failure takes place at not a router level, but instead at a company/policy level. Every edge of the AS graph represents a contract between two ASes, which means that analyzing robustness to edge failure is equivalent to analyzing robustness to contract failure. An ISP can only influence the traffic which flows to, from, or through it. Thus, the question of market power distribution is initially a question of how much traffic each individual ISP can influence. If we substitute "AS" for "ISP", then we take our first step towards stating this question as a graph theory problem on the AS graph.

In much previous research[30, 78], a phenomenon of a "rich club" of densely inter-connected, high degree ASes was noted. This rich club, if centrally placed and small, could serve as a bottleneck and be an example of just the kind of group which, if all the members agreed to behave in a certain way, might influence a large amount of Internet traffic.

In order to calculate the power of an AS, we need to know two things: the AS traffic matrix, as well as the chosen path from every AS to every other AS. Unfortunately, neither of these are directly measurable. Instead, we use the number of IP addresses as a rough estimator for the amount of traffic an AS sends, we use the gravity model to figure out the traffic quantities and destinations, and for every valley-free shortest path for which we do not have data, we randomly choose from among the available shortest valley-free paths on the AS graph. Now, armed with all the data required to tackle this problem, we state it formally as: What percentage of Internet traffic does each AS control, and what is the distribution of this power among ASes?

The distribution of power is key here, because it gives insight into two things: Firstly, the amount of traffic controllable by any given AS should not be too high, where "too high" is a number determined by each individual's political beliefs. Secondly, if the distribution contains large gaps, then that suggests that there may be significant barriers to entry in the market. If there are no ASes of middling influence, merely extremely weak ones and extremely powerful ones, then there exists no smooth transition an AS may make on its way from being small to becoming large, or vice versa. From the mean value theorem in calculus, we know that to go from small to large, we must go through medium. Thus, if there are no ISPs of medium size, it implies that it may be difficult for an AS to grow from small size to a large size. In an effort to determine these factors, we define the problem VERTEXPOWER.

Problem 6.8.1 (VERTEXPOWER).

INSTANCE: A graph $G = (V, E)$, a spanning set of shortest valley-free paths P_v for every vertex $v \in V$ from $G = (V, E)$, a flow quantity for each path, and vertex of interest u, and a percentage p.

QUESTION: What percentage of the overall traffic flows to, through, and from the vertex u?

This problem is easily solved by summing all of the traffic flows along all of the paths which contain the vertex in question, and then dividing by the total amount of traffic along all paths in the graph. The algorithm is written out explicitly in Figure 37(A).

The runtime of this algorithm is $O(PL)$, where PL is the total length of all the paths ($PL = \sum_{v \in V} \sum_{p \in P_v} |p|$). If we would like to find the VERTEXPOWER of every vertex in the graph, so as to form a distribution, we call the problem NETWORKPOWER. We note that there is a faster algorithm for NETWORKPOWER than just performing the algorithm for VERTEXPOWER for every vertex, which

would be $O(n * PL)$. Instead, we transform the VERTEXPOWER algorithm slightly and it becomes the algorithm in Figure 37(B). That algorithm for NETWORKPOWER is also $O(PL)$, which is potentially a significant savings of both time and effort.

Both VERTEXPOWER and NETWORKPOWER bear a strong resemblance to the idea of betweenness centrality, which is a measurement that attempts to assess how "central" a vertex is by calculating the percentage of shortest paths that contain the given vertex. Betweenness centrality has many variants, and the methods for calculating each of them are spelled out in a survey paper by Brandes[13], but our situation is somewhat unique. The advantage that our methods have over betweenness centrality is that our methods take into account the unique structure of the AS graph, and the valley-free model. Again we have an example of a case where the AS graph is just special enough to require the development of a new method instead of the simple adaptation of an established one.

However, the market power of single ASes is not the only thing to worry about. It is possible that several autonomous systems might join forces and form an oligopoly. Thus, we not only have to worry about individual ASes behaving badly, we must also worry about multiple ASes behaving badly in concert. This problem becomes even more algorithmically interesting once we notice that it is incorrect to simply add the power of two ASes in order to find the power of those two ASes acting in concert. Consider the bowtie example again, and make a cabal of size two consisting of the gray vertex as well as one of its neighbors. If we were to simply add the power of each of the vertices together, then we would double-count all traffic which goes through both vertices, which in this case includes all the traffic between the two large ASes — yielding the possibility of a significant overcount.

Formally, we define the problem OLIGOPOLYVULNERABILITY as follows:

Problem 6.8.2 (OLIGOPOLYVULNERABILITY).

Vertex-Market-Power (P, v)
// P is a set of tuples (p, t) of paths p and each path's traffic t
// v is the vertex of the graph under test
$power \leftarrow 0$
$total \leftarrow 0$
for each $(p, t) \in P$
    $total \leftarrow total + t$
    for every vertex $u \in p$
        if $v == u$
            $power \leftarrow power + t$
return $\frac{power}{total}$

(A) An algorithm for VertexPower

Network-Market-Power (P, V)
// P is a set of tuples (p, t) of paths p and each path's traffic t
// V is the set of all vertices
$power \leftarrow [0, 0, \ldots 0, 0]$ // The length of the array is $|V|$
$total \leftarrow 0$
for each $(p, t) \in P$
    $total \leftarrow total + t$
    for every vertex $v \in p$
        $power[v] \leftarrow power[v] + t$
for each $v \in V$
    $power[v] \leftarrow \frac{power[v]}{total}$
return *power*

(B) An algorithm for NetworkPower

FIGURE 37. Two algorithms for assessing market power on a graph

INSTANCE: A flow quantity f, a spanning set of shortest valley-free paths P_v for every vertex $v \in V$ from $G = (V, E)$, and the amount of traffic that flows along each path.

QUESTION: What is the minimum set of vertices that can control f of the flow in G?

First, we give some bad news and prove[2] that this problem is $\mathcal{NP}$-Complete via a reduction from HITTINGSET. Even worse news is that HITTINGSET is not approximable to within a constant factor[80], so finding an answer to our question which might even have a bound on its correctness is quite difficult.

Theorem 6.8.3 (OLIGOPOLYVULNERABILITY is $\mathcal{NP}$-Complete)**.** *Given an instance of* OLIGOPOLYVULNERABILITY *consisting of a graph G, a set of paths with flows P, and a total flow quantity f, the question of whether or not there exists a set of vertices of size k or less which can control f flow is $\mathcal{NP}$-Complete.*

Proof. First, we note that given a certificate consisting of the set of vertices, we can verify in polynomial time that the set controls at least a flow of at least f. Therefore, the problem is in $\mathcal{NP}$.

To prove completeness, we reduce from HITTINGSET. An instance of HITTINGSET consists of set of sets S, and a set of elements U that is the union of all the sets in S, and a parameter k. The problem is to decide whether there exists a subset of U of size no more than k such that each at least one element from each set in S is present.

---

[2]Thank you to Daniel Lokshtanov for valuable help with these reductions and a literature search.

We transform an instance of HITTINGSET into an instance of OLIGOPOLYVULNERABILITY in the following manner. We make the set of vertices of the graph G be equal to U. For each set in S we create a path p with flow 1 consisting of the elements of the set in arbitrary order. We augment the edge set of G with all the edges on the path. We then ask if there exists a set of vertices of size no more than k which has a flow of at least $|S|$.

If there is a hitting set of S of size k, then that hitting set will, in our formulation, also contain vertices that are on each of the $|S|$ paths, and so will have a total flow of $|S|$. Therefore, a yes answer to HITTINGSET directly implies a yes answer for OLIGOPOLYVULNERABILITY.

If there is a set of vertices of size k that can control $|S|$ flow, then that set must contain vertices that are on at least $|S|$ different paths. Therefore, the same set of vertices will also form a hitting set of size k. Thus, a yes answer to OLIGOPOLYVULNERABILITY directly implies a yes answer to HITTINGSET.

We have now shown that OLIGOPOLYVULNERABILITY is in $\mathcal{NP}$and the $\mathcal{NP}$-complete problem HITTINGSET may be reduced to it. Therefore, OLIGOPOLYVULNERABILITY is $\mathcal{NP}$-complete. □

Because OLIGOPOLYVULNERABILITY is $\mathcal{NP}$-complete, our next step is to attempt to approximate the optimal answer. Unfortunately, HITTINGSET is not approximable to within better than a factor of $\lg n$ of optimal unless P is equal to $\mathcal{NP}$[31]. In our reduction, the existence of a hitting set of size k directly implies the existence of a set of size k for OLIGOPOLYVULNERABILITY, and vice versa. Therefore, it follows immediately that any ϵ-approximation of OLIGOPOLYVULNERABILITY will also be an ϵ-approximation of HITTINGSET. Therefore, OLIGOPOLYVULNERABILITY is also not approximable to within a factor of better than $\lg n$ (unless $\mathcal{P} = \mathcal{NP}$).

All is not lost however, as the dual of HITTINGSET is SETCOVER and when there is a curve of diminishing returns, like we see in Figure 39, then SETCOVER can be approximated to within a factor of $1 - \frac{1}{e}$ via the greedy algorithm[75]. Let us attempt to leverage this surprising-seeming result and look look at the dual of OLIGOPOLYVULNERABILITY, which we call OLIGOPOLYPOWER and define as follows:

Problem 6.8.4 (OLIGOPOLYPOWER).

INSTANCE: A graph $G = (V, E)$, a number of vertices k, a spanning set of shortest valley-free paths P_v for every vertex $v \in V$ from $G = (V, E)$, and the traffic quantity that flows along each path.

QUESTION: What is the maximum amount of traffic controllable by a set of k vertices from V?

Note that the main difference between OLIGOPOLYVULNERABILITY and OLIGOPOLYPOWER is that in the first one, we attempt to minimize the set size for a given flow quantity, and in the second, we attempt to maximize the flow quantity for a given set size.

OLIGOPOLYPOWER is of course still $\mathcal{NP}$-complete in the general case. It is the dual of an NP-complete problem, which means that a polynomial time algorithm for OLIGOPOLYPOWER would immediately imply a polynomial-time algorithm for OLIGOPOLYVULNERABILITY.

Theorem 6.8.5 (OLIGOPOLYPOWER is $\mathcal{NP}$-Complete). *Given an instance of* OLIGOPOLYPOWER *and a flow quantity f, the question of whether there exists a set of vertices of size k which can control f flow is $\mathcal{NP}$-Complete.*

However, we should point out that there is an obvious $O(|V|^k * PL)$ algorithm for small k — enumerate all sets of size k and check if they satisfy our criteria! Even more nicely, the greedy algorithm will well-approximate OLIGOPOLYPOWER, using

a result of Nemhauser et al. which states that a maximization problem that is maximizing a monotone submodular function is $1 - \frac{1}{e}$ approximable using the greedy algorithm[52]. Submodularity and monotonicity are defined as in Asahiro et al. [75] as:

Definition 6.8.6. *Let S be a finite set, and $f : 2^S \to \mathbb{R}$ be a function with $f(\emptyset) = 0$.*

*f is called **submodular** if for any sets $X, Y \subseteq S$,*

$$f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$$

*f is called **monotone** if for any set $X \subseteq S$ and $s \in S \setminus X$,*

$$f(X \cup \{s\}) - f(X) \geq 0$$

To use this result, we must show that the function we are maximizing (total flow to and through all vertices in the set) meets this criteria.

Lemma 6.8.7. OligopolyPower *is submodular. That is, if we define f to be the flow to and through a given vertex set, then, for all sets of vertices $X, Y \subseteq V$, it is true that $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$.*

Proof. e prove this by noting that $f(X) + f(Y) \geq f(X \cup Y)$ because some flow goes through vertices of both and X and Y. The "double-counted" flow in $f(X) + f(Y)$ that goes through both X and Y is at least equal to $f(X \cap Y)$, although it may be greater than that. To show this, consider the case where $X \neq Y$. In this case, then some flow gets double counted because it comes from a member of $X \setminus Y$ and flows to a member of $Y \setminus X$. This flow will not necessarily be included in $f(X \cap Y)$, but it will be counted in both $f(X)$ and $f(Y)$. All of which is to say that

$f(X) + f(Y) - f(X \cap Y) \geq f(X \cup Y)$, which immediately implies our conclusion that $f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$ □

Lemma 6.8.8. OLIGOPOLYPOWER *is monotone. That is for any set* $X \subseteq V$ *and* $s \in V \setminus X$, $f(X \cup \{s\}) - f(X) \geq 0$

Proof. Adding a vertex to a set will never decrease the total flow to and through a set, as long as all flows are positive (and they are). That is, $f(X \cup \{s\}) \geq f(X)$ for all $X \subseteq V$ and $s \in V$. This immediately implies our desired conclusion. □

These two lemmas, taken together, imply that the flow function in OLIGOPOLYPOWER is both submodular and monotone, and therefore the results of Nemhauser et al. apply, and the greedy algorithm will achieve an approximation ratio of at least $1 - \frac{1}{e}$. Therefore, after performing all of our reductions and analysis, we find that OLIGOPOLYPOWER is the question which we may best answer, and to answer it, the greedy algorithm is the algorithm of choice. This final algorithm in all its simplicity is enumerated in Figure 38.

Unfortunately, calculating the influence for everything would be cost-prohibitive, as enumerating the set of all paths is $O(V^2 * p)$, where p is the length of the longest valley-free shortest path. This run time, while technically in $\mathcal{P}$, is not in-practice fast enough. However, we have gone as far as we can with theory-based speedups. In the next sections, we look at speedups and lower bounds that are based on specific features of our input data.

6.8.1 An Easy Lower Bound on Oligopoly Power

As we established in Section 6.8, determining the maximum influence that can be exerted by a cabal of ASes is $\mathcal{NP}$-complete. However, our proof of this fact hinged on the idea that an AS can exert power not just on the traffic it sends and receives, but also on the traffic it forwards. If we restrict ourselves to only considering the

```
Oligopoly-Power (P, k, G = (V, E))
  // P is a set of tuples (p, t) of paths p and each path's traffic t
  // k is the number of vertices we would like to put in our answer set
  // G is the graph
  total ← 0
  repeat k times
      power ← [0, 0, ... 0, 0] // The length of the array is |V|
      for each (p, t) ∈ P
          for every vertex v ∈ p
              power[v] ← power[v] + t
      v ← the index of the maximum element of power
      total ← total + power[v]
      for each (p, t) ∈ P
          if v ∈ p
              remove (p, t) from P
  return total
```

FIGURE 38. The greedy approximation algorithm for OLIGOPOLYPOWER.

traffic an AS sends and receives, then our analysis becomes a lot easier, and can serve as a lower bound for our connectivity-sensitive analysis. In particular, we note that connectivity will never reduce the amount of traffic originated by an AS — the only thing that connectivity affects is the amount of traffic that goes through an AS.

Therefore, to establish the power of a cabal that refuses to influence traffic across their network, we can simply take the most powerful ASes in order of their expected traffic loads. There are only two things we must be careful of when performing this analysis. This first thing, is that we should be careful that we do not double-count any netblocks; the netblock `10.2.5.0/24` is completely contained inside the netblock `10.0.0.0/8`. If an AS were to announce both blocks, we should discard `10.2.5.0/24` in an effort to not double-count the number of addresses controlled by an AS. The second note is that we should not entirely ignore topology. In particular, sibling-sibling links indicate that the two ASes are actually controlled

by a single entity. In that case, we should treat the two ASes as a single AS for the purposes of cabal size.

Thus, our algorithm for connectivity-insensitive oligopoly power is as follows:

1. If two ASes are linked by a sibling-sibling edge, combine those two ASes into a single AS.

2. Count the total number of IP addresses controlled by each AS, being careful not to double-count.

3. The power of a cabal of size k, is the power of the k ASes that control the largest number of IP addresses.

We run this algorithm for a single day (13 April 2008) and graph the results in Figure 39. From this graph, we see that the situation, even without considering the topology, is potentially quite dire! The power of a small group of ASes grows so dramatically that we must resort to a logarithmic scale to see the dramatic initial growth in power.

On that day, 10% of ASes controlled 90% of Internet traffic. And the problem was even worse as we zoom in on the logarithmic scale. Looking there, we see that 1% of ASes control 40% of Internet traffic! These results are even more surprising when we recall that we have not yet taken topology into account at all — the only topological concern we addressed was the sibling edges, where we considered ASes that were linked via a sibling edge to be part of the same umbrella organization.

Based on that single day, we note that a small number ASes can control a significant amount of Internet traffic. Has it always been this way, or has it been getting worse over time? In order to answer this question, we track the power of a cabal of ASes of size 18 over time to see if the power of a cabal of that size has been growing or shrinking. We chose 18 simply because on the day we tested, a cabal of size 18 could control 30% of Internet traffic. Note that we would expect a fixed-size

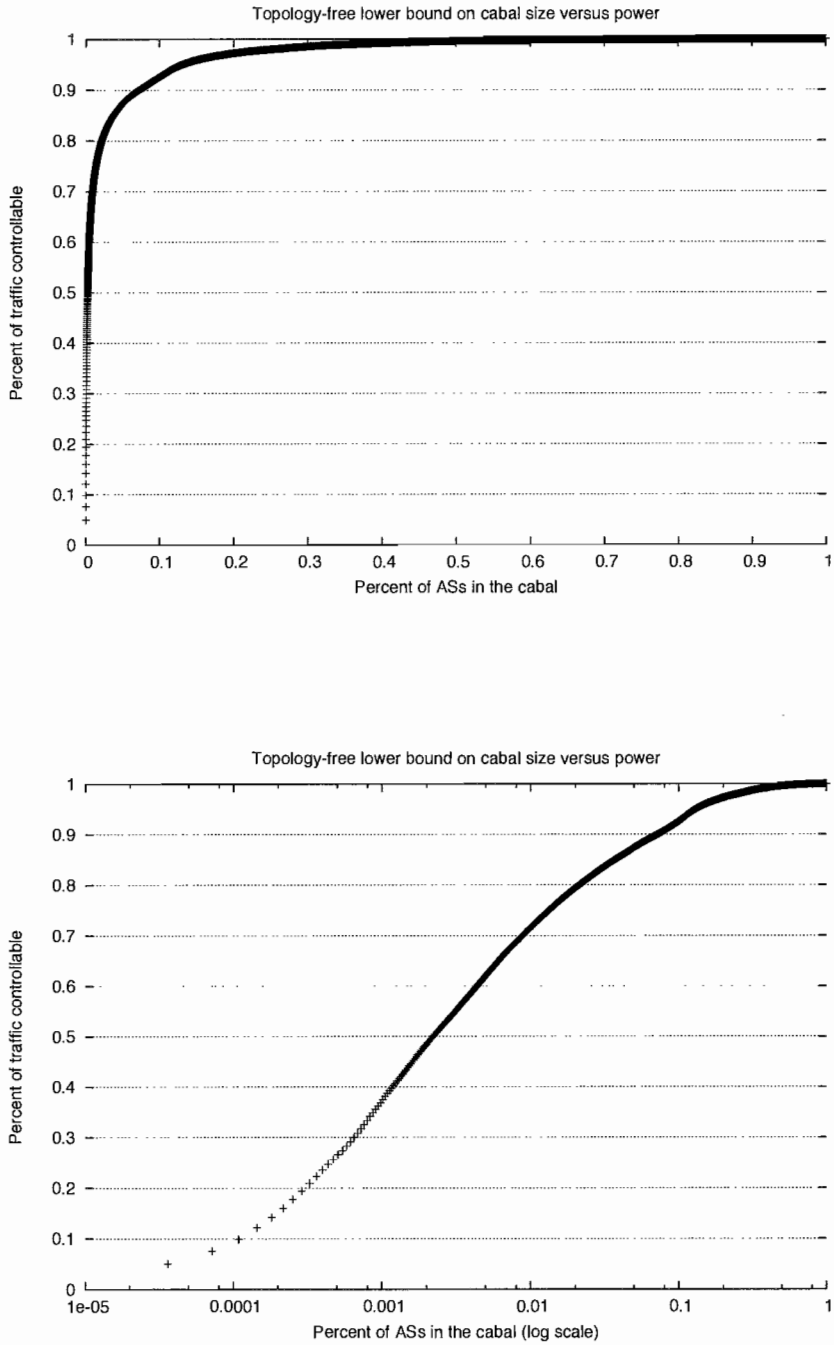


FIGURE 39. Topology-free measure of cabal size versus cabal power for the AS graph on 13 April 2008

cabal to be more powerful in the past, because the network itself was smaller then. In an effort to counter-balance this objection, we note that 18 ASes is .06% of active ASes on the day that we measured. Therefore, as well as tracking the power of a fixed number of ASes, we also track the power of a cabal consisting of .06% of active ASes.

When we look at this number over time, we see that the power of an oligopoly of .06% of ASes has actually been increasing over time. Not by much, but by a small amount. In particular, in the beginning of 2004, a cabal of that size could only control 25% of the traffic on the Internet, while in 2008 it could control 30%, and by 2009, a cabal of just .06% of ASes could control almost a third of Internet traffic. The full graph of these changes from 2004 to now can be seen in Figure 40. Looking at this graph, we find that even when we largely neglect issues of topology, there has, over time, been an increase in the power of a fixed percentage of ASes.

In Figure 41, we see the power of 18 ASes over time.

This topology-insensitive measurement will serve as a lower bound on the centrality of the network, because taking topology into account will strictly increase the amount of traffic an AS can control, and will never decrease it. In the next section we examine how much greater the OligopolyPower becomes when connectivity patterns are taken into account.

6.8.2 An Approximation of Oligopoly Power

Our algorithm for OligopolyPower (Figure 38) runs in time proportional to k times the total length of all the shortest paths in the graph. This is at least $O(k * V^2)$, and, depending on this average shortest path length, could be much higher. Despite being polynomial, this algorithm takes too long in practice to be useful. Can we approximate our approximation? While the general case appears difficult, the origins of traffic flow in the AS graph are, as we saw in the previous

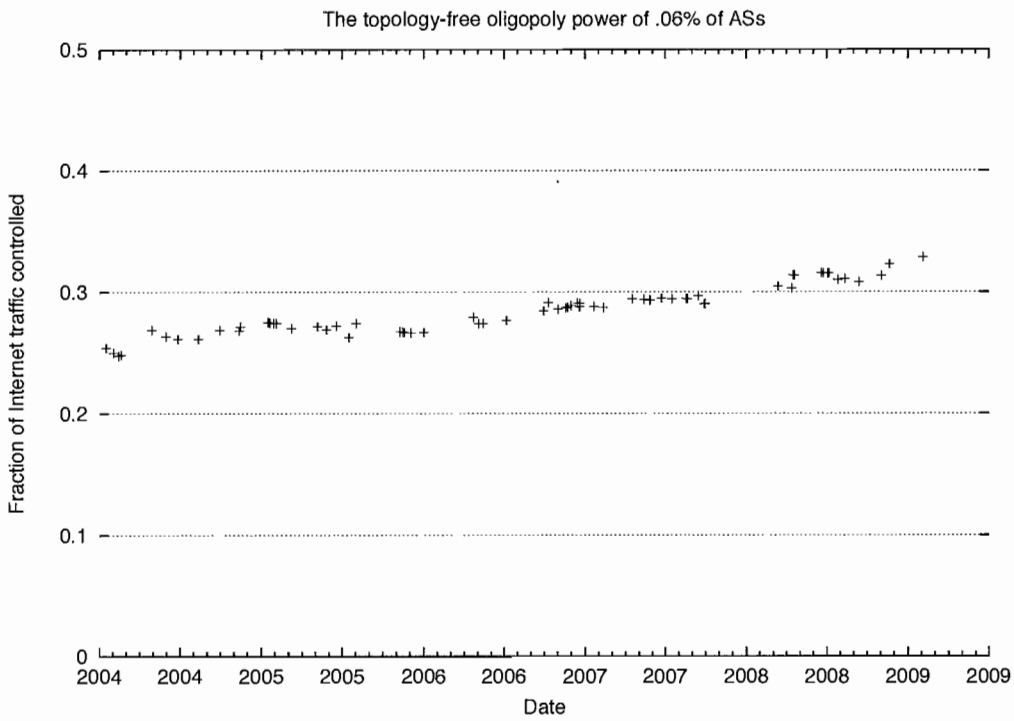


FIGURE 40. The topology-free oligopoly power of .06% of the AS graph over time

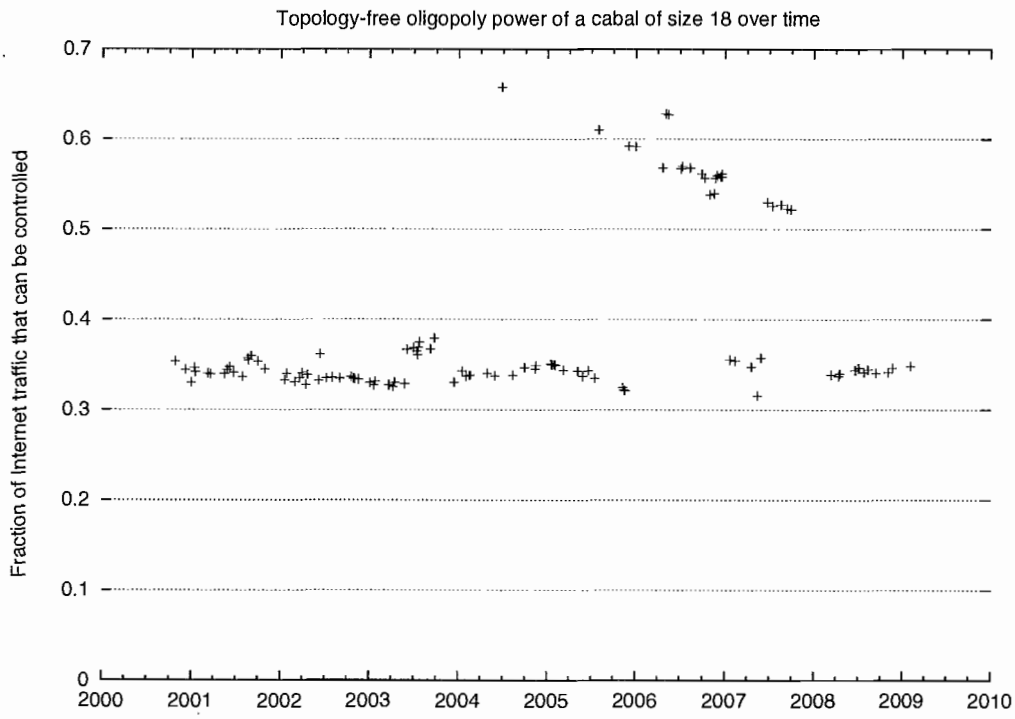


FIGURE 41. The topology-free oligopoly power of 18 ASes over time

section, very clustered around relatively few ASes. Therefore, we can simply take the ASes that send and receive, say, 90% of Internet traffic, and if we only calculate the traffic flows for those ASes, we can be confident that our results, in the worst case, are off by no more than 10% of the total Internet traffic flow.

Note that while we only calculate the flow through other ASes from the large ASes, we will still take into account the flow sent and received by the smaller ASes. We simply neglect to calculate the flow through the AS graph that originates from the smaller ASes.

Several other data-based facts allow for other speedups. For example, there is a very large number of edges which may be present in the graph. Therefore, rather than enumerating the set of possible edges, it turns out to be faster to simply guess an edge at random, and then check whether that edge is a possible edge. For a situation where the set P is very small, this algorithm could take much more time than the deterministic algorithm, but using this method on our data yields a significant speedup in practice.

The in-practice fastest algorithm for approximating OLIGOPOLYPOWER that we use is as follows:

1. Generate the measured AS graph from combination of CAIDA data and Route Views and RIPE as in Chapter III.

2. Augment the measured AS graph by the number of edges indicated by the capture/recapture method in Chapter V.

3. Find the ASes that send and receive the most traffic, as measured by the number of IP addresses they control.

4. Take enough ASes from this set to control the desired percentage of Internet traffic.

5. Calculate the traffic flow from each of these ASes to and through every other AS by finding a valley-free shortest path for each of them and using the gravity model as outlined in Section 6.7.1. Make sure to save the contribution of each source AS to each other AS's total overall flow.

6. Now, select the AS that controls the greatest amount of traffic, and add it to the cabal. Update the flows through every other AS to reflect the fact that the traffic from, to, and through the newly selected AS is now already controlled. Repeat this process until the cabal is of the desired size.

7. Report the power controlled by the cabal, and the likelihood of the graph from which we derived said cabal.

This can be seen, in pseudocode, in Figure 42. If we choose to propagate only, say 90% of Internet traffic, then we can be sure that this algorithm will well-approximate OLIGOPOLYPOWER with an approximation ratio of $(1 - \frac{1}{e}) * .9$.

Using this modified version of the algorithm, we find that the power of a small cabal has been growing over time. This is, of course, not a surprise, as the topology-insensitive lower-bound on OLIGOPOLYPOWER also indicated an increase in the power of a small cabal.

The final graphs, to which this dissertation has been leading, are in Figures 44, 45, and 46. In these graphs, we see the oligopoly power versus oligopoly size for a single day, just like in Figure 39, and we see the growth in power of a cabal of .06% over time, and the growth in power of a cabal of size 18 over time, just like in Figure 40. As we can see, the growth of the power of a cabal is even faster once the topology is taken into account. The power of a group of size 18 has stayed relatively constant, even as the graph has grown, while the power of a group of 0.06% of ASes has grown over time. Thus, we see no evidence that power is dispersing, but not very strong evidence that power is centralizing. Rather, we find that through all of

```
Greedy-Cabal-Power-Approximation (G = (V, E), S, p, k)
  // G = (V, E) is the graph from one of the methods of Section 5.6.2
  // S is the traffic to/from each vertex in V
  // p is the percentage of flow
  // k is the cabal size

  // Perform UNION-FIND for all sibling edges
  for each v ∈ V
      MAKE-SET(v)
  for each (u, v) ∈ E
      if EDGE-TYPE(u, v) = sibling
          UNION(u, v)

  // Calculate the amount of traffic each vertex sends and receives
  sizes ← {}
  for each v ∈ V
      u ←FIND(v)
      sizes[u] ← 0
  for each v ∈ V
      u ←FIND(v)
      sizes[u] ← sizes[u] + S_v

  // Find the set through which the largest amount of traffic flows
  Cabal ← {}
  flow ← 0
  for i = 0 ... k
      v, f ← VALLEY-FREE-BFS-FLOW(G, sizes, Sources, Cabal) // Figure 43
      Cabal ← Cabal ∪ {u ∈ V | FIND(u) = FIND(v)}
      flow ← flow + f
  return flow
```

FIGURE 42. The algorithm for finding a lower bound on Oligopoly Power

```
Valley-Free-BFS-Flow (G, sizes, Sources, Cabal)
  total ← ∑_{v∈V} size[v]
  missingtotal ← total − ∑_{v∈Sources} size[v]
  // F is a list where F_v is the uninfluenced flow through v
  for v ∈ V
      w ←FIND(v)
      if w ∈ Sources ∪ Cabal
          F_w ← 0
      else // Keep track of the flows that we won't be testing with BFS
          F_w ← sizes[w] * missingtotal/total
  for start ∈ Sources \ Cabal
      Q ←CREATE-QUEUE()
      upseen ← {} // The set of vertices we have seen on an upwards-only path
      seen ← {} // The set of vertices we have seen on any path
      ENQUEUE(start, "up", {})
      while NOT-EMPTY(Q)
          v, direction, path ← DEQUEUE(Q)
          if v ∈ seen ∧ v ∈ upseen
              Next-while // Don't revisit already-seen vertices
          if path ∩ Cabal = {} ∧ v ∉ seen // Only augment F for uninfluenced flow
              for u ∈ path
                  w ←FIND(u)
                  F_w ← F_w + size[start] * size[v]/total/d²(start, v)
          // Make sure the search continues to be Valley-Free
          if direction = "up" ∧ v ∉ upseen
              upseen ← upseen ∪ {v}
              for u ∈NEIGHBOR(v, G)
                  if EDGE-TYPE(u, v) ∈ {customer, sibling}
                      ENQUEUE(u, "up", path ∪ {v})
                  else
                      ENQUEUE(u, "down", path ∪ {v})
          elseif direction = "down" ∧ v ∉ seen
              for u ∈NEIGHBOR(v, G)
                  if EDGE-TYPE(u, v) ∈ {provider, sibling}
                      ENQUEUE(u, "down", path ∪ {v})
          seen ← seen ∪ {v}
      return the maximum flow, F_v, and the vertex v with that flow
```

FIGURE 43. An algorithm for finding the vertex that can influence the greatest amount of uninfluenced traffic. The runtime for this algorithm is $O(V^2E)$ in the worst-case, but, because our input data is far from the worst case, is faster than a naïve algorithm on our data.

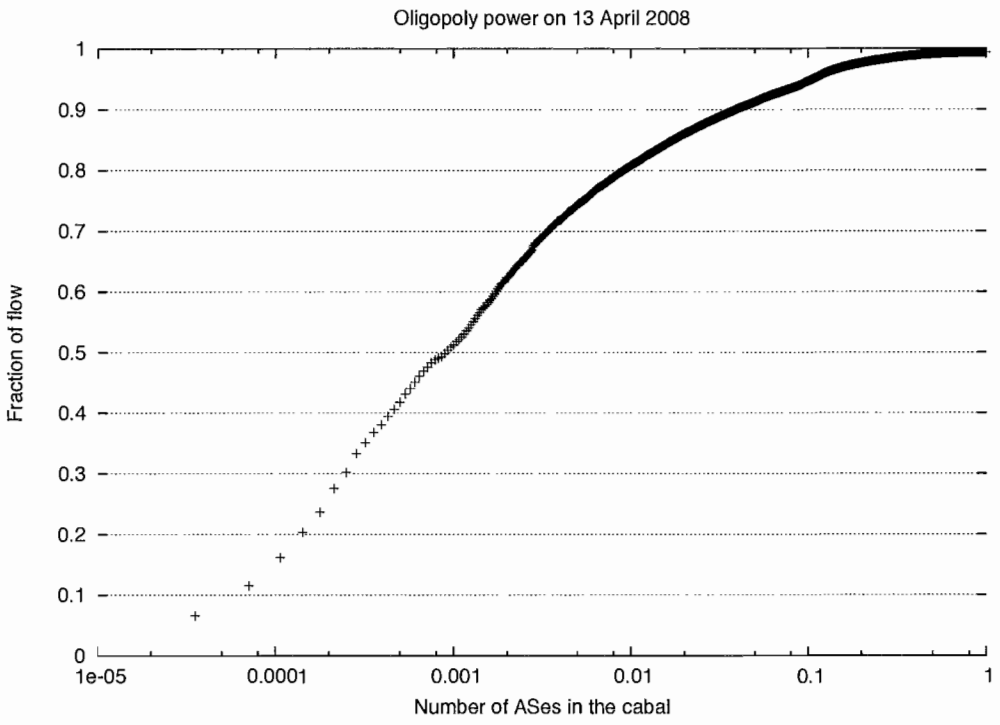


FIGURE 44. An approximation of OLIGOPOLYPOWER on 14 April 2008.

its rapid growth, the Internet is, on an absolute scale, no more vulnerable to oligopoly control than it ever was. On a relative scale, the power of a small oligopoly has grown, but that is because of a changing definition of "small", instead of a changing graph structure. Of course, both of these analyses presume that the value of the Internet has been held constant over time. It is quite possible that a level of oligopoly vulnerability that used to be acceptable is no longer acceptable if the Internet has grown in importance over time.

Despite dramatic growth, OLIGOPOLYPOWER on the AS graph has changed very little. The only sense in which power has changed is that the group of ASes

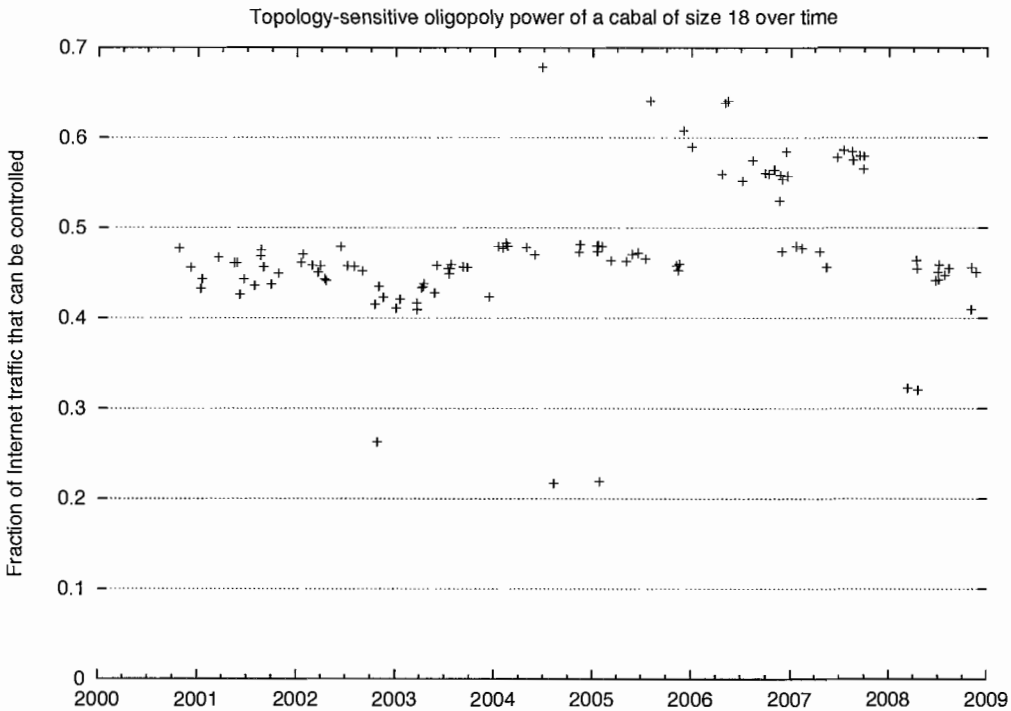


FIGURE 45. An approximation of the OLIGOPOLYPOWER of 18 ASes over time.

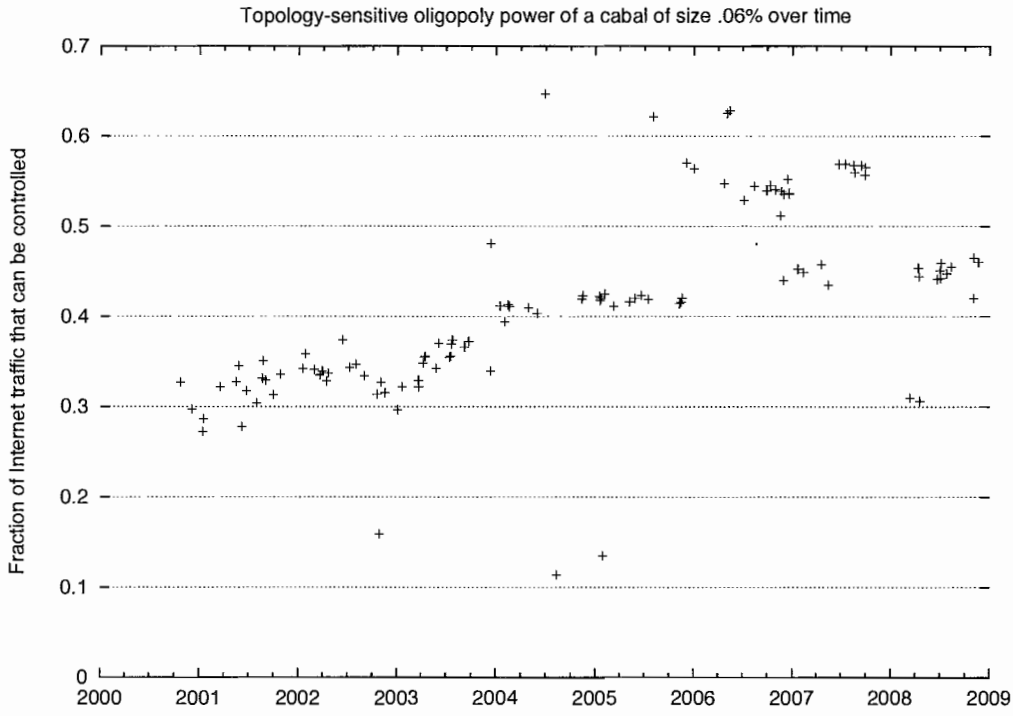


FIGURE 46. An approximation of the OLIGOPOLYPOWER of 0.06% of ASes over time.

that can control Internet traffic has failed to grow along with the AS graph itself. We therefore must conclude that the AS graph is no more vulnerable to oligopoly control than it ever was.

6.9 Chapter Summary

We have, finally, actually measured the properties of the AS graph. We found that many classical graph analysis techniques glossed over important features of the AS graph. We then developed our own metric, designed to discover whether Internet traffic has been centralizing over time, and found that the power of a small cabal of ASes has grown relatively, but not absolutely — it has held constant, even as the AS graph has trebled in the number of vertices and quadrupled in the number of edges. As a preliminary result, this is quite interesting, and shows off the potential of these new analysis methods to shed light on the AS graph.

The new metric we used was actually NP-complete to calculate, but we were able to use various combinations of theoretical approximation algorithms and data-specific speedups to actually calculate the graph properties in a reasonable time. We hope that this metric, a variant of betweenness centrality, can be useful in other contexts involving shortest paths, as very little of our derivation was specific to the AS graph.

CHAPTER VII

SUMMARY

Where we bring it all together and review what has been accomplished

The Internet, which is a very specific internet, has many different layers, each of which could be thought of as "the Internet graph". We concentrated on the layer which best reflected the social aspect of the Internet – the AS graph. In the AS graph individual networks, or autonomous systems (ASes), are represented by a single vertex. Two networks are linked to each other if there exists a contract between them to exchange traffic. Broadly speaking, there are three types of contract that can exist between ASes: peer-to-peer, sibling-to-sibling, and customer-to-provider. That last type of edge is a directed edge from the customer to the provider.

Internet traffic on the AS graph flows in a very particular way. We model this with the valley-free model of Internet routing. The valley-free model, as detailed in Chapter II, restricts both the flow of traffic, and also the available data. When we attempt to model the process of deciding from where in the network our data should be gathered, we find that the problem is NP-complete, although there exist graph classes in which individual models may be efficiently computable.

The incompleteness of our data informed our analysis in Chapter V, where we delved into methods both to add direction to any undirected edges, and to decide how many edges were missing from the AS graph at a given time. We ended up enumerating three methods: lucky, frequentist, and bayesian, but in subsequent analyses in Chapter VI, we found that we were fortunate enough to be able to apply the much easier lucky method.

The analyses of the AS graph in Chapter VI were both a driving force behind the previous chapters on parsing and incompleteness, as well as being a nice example of what can be done with these techniques. We analyzed the AS graph and found that its degree distribution was fat-tailed, but not necessarily power law. We also found that its clustering coefficient was now the same as in 2004, but it had varied significantly between then and now. We also analyzed just the size of the AS graph, and in doing so found that the number of ASes (vertices) has trebled in the past 7 years, while the number of edges has more than quadrupled.

We ended the chapter by developing a new graph metric, related to the idea of betweenness centrality, that tries to express the proportion of total Internet traffic that a single AS or a group of ASes might control. When we did this, we found that 18 ASes could control 45% of Internet traffic, and that this number has remained constant over time, despite the rapid growth in the AS graph.

7.1 Future Work

Although there could be more interesting graph classes for the graph cover problems in Chapter IV, the greatest amount of future work lies in building on the work in Chapter VI. In particular, analyzing the structure of the AS graph to determine exactly how the graph could grow so much without distributing traffic flows more than it has. Other work could analyze the makeup of the group of 18

ASes to attempt to determine whether it was the same 18 ASes over time, or whether new ASes were gaining and losing power.

Another aspect that was not considered in the dissertation was the fact that ASes do not often cross international boundaries, and therefore the concentration of power within a country might be very different than the concentration of power on the AS graph as a whole. Yet another avenue might be to investigate how much of a cabal's power remains even if we allow affected traffic to attempt to route around the cabal if possible.

7.2 In Conclusion

In this dissertation, we have described how to process AS graph data, analyzed the graph theoretic problems that arise when you consider the measurement biases and methods of AS graph data, developed methods to counteract bias in our data, developed a graph theoretic measure of oligopoly market power, and applied our measure to our data in a statistically valid way.

Along the way, we ended up finding out that most interesting questions we might ask of the AS graph have intractable solutions: sometimes the problems were $\mathcal{NP}$-complete, but other times the sheer size of our data made a mockery of the traditional idea that polynomial was the same thing as solvable. The question of network neutrality/oligopoly power, however, can be efficiently approximated when we take into account both the structure of the problem's solution in the general case, as well as the clustering properties of the AS graph data.

Most of the techniques developed in this dissertation are modular by design, which means that they could hopefully be used in other contexts, or to analyze other questions of AS graph data. In this way, we hope to have served not just the interests of those who are curious about network neutrality and the distribution of

traffic flows on the Internet, but also lent a helping hand to other researchers who might be faced with similar problems in a different domain.

APPENDIX

CODE

Code for Parsing New-Style Routing Table Dumps

```
1   static const char RCSID[] = "$Id: bgprocess.c,v 1.2 2008-03-24 08:52:06 peter Exp $";
2   /*
3
4   Copyright (c) 2002                      RIPE NCC
5
6
7   All Rights Reserved
8
9   Permission to use, copy, modify, and distribute this software and its
10  documentation for any purpose and without fee is hereby granted, provided
11  that the above copyright notice appear in all copies and that both that
12  copyright notice and this permission notice appear in supporting
13  documentation, and that the name of the author not be used in advertising or
14  publicity pertaining to distribution of the software without specific,
15  written prior permission.
16
17  THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23
24  */
25
26  /*
27
28  Parts of this code have been engineered after analiyzing GNU Zebra's
29  source code and therefore might contain declarations/code from GNU
30  Zebra, Copyright (C) 1999 Kunihiro Ishiguro. Zebra is a free routing
31  software, distributed under the GNU General Public License. A copy of
32  this license is included with libbgpdump.
33
34  */
35
36  /*
```

```
37  ------------------------------------------------------------------------------
38  Module Header
39  Filename          : bgprocess.c
40  Author            : Dan Ardelean (dan@ripe.net) and then Peter Boothe (peter@cs.uoregon.edu)
41  $Id:$
42  ------------------------------------------------------------------------------
43  */
44
45  #include "bgpdump_lib.h"
46  #include <time.h>
47
48  #include <stdlib.h>
49  #include <netinet/in.h>
50  #include <sys/socket.h>
51  #include <arpa/inet.h>
52
53      void process(BGPDUMP_ENTRY *entry);
54      void show_attr(struct attr *attr);
55      void show_prefixes(int count,struct prefix *prefix);
56  #ifdef BGPDUMP_HAVE_IPV6
57      void show_v6_prefixes(int count, struct prefix *prefix);
58  #endif
59
60  int main(int argc, char **argv) {
61      BGPDUMP *my_dump;
62      BGPDUMP_ENTRY *my_entry=NULL;
63
64      if(argc>1) {
65          my_dump=bgpdump_open_dump(argv[1]);
66      } else {
67          my_dump=bgpdump_open_dump("dumps/updates.20020701.0032");
68      }
69
70      if(my_dump==NULL) {
71          printf("Error opening dump file ...\n");
72          exit(1);
73      }
74
75      do {
76          my_entry=bgpdump_read_next(my_dump);
77          if(my_entry!=NULL) {
78              process(my_entry);
79              bgpdump_free_mem(my_entry);
80          }
81      } while(my_dump->eof==0);
82
83      bgpdump_close_dump(my_dump);
84   return 0;
85  }
86
87  char *bgp_state_name[] = {
88      "Unknown",
89      "IDLE",
90      "CONNECT",
91      "ACTIVE",
92      "OPEN_SENT",
```

```
93      "OPEN_CONFIRM",
94      "ESTABLISHED",
95      NULL
96  };
97
98  char *bgp_message_types[] = {
99      "Unknown",
100     "Open",
101     "Update/Withdraw",
102     "Notification",
103     "Keepalive"
104 };
105
106 char *notify_codes[] = {
107     "Unknown",
108     "Message Header Error",
109     "OPEN Message Error",
110     "UPDATE Message Error",
111     "Hold Timer Expired",
112     "Finite State Machine Error",
113     "Cease"
114 };
115
116 char *notify_subcodes[][12] = {
117     { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
118     /* Message Header Error */
119     {
120         "None",
121          "Connection Not Synchronized",
122         "Bad Message Length",
123         "Bad Message Type",
124         NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
125     },
126     /* OPEN Message Error */
127     {
128         "None",
129         "Unsupported Version Number",
130         "Bad Peer AS",
131         "Bad BGP Identifier",
132         "Unsupported Optional Parameter",
133         "Authentication Failure",
134         "Unacceptable Hold Time",
135         NULL, NULL, NULL, NULL, NULL
136     },
137     /* UPDATE Message Error */
138     {
139         "None",
140         "Malformed Attribute List",
141         "Unrecognized Well-known Attribute",
142         "Missing Well-known Attribute",
143         "Attribute Flags Error",
144         "Attribute Length Error",
145         "Invalid ORIGIN Attribute",
146         "AS Routing Loop",
147         "Invalid NEXT_HOP Attribute",
148         "Optional Attribute Error",
```

```
149           "Invalid Network Field",
150           "Malformed AS_PATH"
151       },
152       /* Hold Timer Expired */
153       { "None", NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
154       /* Finite State Machine Error */
155       { "None", NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
156       /* Cease */
157       { "None", NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL }
158
159   };
160
161   void process(BGPDUMP_ENTRY *entry) {
162       char prefix[BGPDUMP_ADDRSTRLEN], peer_ip[BGPDUMP_ADDRSTRLEN];
163       int i;
164           BGPDUMP_TABLE_DUMP_V2_PREFIX *e;
165
166   if(entry->type == BGPDUMP_TYPE_ZEBRA_BGP
167           && entry->subtype == BGPDUMP_SUBTYPE_ZEBRA_BGP_MESSAGE
168           && entry->body.zebra_message.type == BGP_MSG_KEEPALIVE)
169       return;
170   if(entry->type == BGPDUMP_TYPE_ZEBRA_BGP
171           && entry->subtype == BGPDUMP_SUBTYPE_ZEBRA_BGP_MESSAGE
172           && entry->body.zebra_message.type == BGP_MSG_OPEN)
173       return;
174   if(entry->type == BGPDUMP_TYPE_ZEBRA_BGP
175           && entry->subtype == BGPDUMP_SUBTYPE_ZEBRA_BGP_MESSAGE
176           && entry->body.zebra_message.type == BGP_MSG_NOTIFY)
177       return;
178   if(entry->type == BGPDUMP_TYPE_ZEBRA_BGP
179           && entry->subtype == BGPDUMP_SUBTYPE_ZEBRA_BGP_STATE_CHANGE
180           && entry->length == 8)
181       return;
182
183       switch(entry->type) {
184           case BGPDUMP_TYPE_MRTD_TABLE_DUMP:
185               if(entry->subtype == AFI_IP) {
186                   strcpy(prefix, inet_ntoa(entry->body.mrtd_table_dump.prefix.v4_addr));
187                   strcpy(peer_ip, inet_ntoa(entry->body.mrtd_table_dump.peer_ip.v4_addr));
188   #ifdef BGPDUMP_HAVE_IPV6
189               } else if(entry->subtype == AFI_IP6) {
190                   inet_ntop(AF_INET6, &entry->body.mrtd_table_dump.prefix.v6_addr, prefix,
191                             sizeof(prefix));
192                   inet_ntop(AF_INET6, &entry->body.mrtd_table_dump.peer_ip.v6_addr, peer_ip,
193                             sizeof(peer_ip));
194   #endif
195               } else {
196                   *prefix = '\0';
197                   *peer_ip = '\0';
198               }
199               printf("%s/%d ",prefix,entry->body.mrtd_table_dump.mask);
200               show_attr(entry->attr);
201               break;
202
203           case BGPDUMP_TYPE_TABLE_DUMP_V2:
204
```

```
205                 e = &entry->body.mrtd_table_dump_v2_prefix;
206
207             if(e->afi == AFI_IP) {
208                         strcpy(prefix, inet_ntoa(e->prefix.v4_addr));
209 #ifdef BGPDUMP_HAVE_IPV6
210             } else if(e->afi == AFI_IP6) {
211                         inet_ntop(AF_INET6, &e->prefix.v6_addr, prefix, INET6_ADDRSTRLEN);
212 #endif
213             } else {
214                         printf("Error: BGP table dump version 2 entry with unknown subtype\n");
215                         break;
216             }
217
218                 for(i = 0; i < e->entry_count; i++){
219                     printf("%s/%d ",prefix,e->prefix_length);
220
221                         if(e->entries[i].peer->afi == AFI_IP){
222                                 inet_ntop(AF_INET, &e->entries[i].peer->peer_ip, peer_ip, INET6_ADDRSTRLEN);
223 #ifdef BGPDUMP_HAVE_IPV6
224                         } else if (e->entries[i].peer->afi == AFI_IP6){
225                                 inet_ntop(AF_INET6, &e->entries[i].peer->peer_ip, peer_ip, INET6_ADDRSTRLEN);
226 #endif
227                         } else {
228                                 sprintf(peer_ip, "N/A, unsupported AF");
229                         }
230
231                            show_attr(e->entries[i].attr);
232                 }
233
234             break;
235
236         default:
237             printf("TYPE            : Unknown %d\n", entry->type);
238             show_attr(entry->attr);
239
240     }
241 }
242
243 void show_attr(struct attr *attr) {
244
245     if(attr != NULL) {
246             if( (attr->flag & ATTR_FLAG_BIT(BGP_ATTR_AS_PATH) ) !=0)                printf("%s\n",attr->aspath->str);
247             else printf("\n");
248     }
249 }
250
251 void show_prefixes(int count,struct prefix *prefix) {
252     int i;
253     for(i=0;i<count;i++)
254         printf("      %s/%d\n",inet_ntoa(prefix[i].address.v4_addr),prefix[i].len);
255 }
256
257 #ifdef BGPDUMP_HAVE_IPV6
258 void show_v6_prefixes(int count, struct prefix *prefix) {
259     int i;
260     char str[INET6_ADDRSTRLEN];
```

```
261
262         for(i=0;i<count;i++){
263             inet_ntop(AF_INET6, &prefix[i].address.v6_addr, str, sizeof(str));
264             printf("      %s/%d\n",str, prefix[i].len);
265         }
266     }
267     #endif
```

Code for Creating the Graph

save_day.py

```
1   #!/usr/bin/env python
2
3   import sys, os, commands
4   sys.path.append('/opt/rocks/lib/python2.4/site-packages/')
5
6   verbose = True
7   class Debug:
8       def write(self, s):
9           if verbose:
10              sys.stderr.write(s)
11  debug = Debug()
12
13  def main():
14      global verbose, options
15      from optparse import OptionParser
16      parser = OptionParser()
17      parser.add_option("-d", "--day", dest="day", metavar="day", type="int",
18              help="*REQUIRED* The DAY to analyze")
19      parser.add_option("-m", "--month", dest="month", metavar="MONTH",
20              type="int", help="*REQUIRED* The MONTH to analyze")
21      parser.add_option("-y", "--year", dest="year", metavar="YEAR", type="int",
22              help="*REQUIRED* The YEAR to analyze")
23      parser.add_option("-s", "--sources", dest="sources",
24              default="ripe,routeviews", help="Only use SOURCES for the data",
25              metavar="SOURCES")
26      parser.add_option("-v", action="store_true", dest="verbose", default=True,
27              help="Be verbose (right now this is always on)")
28
29      options, args = parser.parse_args()
30      verbose = options.verbose
31      if options.day == None or options.month == None or options.year == None:
32          print >>sys.stderr, "Randomly choosing a date..."
33
34          if options.year == None:
35              options.year = random.choice([2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008])
36          if options.month == None:
37              options.month = random.choice(range(1, 13))
38
39          if options.day == None:
```

```
40              options.day = random.choice(range(1, 32))
41
42          print >>sys.stderr, "Date is:", options.year, options.month, options.day
43
44      sources_copy = sources.copy()
45      for source in options.sources.split(','):
46          if source not in sources:
47              print >>sys.stderr, "Unknown source:", source
48              sys.exit(1)
49
50      for source in sources:
51          if source not in options.sources.split(','):
52              del sources_copy[source]
53
54      if not sources_copy:
55          print >>sys.stderr, "No sources left!  Something is wrong."
56          sys.exit(1)
57
58      save_day(options.year, options.month, options.day, sources_copy)
59      save_caida(options.year, options.month, options.day)
60
61  def save_caida(year, month, day):
62      import urllib
63      dir = 'http://as-rank.caida.org/data/%d/' % year
64      page = urllib.urlopen(dir).read()
65      import re
66      matches = [ x[1:-1] for x in re.findall('>as-rel.*.txt<', page) ]
67      datestring = '%d%02d%02d' % (year, month, day)
68
69      m = None
70      for m in matches:
71          if m.split('.')[1] >= datestring:
72              break
73      if m != None:
74          urllib.urlretrieve(dir + m, filename='/home/peter/textdata/%d-%d-%d/caidagraph' % (year, month, day))
75      else:
76          open('/home/peter/textdata/%d-%d-%d/caidagraph' % (year, month, day), 'w')
77
78  def save_day(year, month, day, slist=None):
79      if slist == None:
80          slist = sources
81
82      newdir = None
83      for url, localfilename, source in getDataFiles(year, month, day, slist):
84          for block, path in parse_file(localfilename):
85              newdir = save(block, path, year, month, day, url, source, tmpdir)
86
87      finish(tmpdir, newdir)
88      if tmpdir:
89          os.rmdir(tmpdir)
90
91  sources = {
92      'routeviews': """
93  http://archive.routeviews.org/bgpdata/\
94  %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
95  http://archive.routeviews.org/route-views.eqix/bgpdata/\
```

```
96  %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
97  http://archive.routeviews.org/route-views.isc/bgpdata/\
98  %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
99  http://archive.routeviews.org/route-views.kixp/bgpdata/\
100 %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
101 http://archive.routeviews.org/route-views.linx/bgpdata/\
102 %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
103 http://archive.routeviews.org/route-views.wide/bgpdata/\
104 %(year)d.%(month)02d/RIBS/rib.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].bz2
105 http://archive.routeviews.org/oix-route-views/\
106 %(year)d.%(month)02d/oix-full-snapshot-%(year)d-%(month)02d-%(day)02d-[0-9][0-9][0-9][0-9].dat.bz2
107 http://archive.routeviews.org/route-views3/\
108 %(year)d.%(month)02d/route-views3-full-snapshot-%(year)d-%(month)02d-%(day)02d-[0-9][0-9][0-9][0-9].dat.bz2
109 """.split(),
110     'ripe': """
111 http://data.ris.ripe.net/rrc00/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
112 http://data.ris.ripe.net/rrc01/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
113 http://data.ris.ripe.net/rrc02/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
114 http://data.ris.ripe.net/rrc03/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
115 http://data.ris.ripe.net/rrc04/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
116 http://data.ris.ripe.net/rrc05/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
117 http://data.ris.ripe.net/rrc06/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
118 http://data.ris.ripe.net/rrc07/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
119 http://data.ris.ripe.net/rrc08/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
120 http://data.ris.ripe.net/rrc09/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
121 http://data.ris.ripe.net/rrc10/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
122 http://data.ris.ripe.net/rrc11/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
123 http://data.ris.ripe.net/rrc12/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
124 http://data.ris.ripe.net/rrc13/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
125 http://data.ris.ripe.net/rrc14/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
126 http://data.ris.ripe.net/rrc15/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
127 http://data.ris.ripe.net/rrc16/%(year)d.%(month)02d/b?view.%(year)d%(month)02d%(day)02d.[0-9][0-9][0-9][0-9].gz
128 """.split()
129     }
130
131
132 from urllib2 import urlopen, URLError
133 import re
134 import random
135 import tempfile
136 from filesaver import save, finish
137 tmpdir = None
138
139 def downloadAndOpen(repository, directory, name):
140     global tmpdir
141     print >>debug, 'Getting the listing in', directory
142     try:
143         listing = urlopen(directory).read()
144     except URLError, e:
145         print >>debug, "Directory not found:", directory
146         print >>debug, e
147         return None, None
148
149     possibilities = []
150     for possibility in re.finditer(name, listing):
151         possibilities.append(possibility.group())
```

```
152         if not possibilities:
153             print >>debug, "nothing matching", name, "found in", directory
154             return None, None
155 
156         url = directory + '/' + random.choice(possibilities)
157         print >>debug, 'Using the file', url
158         if not tmpdir:
159             tmpdir = tempfile.mkdtemp("thesis", "peter", '/tmp/')
160 
161         if url.endswith('.gz') or url.endswith('.bz2'):
162             tmpfilename = tmpdir + '/' + (url.rsplit('/', 1)[-1])
163             commands.getoutput('cd %s ; wget -T 30 -o wgetlog %s' % (tmpdir, url))
164             print >>debug, "File downloaded to", tmpfilename
165             return tmpfilename, url
166         else:
167             assert False, 'Unknown file type %s - this should never happen' % name
168 
169     def getDataFiles(year, month, day, sources):
170         for repository in sources:
171             for source in sources[repository]:
172                 source %= {'year': year, 'month': month, 'day': day}
173                 directory, fname = source.rsplit('/', 1)
174                 filename, url = downloadAndOpen(repository, directory, fname)
175                 if filename:
176                     print >>debug, 'Successfully opened', url
177                     yield url, filename, repository
178                     os.remove(filename)
179                     print >>debug, 'Removed', filename
180 
181     import os
182     def parse_file(filename):
183         _, input = os.popen2('~/thesis/code/libbgpdump-1.4.99.8/peter %s' % filename)
184         for line in input:
185             try:
186                 block, path = line.strip().split(' ', 1)
187                 yield block, path
188             except ValueError:
189                 print >>debug, "BAD LINE:", line
190                 continue
191         print >>debug, "Read all the output from", filename
192 
193     if __name__ == '__main__':
194         main()
```

predictor.py

```
1   #!/usr/bin/env python
2   
3   import os
4   import save_day
5   from collections import defaultdict
6   
7   class Predictor:
8       def __init__(self, year, month, day):
```

```
9           neighbors = defaultdict(lambda: defaultdict(set))
10
11          dname = '/home/peter/textdata/%d-%d-%d' % (year, month, day)
12          repositories = [ 'routeviews', 'ripe' ]
13
14          for repo in os.listdir(dname):
15              if repo not in repositories: continue
16              for source in os.listdir(dname + '/' + repo):
17                  print repo, source
18                  self.addSource(source,
19                          open('/'.join((dname, repo, source))),
20                          neighbors)
21
22          self.degree = {}
23          ns = len(neighbors)
24          c = 0
25          for v in neighbors:
26              c += 1
27              if c % 1000 == 0:
28                  print c, '/', ns
29
30              prediction = int(self.predict(v, neighbors[v]))
31              if prediction > 0:
32                  self.degree[v] = prediction
33
34      def addSource(self, source, data, neighbors):
35          for line in data:
36              try:
37                  path = line.split()[1:]
38              except:
39                  print "BAD LINE:", line
40                  continue
41
42              for i in range(len(path)-1):
43                  fr = path[i]
44                  to = path[i+1]
45                  neighbors[fr][source].add(to)
46                  neighbors[to][source].add(fr)
47
48
49      def predict(self, v, data):
50          pairs = []
51          keys = data.keys()
52
53          for s in range(len(keys)):
54              ssize = len(data[keys[s]])
55              for t in range(s+1, len(keys)):
56                  both = len(data[keys[s]].intersection(data[keys[t]]))
57                  if both <= 2:
58                      continue
59
60                  tsize = len(data[keys[t]])
61                  pairs.append(ssize * tsize / float(both))
62
63          pairs.sort()
64          if pairs:
```

```
65              return pairs[len(pairs) // 2]
66          else:
67              return 0
68
69
70  def getPredictor(year, month, day):
71      saveMeasuredDay(year, month, day)
72
73      import pickle
74      fname = '/home/peter/textdata/%d-%d-%d/predictor' % (year, month, day)
75      if not os.path.exists(fname):
76          predictor = Predictor(year, month, day)
77          f = file(fname, 'w')
78          pickle.dump(predictor, f)
79      else:
80          predictor = pickle.load(file(fname, 'r'))
81      return predictor
82
83  def saveMeasuredDay(year, month, day):
84      dname = '/home/peter/textdata/%d-%d-%d' % (year, month, day)
85      if not os.path.exists(dname + '/finished'):
86          save_day.save_day(year, month, day)
87
88      if not os.path.exists(dname + '/caidagraph'):
89          save_day.save_caida(year, month, day)
90
91  if __name__ == '__main__':
92      predictor = getPredictor(2008, 4, 13)
93      total = 0
94      for v in predictor.degree:
95          total += predictor.degree[v]
96      print total // 2, len(predictor.degree)
```

generate.py

```
1   #!/usr/bin/env python
2
3   # Both of these are for speed.  We disable gc because we never create cyclical
4   # garbage, and psyco is a JIT compiler for x86
5   import gc; gc.disable()
6   try:
7       import psyco; psyco.full()
8   except:
9       pass
10
11  import os
12  import math
13  import pickle
14  import random
15  from collections import defaultdict
16
17  import save_day
18  from predictor import Predictor, getPredictor
19
```

```
20
21  class ASgraph:
22      def __init__(self):
23          self.netblocks = defaultdict(dict)
24          self.graph = defaultdict(dict)
25
26      def __getitem__(self, item):
27          return self.graph.__getitem__(item)
28
29      def __setitem__(self, item, value):
30          return self.graph.__setitem__(item, value)
31
32
33  class Generator:
34      def __init__(self, year, month, day, predictor):
35          self.distances = defaultdict(dict)
36          self.netblocks = defaultdict(dict)
37          self.graph = defaultdict(dict)
38          self.predictor = predictor.degree
39
40          dir = '/home/peter/textdata/%d-%d-%d' % (year, month, day)
41
42          if os.path.exists(dir + '/caidagraph'):
43              self.addCAIDA(open(dir + '/caidagraph'))
44
45          print "# Reading BGP data"
46          for repo in os.listdir(dir):
47              if repo not in ['routeviews', 'ripe']: continue
48              sources = os.listdir(dir + '/' + repo)
49              for source in sources:
50                  print '#', repo, source
51                  self.addFile(source, open('/'.join((dir, repo, source))))
52
53
54      def addCAIDA(self, data):
55          print "# Reading CAIDA data"
56          relations = { "-1": "c2p", "0": "p2p", "1": "p2c", "2": "s2s" }
57
58          for line in data:
59              line = line.split('#')[0].strip()
60              if not line: continue
61              fr, to, rel = line.split()
62              rel = relations[rel]
63              self.graph[fr][to] = rel
64
65
66      def addFile(self, source, data):
67          for line in data:
68              line = line.split()
69              netblock = line[0]
70              path = line[1:]
71              if not path: continue
72              self.netblocks[path[-1]][netblock] = None
73              self.addPath(source, path)
74
75      def legalUplink(self, fr, to):
```

```
76          # If the link exists, is it an up link?
77          if (fr in self.graph) and (to in self.graph[fr]):
78              return (self.graph[fr][to] in ("c2p", "s2s"))
79
80          # The link does not exist.  Therefore it COULD be an UP link
81          return True
82
83
84      def addPath(self, source, path):
85          # By default, if possible, make the path one continuous UP
86          i = 1
87          while i < len(path) and self.legalUplink(path[i-1], path[i]):
88              if path[i] not in self.graph[path[i-1]]:
89                  self.graph[path[i-1]][path[i]] = "c2p"
90                  self.graph[path[i]][path[i-1]] = "p2c"
91                  self.distances[path[i]][source] = (i, "up")
92              i += 1
93
94          # Now, we can no longer go up.  So we go down.
95          while i < len(path):
96              if path[i] not in self.graph[path[i-1]]:
97                  self.graph[path[i-1]][path[i]] = "p2c"
98                  self.graph[path[i]][path[i-1]] = "c2p"
99                  self.distances[path[i]][source] = (i, "down")
100             i += 1
101
102     def legalEdgeTypes(self, fr, to):
103         possible = set("c2p p2c p2p".split())
104         for source in self.distances[fr]:
105             if source not in self.distances[to]: continue
106             if not possible: return []
107
108             distf, dirf = self.distances[fr][source]
109             distt, dirt = self.distances[to][source]
110
111             if distf + 1 < distt:
112                 if dirf == "up":
113                     if dirt == "down":
114                         return []
115                     else:
116                         possible.discard("c2p")
117                         possible.discard("p2p")
118                 else: # dirf == "down"
119                     if dirf == "down":
120                         possible.discard("p2c")
121                         possible.discard("p2p")
122                     else:
123                         pass
124         return [ p for p in possible ]
125
126     def generate(self):
127         # First, we count up how many half-edges we are missing
128         halfedges = []
129         for v in self.predictor:
130             pred = int(self.predictor[v])
131             deg = len([ u for u in self.graph[v] if u in self.predictor ])
```

```
132
133             if deg < pred:
134                 for i in range(deg, pred):
135                     halfedges.append(v)
136
137         # Now we make a new AS graph
138         asgraph = ASgraph()
139         asgraph.netblocks = self.netblocks
140
141         # Now we copy over the entirety of X, the graph that is known to exist
142         for u in self.graph:
143             for v in self.graph[u]:
144                 asgraph[u][v] = self.graph[u][v]
145
146         # Now we use the half-edges to try to generate edges from P
147         opposite = { "p2c":"c2p", "c2p":"p2c", "p2p":"p2p", "s2s":"s2s" }
148
149         count = 10*len(halfedges)
150         print "# Now adding", len(halfedges), "halfedges (hopefully)"
151         while count > 0 and len(halfedges) > 0:
152             count -= 1
153             frindex = random.randint(0, len(halfedges)-1)
154             toindex = random.randint(0, len(halfedges)-1)
155             fr = halfedges[frindex]
156             to = halfedges[toindex]
157
158             if fr == to: continue
159             if to in self.graph[fr]: continue
160
161             types = self.legalEdgeTypes(fr, to)
162             if not types: continue
163
164             edgeType = random.choice(types)
165             asgraph[fr][to] = edgeType
166             asgraph[to][fr] = opposite[edgeType]
167
168             if frindex > -1 and toindex > -1:
169                 halfedges[-1], halfedges[frindex] = halfedges[frindex], halfedges[-1]
170                 halfedges[-1], halfedges[toindex] = halfedges[toindex], halfedges[-1]
171                 halfedges.pop()
172                 halfedges.pop()
173             elif frindex > -1:
174                 halfedges[-1], halfedges[frindex] = halfedges[frindex], halfedges[-1]
175                 halfedges.pop()
176             elif toindex > -1:
177                 halfedges[-1], halfedges[toindex] = halfedges[toindex], halfedges[-1]
178                 halfedges.pop()
179
180         print '#', len(halfedges)//2, "remain un-added to the graph"
181
182         return asgraph
183
184 def getGenerator(year, month, day):
185     pred = getPredictor(year, month, day)
186
187     fname = '/home/peter/textdata/%d-%d-%d/generator' % (year, month, day)
```

```
188         if not os.path.exists(fname):
189             print "# Making new generator"
190             generator = Generator(year, month, day, pred)
191             f = file(fname, 'w')
192             pickle.dump(generator, f)
193         else:
194             generator = pickle.load(file(fname, 'r'))
195         return generator
196
197     if __name__ == '__main__':
198         y,m,d = 2008, 4, 13
199
200         import sys
201         if len(sys.argv) == 4:
202             y,m,d = map(int, sys.argv[1:])
203
204         generator = getGenerator(y, m, d)
205
206         print len(generator.graph), 'vertices', \
207                 sum([len(generator.graph[v]) for v in generator.graph])/2, \
208                 'edges measured'
209
210         asgraph = generator.generate()
211
212         print len(asgraph.graph), 'vertices', \
213                 sum([len(asgraph.graph[v]) for v in asgraph.graph])/2, \
214                 'edges predicted'
```

Code for Analyzing the Oligopoly Vulnerability of the AS graph

centralization.py

```
1   #!/usr/bin/env python
2   import psyco; psyco.full()
3
4   import os, sys
5   import random
6   from collections import defaultdict
7   import generate
8   from generate import Generator
9   from collections import deque
10  from nb import Netblock
11
12  class UnionFind:
13          def __init__(self, data):
14                  self.parent = self
15                  self.data = data
16          def find(self):
```

```
17                  if self.parent != self:
18                          self.parent = self.parent.find()
19                          return self.parent
20                  else:
21                          return self
22          def union(self, other):
23                  self.find().parent = other.find()
24          def __repr__(self):
25                  return "UF(%s -> %s)" % (str(self.data), str(self.parent.data))
26
27  def find_cliques(graph):
28          uf = {}
29          for v in graph:
30                  uf[v] = UnionFind(v)
31          for v in graph:
32                  for u in graph[v]:
33                          if graph[u][v] == 's2s':
34                                  uf[u].union(uf[v])
35          cliques = {}
36          for v in graph:
37                  cliques[v] = uf[v].find().data
38          return cliques
39
40  def test_find_cliques():
41          tg = {}
42          for i in range(10):
43                  tg[i] = {}
44          for i in range(10):
45                  tg[i][(i+1) % 10] = 'p2p'
46                  tg[(i+1) % 10][i] = 'p2p'
47          tg[4][6] = 's2s'
48          tg[6][4] = 's2s'
49          cliques = find_cliques(tg)
50          assert cliques[4] == cliques[6]
51          for i in range(10):
52                  assert cliques[i] != cliques[(i+1) % 10]
53          print '# find_cliques() tests okay!'
54
55  def compact_blocks(blocks):
56          blocks.sort()
57          blocks.reverse()
58          newblocks = []
59          for test in blocks:
60                  for b in blocks:
61                          if test != b and test in b:
62                                  break
63                  else:
64                          newblocks.append(test)
65          return newblocks
66
67  def test_compact_blocks():
68          s = "0.0.0.0/0 10.2.3.4/16"
69          blocks = [Netblock(b) for b in s.split()]
70          assert 1 == len(compact_blocks(blocks)), str(blocks)
71          s = "10.0.0.0/8 10.2.3.4/16"
72          blocks = [Netblock(b) for b in s.split()]
```

```
73          assert 1 == len(compact_blocks(blocks)), str(blocks)
74          s = "3.0.0.0/8 10.2.3.4/16"
75          blocks = [Netblock(b) for b in s.split()]
76          assert 2 == len(compact_blocks(blocks)), str(blocks)
77          print "# compact_blocks tests okay!"
78
79
80  def find_sizes(asgraph):
81          sizes = defaultdict(lambda: 0)
82          for v in asgraph.graph:
83                  blocks = [ Netblock(b) for b in asgraph.netblocks[v] if '.' in b and b != '0.0.0.0/0' ]
84                  blocks = [ b for b in blocks if b.bits >= 8 ]
85                  blocks = compact_blocks(blocks)
86                  sizes[v] = sum([b.size() for b in blocks])
87          return sizes
88
89
90  def calc_flow(start, graph, sizes):
91          extraflow = defaultdict(lambda: 0.0)
92          upseen = set()
93          dnseen = set()
94          tbd = []
95          next = []
96
97          tbd.append((start, "up", []))
98          while tbd or next:
99                  if not tbd:
100                         tbd = next
101                         random.shuffle(tbd)
102                         next = []
103
104                 v, dir, path = tbd.pop()
105
106                 if v not in upseen and v not in dnseen and path:
107                         flow = sizes[start] * sizes[v] / float(len(path)**2)
108                         for u in path[:-1]:
109                                 extraflow[u] += flow
110
111                 if dir == "up" and v not in upseen:
112                         upseen.add(v)
113                 elif dir == "dn" and v not in dnseen:
114                         dnseen.add(v)
115                 else:
116                         continue
117
118                 for u in graph[v]:
119                         if dir == "up":
120                                 if graph[v][u] in [ "c2p", "s2s" ]:
121                                         next.append((u, "up", path + [ u ] ))
122                                 else:
123                                         next.append((u, "dn", path + [ u ] ))
124                         elif dir == "dn":
125                                 if graph[v][u] in [ "p2c", "s2s" ]:
126                                         next.append((u, "dn", path + [ u ] ))
127
128         # Now we need to normalize the augmented flow
```

```
129         total = sum(extraflow.values())
130         factor = sizes[start] / float(total)
131         for v in extraflow:
132                 extraflow[v] *= factor
133         return extraflow
134
135
136 if __name__ == '__main__':
137         try:
138                 y,m,d = map(int, sys.argv[1].split('-'))
139                 percent = float(sys.argv[2])
140         except:
141                 print "# USING THE DEFAULT DAY AND PERCENT"
142                 y,m,d = 2008, 4, 13
143                 percent = .75
144                 test_find_cliques()
145                 test_compact_blocks()
146
147         asgraph = generate.getGenerator(y, m, d).generate()
148
149         print "# Finding cliques"
150         cliques = find_cliques(asgraph.graph)
151         clique_members = defaultdict(set)
152         for f,t in cliques.items():
153                 clique_members[t].add(f)
154
155         print "# Finding sizes"
156         sizes = find_sizes(asgraph)
157
158         clique_sizes = defaultdict(lambda: 0)
159         for rep in clique_members:
160                 for member in clique_members[rep]:
161                         clique_sizes[rep] += sizes[member]
162
163         print "# Processing sizes"
164         csizes = [ (s, r) for (r,s) in clique_sizes.items() ]
165         csizes.sort()
166         csizes.reverse()
167         totalsize = sum(s for (s, _) in csizes)
168
169         print "# Finding the numer required to get to 75%"
170         total = 0.0
171         for i in range(len(csizes)):
172                 total += csizes[i][0]
173                 if total >= percent * totalsize:
174                         break
175         print "# %d ASes control %f%% of the traffic" % (i, 100*percent)
176
177         print "# Augmenting the sizes with flow"
178         extraflow = {}
179         for (_, rep) in csizes[:i]:
180                 rep = cliques[rep]
181                 for AS in clique_members[rep]:
182                         if sizes[AS] == 0: continue
183                         extraflow[AS] = calc_flow(AS, asgraph.graph, sizes)
184
```

```
185         print "# Calculating the total flow"
186         totalflow = {}
187         for v in sizes:
188                 totalflow[v] = sizes[v]
189                 for u in extraflow:
190                         totalflow[v] += extraflow[u][v]
191
192         print "# Finding the ASes with the greatest flow, in order"
193         AS_ordering = [ (f, AS) for (AS, f) in totalflow.items() ]
194         AS_ordering.sort()
195         AS_ordering.reverse()
196
197         available = set(AS for AS in totalflow)
198
199         print "# n n% flow flow% totalflow totalflow%"
200         count = 0
201         cliquecount = 0
202         total = 0.0
203         for flow, AS in AS_ordering:
204                 oldtotal = total
205                 if AS not in available: continue
206                 rep = cliques[AS]
207                 cliquecount += 1
208                 for suborned in clique_members[rep]:
209                         count += 1
210                         total += totalflow[suborned]
211                         available.discard(suborned)
212                         # Now we need to fix overlapping flow problems
213                         if suborned in extraflow:
214                                 for v in extraflow[suborned]:
215                                         totalflow[v] -= extraflow[suborned][v]
216                                 if v not in available:
217                                         total -= extraflow[suborned][v]
218                         for u in extraflow:
219                                 if suborned in extraflow[u] and u in suborned:
220                                         total -= extraflow[u][suborned]
221
222                 print cliquecount, float(cliquecount)/len(clique_members), count, \
223 count/float(len(AS_ordering)), total-oldtotal, \
224 (total-oldtotal)/totalsize, total, total/totalsize, '#', AS
```

nb.py

```
1  class Netblock:
2      def __init__(self, block):
3          ip, bits = block.split('/')
4          self.bits = int(bits)
5          self.block = block
6          ip = map(int, ip.split('.'))
7          self.ip = (ip[0] << 24) + (ip[1] << 16) + (ip[2] << 8) + ip[3]
8          self.mask = (0xFFFFFFFF << (32 - self.bits)) & 0xFFFFFFFF
9          assert self.bits <= 32, "Bad block:" + block
10         self.length = int(1 << (32-self.bits))
11
```

```
12      def __contains__(self, nb):
13          return self.bits <= nb.bits and ((self.ip & self.mask) == (nb.ip & self.mask))
14
15      def __str__(self):
16          return "Netblock(" + self.block + ")"
17      def __repr__(self):
18          return "Netblock(" + self.block + ")"
19
20      def size(self):
21          return self.length
22
23      def __cmp__(self, other):
24          assert type(other) == type(self), 'What is ' + str(other) + ' doing in the array?'
25          sizecmp = cmp(self.length, other.length)
26          if sizecmp != 0:
27              return sizecmp
28          else:
29              return -1 * cmp(self.ip, other.ip)
30
31
32  if __name__ == '__main__':
33      # run some tests
34      assert Netblock('10.0.1.0/24') in Netblock('10.0.0.0/8')
35      assert Netblock('10.0.1.0/8') not in Netblock('10.0.0.0/16')
36      assert Netblock('10.0.0.0/8') not in Netblock('10.0.0.0/16')
37      assert Netblock('10.0.1.0/24') not in Netblock('11.0.0.0/8')
38      assert Netblock('10.0.0.0/8') not in Netblock('11.0.0.0/8')
39      assert Netblock('10.0.1.0/24').size() == 256
40      assert Netblock('10.0.1.0/8').size() == 256*256*256
41
42      tf = Netblock('10.0.1.0/24')
43      to = Netblock('11.0.0.0/8')
44      l = [to, tf, to, tf]
45      l.sort()
46      l.reverse()
47      assert tuple(l) == ( to, to, tf, tf )
```

BIBLIOGRAPHY

[1] D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the Bias of Traceroute Sampling; or, Power-law Degree Distributions in Regular Graphs. In *Symposium on Theory of Computing*, pages 694 – 703, May 2005.

[2] A. Akella, S. Chawla, A. Kannan, and S. Seshan. Scaling properties of the Internet graph. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 337–346, 2003.

[3] D. Alderson, L. Li, W. Willinger, and J. C. Doyle. Understanding Internet topology: principles, models, and validation. *IEEE/ACM Transactions on Networking*, 13(6):1205–1218, 2005.

[4] D. L. Alderson. Technological and Economic Drivers and Constraints in the Internet's "Last Mile". Technical Report CIT-CDS 04-004, California Institute of Technology, 2004.

[5] D. G. Andersen, N. Feamster, S. Bauer, and H. Balakrishnan. Topology inference from BGP routing dynamics. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 243–248, 2002.

[6] G. D. Battista, T. Erlebach, A. Hall, M. Patrignani, M. Pizzonia, and T. Schank. Computing the types of the relationships between autonomous systems. *IEEE/ACM Transactions on Networking*, 15(2):267–280, 2007.

[7] Z. Beerliova, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, M. Mihalak, and L. S. Ram. Network Discovery and Verification. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2005)*, volume LNCS 3787, pages 127–138. Springer, 2005.

[8] S. T. Berners-Lee. `http://dig.csail.mit.edu/breadcrumbs/node/144`. It seems appropriate to use a URL to cite the inventor of the world wide web.

[9] A. Bonato. A survey of models of the web graph. In *Combinatorial and Algorithmic Aspects of Networking*, pages 159–172. Springer-Verlag LNCS, 2004.

[10] A. Bonato and J. Janssen. Infinite Limits of Copying Models of the Web Graph. *Internet Mathematics*, 1(2):193–213, 2004.

[11] P. Boothe, Z. Dvořák, A. Farley, and A. Proskurowski. Graph covering via shortest paths. *Congressus Numerantium*, 2007.

[12] S. Bradner and A. Mankin. The Recommendation for the IP Next Generation Protocol. Internet Engineering Task Force: RFC 1752, January 1995.

[13] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, 2008.

[14] U. Brandes and T. Erlebach, editors. *Network Analysis*, volume 3418 of *LNCS*. Springer-Verlag, 2005.

[15] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph classes: a survey*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[16] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[17] A. Broido and kc claffy. Internet Topology: connectivity of IP graphs. In *Proceedings from the SPIE International Symposium on Convergence of IT and Communication*, pages 172–187, August 2001.

[18] A. Broido, E. Nemeth, and kc claffy. Internet Expansion, Refinement and Churn. *European Transactions on Telecommunications*, 13(1):33–51, January 2002.

[19] T. Bu and D. Towsley. On distinguishing between Internet power law topology generators. In *Proceedings of IEEE INFOCOM*, pages 638–647, June 2002.

[20] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38(1):2, 2006.

[21] H. Chang, S. Jamin, Z. M. Mao, and W. Willinger. An empirical approach to modeling inter-AS traffic matrices. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[22] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link Evolution: Analysis and Algorithms. *Internet Mathematics*, 1(3):277–304, 2004.

[23] D. Clark. Design Philosophy of the DARPA Internet Protocols. In *Proceedings of ACM SIGCOMM*, pages 106–114, August 1988.

[24] A. Curtis, D. Massey, and R. M. McConnell. Efficient algorithms for optimizing policy-constrained routing. In *International Workshop on Quality of Service (IWQoS 2007)*, pages 113–116, 2007.

[25] B. de Fluiter. *Algorithms for Graphs of Small Treewidth.* PhD thesis, Universiteit Utrecht, March 1997. ISBN 90-393-1528-0.

[26] A. Dhamdhere and C. Dovrolis. Ten years in the evolution of the internet ecosystem. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 183–196, New York, NY, USA, 2008. ACM.

[27] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, kc claffy, and G. Riley. AS Relationships: Inference and Validation. *ACM SIGCOMM Computer Communications Review*, 37(1):29–40, 2007.

[28] D.J. Rose. On simple characterizations of k-trees. *Discrete Math*, 7:317–322, 1974.

[29] J. C. Doyle, D. Alderson, L. Li, S. Low, M. Roughan, S. Shalunov, R. Tanaka, and W. Willinger. The "Robust Yet Fragile" Nature of the Internet. *Proceedings of the National Academy of Sciences*, 102(41):14497–14502, 2005.

[30] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *SIGCOMM*, pages 251–262, 1999.

[31] U. Feige. A Threshold of Ln N for Approximating Set Cover. *Journal of the ACM*, 45:314–318, 1998.

[32] A. D. Flaxman and J. Vera. Bias Reduction in Traceroute Sampling - Towards a More Accurate Map of the Internet. In *Proceedings of the 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, pages 1–15, 2007.

[33] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-Transitive Connectivity and DHTs. In *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS '05)*, pages 55–60, 2005.

[34] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking*, 9(6):733–745, 2001.

[35] L. Gao and F. Wang. The Extent of AS Path Inflation by Routing Policies. In *IEEE Global Internet Symposium*, volume 3, pages 2180–2184, 2002.

[36] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[37] M. X. Goemans and D. P. Williamson. .879-approximation algorithms for MAX CUT and MAX 2SAT. In *STOC '94: Proceedings of the twenty-sixth annual ACM Symposium on Theory of Computing*, pages 422–431, New York, NY, USA, 1994. ACM.

[38] H. Haddadi, D. Fay, A. Jamakovic, O. Maennel, A. W. Moore, R. Mortier, M. Rio, and S. Uhlig. Beyond Node Degree: Evaluating AS Topology Models. Technical Report UCAM-CL-TR-725, University of Cambridge, Computer Laboratory, July 2008.

[39] G. Huston. Exploring Autonomous System Numbers. *The Internet Protocol Journal*, 9(1), March 2006.

[40] Y. Hyun, A. Broido, and kc claffy. Traceroute and BGP AS Path Incongruities. Technical report, University of California, San Diego, 2003.

[41] Y. Hyun, B. Huffaker, D. Andersen, E. Aben, M. Luckie, kc claffy, and C. Shannon. The IPv4 Routed /24 AS Links Dataset. `http://www.caida.org/data/active/ipv4_routed_topology_aslinks_dataset.xml`.

[42] J. Matoušek and R. Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12(1):1–22, March 1991.

[43] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The Web as a graph: Measurements, models and methods. In *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*, 1999.

[44] T. Kloks. *Treewidth.* PhD thesis, Rijksuniversiteit te Utrecht, June 1993. ISBN 90-393-0406-8.

[45] A. Lakhina, J. W. Byers, M. Crovella, and P. Xie. Sampling Biases in IP Topology Measurements. In *Proceedings of IEEE INFOCOM*, volume 1, pages 332–341, 2003.

[46] S. H. Lee, P.-J. Kim, and H. Jeong. Statistical properties of sampled networks. *Physical Review E*, 73(1), 2006.

[47] L. Li, D. Alderson, W. Willinger, and J. Doyle. A First-Principles Approach to Understanding the Internet's Router-level Topology. In *Proceedings of ACM SIGCOMM*, pages 3–14, 2004.

[48] D. Liben-Nowell. *An Algorithmic Approach to Social Networks.* PhD thesis, MIT, June 2005.

[49] L. Lovász. Very large graphs. *Current Developments in Mathematics*, (to appear), December 2008.

[50] E. P. Markatos. Tracing a large-scale Peer to Peer System: an hour in the life of Gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 65– 65, 2002.

[51] M. Mitzenmacher. Editorial: The Future of Power Law Research. *Internet Mathematics*, 2(4):525–534, 2006.

[52] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of the approximations for maximizing submodular set functions. *Mathematical Programming*, 14:265–294, 1978.

[53] A. Norman. Information Architecture and the Emergent Properties of Cyberspace. *InterJournal Complex Systems*, 376, 2005.

[54] W. B. Norton. Internet Service Providers and Peering. Technical Report www.equinix.com/pdf/whitepapers/PeeringWP.2.pdf, Equinix, DRAFT 25.

[55] R. V. Oliveira, D. Pei, W. Willinger, B. Zhang, and L. Zhang. In search of the elusive ground truth: the Internet's AS-level connectivity structure. *SIGMETRICS Performance Evaluations Review*, 36(1):217–228, 2008.

[56] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[57] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *1st IEEE International Conference on Peer-to-peer Computing (P2P2001)*, 2001.

[58] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[59] E. C. Rosen. Exterior Gateway Protocol (EGP). Internet Engineering Task Force: RFC 827, October 1982.

[60] M. Roughan, S. J. Tuke, and O. Maennel. Bigfoot, sasquatch, the yeti and other missing links: what we don't know about the as graph. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 325–330, New York, NY, USA, 2008. ACM.

[61] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian approach to filtering junk e-mail. In *AAAI'98 Workshop on Learning for Text Categorization*, 1998.

[62] G. Siganos, M. Faloutsos, P. Faloutsos, and C. Faloutsos. Power laws and the AS-level internet topology. *IEEE/ACM Transactions on Networking*, 11(4):514–524, 2003.

[63] D. B. Stouffer, R. D. Malmgren, and L. A. N. Amaral. Comment on The origin of bursts and heavy tails in human dynamics. e-print (http://arxiv.org/abs/physics/0510216/), October 2005.

[64] D. Stutzbach and R. Rejaie. Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems. In *Proceedings of the Internet Measurement Conference*, pages 49–62, October 2005.

[65] L. Subramanian, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the Internet Hierarchy from Multiple Vantage Points. In *Proceedings of IEEE INFOCOM*, June 2002.

[66] The Cooperative Association for Internet Data Analysis. http://www.caida.org, 1997 – present (2009).

[67] The Internet Engineering Task Force (IETF). MRT routing information export format. `draft-ietf-grow-mrt-04.txt`, 2004.

[68] The RIPE Routing Information Services. http://www.ris.ripe.net, 1999 – present (2009).

[69] The Working Group on Internet Governance (WGIG). Background Report. http://www.wgig.org/docs/Background-Report.htm, June 2005.

[70] University of Oregon Route Views Project. http://routeviews.org, 1997 – present (2009).

[71] D. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

[72] W. Willinger, D. Alderson, and J. C. Doyle. Mathematics and the Internet: A Source of Enormous Confusion and Great Potential. *Notices of the AMS*, 56(5), May 2009.

[73] W. Willinger, D. Alderson, and L. Li. A Pragmatic Approach to Dealing with High-Variability in Network Measurements. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 88–100, Taormina, Sicily, Italy, October 2004.

[74] Young Hyun and Bradley Huffaker and Dan Andersen and Emile Aben and Matthew Luckie and kc claffy and and Colleen Shannon. The IPv4 Routed /24 AS Links Dataset - 2002–2009. `http://www.caida.org/data/active/ipv4_routed_topology_aslinks_dataset.xml`.

[75] Yuichi Asahiro and Eiji Miyano and Toshihide Murata and Hirotaka Ono. On Approximation of Bookmark Assignments . In *Mathematical Foundations of Computer Science*, volume 4708, pages 115–124. Springer-Verlag LNCS, 2007.

[76] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An information-theoretic approach to traffic matrix estimation. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 301–312, New York, NY, USA, 2003. ACM.

[77] S. Zhou and R. J. Mondragon. Accurately Modeling the Internet Topology. *Physical Review E*, 70, 2004.

[78] S. Zhou and R. J. Mondragon. The Rich-Club Phenomenon in the Internet Topology. *IEEE Communication Letters*, 8:180, 2004.

[79] H. Zimmerman. OSI reference model – the ISO model of architecture for open systems interconnection. In *IEEE Transactions on Communications*, volume 28, pages 425–432, 1980.

[80] D. Zuckerman. NP-complete problems have a version that's hard to approximate. In *Proceedings of the Eight Annual Structure in Complexity Theory Conference*, pages 305–312. IEEE Computer Society, 1993.