Pathsift: A Library for Separating the Effects of Topology, Policy, and Protocols on IP Routing

Vijay Ramachandran
Colgate University
vramachandran@colgate.edu

Dow Street
LinQuest Corporation
dow.street@linquest.com

ABSTRACT

Routing in IP networks is a computation distributed across many routers and subnetworks with inputs specified as low-level instructions in a device-by-device manner. Predicting the impact of input changes is extremely difficult; thus, network designers must not only decide what policies to deploy, but also must encode those policies in a low-level configuration and then run the protocols to see the result. Although several tools exist for the last step, almost none exist to help with the first two, particularly when the network of interest spans multiple domains.

In this paper we present Pathsift, a Python library for generating and comparing sets of paths, designed to permit the exploration of high-level routing policies. Our approach permits evaluation of path quality resulting from different routing policies without packet-level protocol simulation. Our library supports computation of inter- and intra-domain routes and generates visualizations of custom metrics on sets of paths. It thus brings together into one toolchain key components of graph libraries, network simulators, and visualization tools.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Design, Experimentation

Keywords

Internet (IP) routing, BGP, OSPF, routing policies and routing configuration, network simulation

1. INTRODUCTION

Routing in IP networks is a process through which the set of all possible paths in a graph is narrowed to a single data-forwarding path for each source-destination pair. Today, this reduction is often an extraordinarily complex function of multiple interacting protocols, configuration parameters, explicit policy directives, and topology abstractions required for scalability. Thus, studying the impact of configuration choices on the network is difficult. Pathsift is a Python library for generation and analysis of sets of viable data-forwarding routes in an IP network, developed for exploring the design space of IP-network configuration through simulation of route computation. It differs from existing simulation tools because it does not reproduce the packet-level dynamics of Internet routing protocols, but it still contains the IP-routing semantics useful to operational networks. Pathsift provides a high-level workflow because users are not required to translate their configuration to a set of device-specific instructions and gather their results from computation state spread across many simulated devices. Instead, using functions provided by the Pathsift library, users can write—in a familiar programming language—definitions of paths that satisfy desired properties (*i.e.*, a high-level description of the configuration) and definitions of metrics on a single path or sets of paths (*i.e.*, evaluation criteria for the routing output). Given an input toplogy and these definitions, routes satisfying the desired properties can be computed and analyzed based on metrics of interest. Routines are also provided to visualize those metrics and to markup topologies with computed paths and their properties. Because of its different approach, Pathsift facilitates network-wide, rather than device- or route-specific, analysis of route computation; and, because the computation is not executed through a complex, distributed interaction among multiple processes, the role of topology, protocol configuration, and routing policy in the computation can be observed, tweaked, and studied. This is something that is quite tedious to do with existing tools.

In this paper, we describe the design of Pathsift and its contributions to network-routing research. In the remainder of this section, we motivate the need for such a tool by discussing the complexities in understanding IP routing today, and we present several target applications for which such a tool is useful. In Section 2, we review shortcomings in the workflows provided by existing tools that could be used for these target applications, highlighting the differences in Pathsift's approach. In Section 3, we discuss the design of the Pathsift library and its components and give examples of its use. We conclude in Section 4 with a discussion of future work.

1.1 Motivation and Approach

Data-forwarding routes in IP networks are computed through a distributed, highly configurable process, making it difficult to characterize the effects of individual input parameters on overall network behavior. The computation is based not only on the physical topology over which traffic is actually forwarded, but also on the representation(s) of that topology in the multiple routing-protocol processes (*e.g.*, BGP [22] and OSPF [15]) through which network-

Simutools 2012, March 19-23, Desenzano del Garda, Italy
Copyright © 2012 ICST 978-1-936968-47-3
DOI 10.4108/icst.simutools.2012.247758

reachability information is exchanged. This mapping between the physical topology and its protocol representation is highly variable, complex, and dynamic, based on an expressive set of configuration parameters that, in many cases, are set by multiple organizations without coordination. Understanding the computational process of these multiple interacting protocols is quite difficult, making good simulation tools for testing network behavior under various conditions essential.

Viewed from a more abstract level, route computation can be characterized as a series of reduction operations applied to a set of network paths, which we call a *pathset*. The initial pathset for a network consists of all possible simple paths among all pairs of network endpoints. This input pathset is reduced via a *filtering function* to an output pathset which has at most one path per ordered endpoint pair. Ideally, the function is constructed to produce an output pathset that satisfies some set of desirable properties. In reality, the filtering function used in IP networks today is specified not by describing its goals explicitly, but through the configuration of routing protocols used to implement routing, including protocol parameters, explicit policy directives, and topology abstractions required for scalability. The function has two main classes of inputs: the network topology itself, and the list of configuration directives for all the protocol processes comprising the routing system. Network operators have the unenviable job of tweaking low-level device configurations in an attempt to achieve high-level behavior—an output pathset with desired properties. As such, protocol configuration can enter the realm of black art, with operators leveraging both designed and consequential properties of protocols to *coax* behavior from the routing system by changing per-device inputs. This approach can work if the network changes slowly, but is ill-suited for predicting behavior following a sudden change in inputs, be it due to a new configuration (possibly accidental) or a change in topology (*e.g.* component failure). This is ironic, given that rapid adaptation in response to failure is one of the core goals of dynamic routing protocols.

Still, the current approach does achieve an important operational goal: It has allowed the network to grow quite large (*i.e.*, the Internet), even though the emergent global filtering function is extremely complex. The IP-routing system attempts to converge to a single set of "best" paths as quickly as possible, so as to enable packet forwarding along these paths with little downtime. To do so, the computation of the filtering function is distributed among the routers in the network, each one eliminating some number of potential paths from the pathset by executing steps of different protocols, providing partial results to its neighbors. However, because the computation is distributed, and since each reduction step can be tuned locally (with limited global visibility), it is difficult to predict how a change to the configuration or input topology will affect the resulting output. The overall computation is something of a "gray" box, with only loosely bounded behavior, and the only way to know the output is to run the computation. The high-level model of Autonomous Systems (ASes)—administratively separate regions of the network—provides a rough contour of the computation logic, *e.g.*, the possible effect scope of some kinds of input changes, but it is insufficient for predicting the effects of changes at protocol boundaries, or within the inter-AS process itself.

Pathsift's development was inspired by the need for simulation tools to help explore the design space of IP-routing policies. When considering a new routing policy, or studying the effects of a topology change on global behavior, it is useful to have visibility into the intermediate results of the network-wide filtering function. These intermediate pathsets describe "potential" paths permitted by the policies applied thus far, even though most such paths will not be present in the final set of best paths. If the current best path becomes unavailable, it is one of these potential paths that will take its place. It is also helpful if routing policies can be described in terms of high-level attributes of nodes, edges, or the graph itself, rather than detailed directives of individual protocol implementations. It is in these areas where we found a gap in existing tools, leading us to develop Pathsift. On one hand, there are a number of graph libraries (*e.g.*, [6, 11, 24]) that can be used to efficiently model network topologies as graphs and analyze their structural properties. These, however, do not include the logic of IP-routing algorithms, and so pathsets produced by standard graph algorithms may not correspond to those produced by the routing system. On the other hand, there are numerous discrete-event or packet-level simulators (*e.g.*, [7, 16, 18, 21]) that do produce routing tables based on IP-routing-protocol logic. These take as input low-level protocol and device configurations, just like real routers, but include few facilities for changing configuration parameters based on topological properties. These tools generally focus on accurate simulation of protocols, meaning that they efficiently compute "best-path" pathsets, at the cost of making intermediate results less accessible. Pathsift bridges these two types of tools, implementing more IP routing logic than common graph libraries, but adhering to more abstract representations of protocol and policy than routing simulators, while exposing intermediate results in the path-reduction function.

1.2 Target Applications

Pathsift was initially developed to study routing policies in multi-organizational military networks, *e.g*, [8, 19, 27]. Such networks have several properties that distinguish them from the Internet, including ad-hoc topology construction, mobility of infrastructure nodes, and routing policies that are oriented toward achieving mission goals rather than business objectives. In many of these networks there is also a greater opportunity for top-down design of the global routing system, making it feasible to instantiate novel routing policies and paradigms using existing IP-routing technology. The Pathsift library was designed specifically to support this kind of analysis, which would be quite tedious given the workflows available in existing tools (see Section 2).

However, Pathsift also serves a broader interest, particularly as the Internet continues to evolve. Given the importance of the Internet to modern society, and the role of routing policy on the Internet's operation, it is perhaps surprising that there has been relatively little work on *prescriptive* routing policy at the interdomain level. A lack of good tools for studying policy is partially to blame. Much of the research literature studying the effects of policies on routing considers worst-case behavior of protocols, giving sufficient conditions to avoid anomalies such as nondeterministic routing and protocol divergence [9, 10, 14]. Caesar and Rexford [4] review several different policy goals found in commercial networks and explain how BGP can be used to achieve them; however, they offer little in the way of prescriptive guidelines for the development of policy goals in the first place. The required combination of settings at numerous devices is complex enough that testing variations of those policies would be quite difficult without going through a nontrivial exercise of mapping policy variations to device-level configurations. Pathsift provides a high-level programming environment in which the properties of different pathsets can be evaluated using a range of existing and custom metrics. The metrics, and associated filtering policies, are defined in terms of high-level node, edge, and graph attributes. This permits more efficient development of prescriptive guidelines for the design of routing policy, as it obviates the need to perform reconfiguration across multiple devices and to run a simulation for every filtering function that one wants to study. Our approach ignores the packet-level details of protocol dynamics, instead implementing a centralized computation of protocol outputs. Compared to protocol simulators, which produce routing tables for each node as output, our methods allow a user to retain a more global view of the route computation; this makes the job of comparing outputs across different configurations much easier.

There are several concrete research areas in which Pathsift would be useful:

1. *Measuring path quality.* Today's protocol simulators do not retain knowledge of abstract node and edge attributes, such as geography, performance, position in a management hierarchy, *etc.* However, it's reasonable to define a routing policy in terms of producing paths that meet particular criteria based on these attributes. For example, nodes could be annotated with the nation-state of their geographic location, allowing paths to be evaluated in terms of the international borders crossed. As another example, links that use key conduits in the underlying infrastructure (*e.g.*, undersea cables) can be annotated as such, allowing for analysis of associated path metrics important to network resiliency (*i.e.*, physical path diversity). Pathsift makes it easy to define different types of metrics and compare pathsets generated by different policies quantitatively and visually.

2. *Military networks with different policy constraints.* In networks where protocol hierarchy and policies are not predetermined by commercial interests, there is a much bigger set of viable inputs to the routing system, and assumptions about network boundaries, transit requirements, and performance may differ from those traditionally used to study routing policies. Pathsift provides an environment where aspects of the routing system can be changed network-wide, as opposed to device-by-device, permitting improved study of this larger space. For example, attributes of interest to military network routing could include the type of node (*e.g.*, camp, aircraft, vehicle), unit affiliation, mission role, *etc.* In Pathsift, custom attributes can be used to set AS boundaries, characterize paths, or define policies germane to that particular network's goals.

3. *Guildelines for policy design.* The research literature has studied several classic examples of routing policies (*e.g.* [4, 9]), but these often have little relationship to network-management concerns, such as resiliency in failure scenarios. Pathsift not only allows easy evaluation of these higher-level management goals, but also reveals intermediate stages of route computation, where multiple viable paths can be observed prior to the down-select to just a single most-preferred path. Pathsift facilitates experimentation with policies that differ from classical characterizations, of interest due to the changing nature of economic models behind peering agreements. For example, content providers (*e.g.*, Google) and access networks (*e.g.*, Comcast) now play a key role in wide-area connectivity; recent work [13] shows that a significant portion of interdomain traffic now flows directly between content providers, suggesting that the role of constituent Internet networks, and correspondingly their routing policies, may be changing. Given these developments, exploring the space of routing policies—from the perspective of effects on properties of induced pathsets—is necessary for disciplined network engineering.

4. *Evaluating extensions to BGP.* BGP [22], the Internet's interdomain-routing protocol, supports extensions and optional path attributes to allow more expressive route semantics to be exchanged between BGP-speaking nodes. These attributes, *e.g.*, extended communities [23], can be used to communicate policy directives across multiple devices and domains. Pathsift allows for testing the effects of high-level, multi-device policy changes without needing to encode those policies in BGP attributes, or implement corresponding modifications to BGP's decision process. Moreover, BGP has evolved in a somewhat ad-hoc fashion, with extensions often targeting specific operational problems, rather than resulting from rigorous analysis of formal language properties. As such, the bounds of BGP's semantic expressiveness are not fully understood, so there may be desirable policies that are difficult, or even impossible, to implement in BGP's existing mechanisms that would be both fully expressible in Pathsift and of interest to future Internet routing.

5. *Understanding the role of administrative boundaries.* BGP operates at the AS level of abstraction, which promotes scalability: Internal route updates of one network are only seen by other networks when changes to interdomain routes are necessary. Today, Autonomous Systems generally map to organizations (defined commercially or administratively), and very often AS boundaries are treated as something of a given, or constraint, imposed from outside the routing system. It's quite possible, though, that an alternate global scheme for defining boundaries could emerge. For example, it is already common for large networks to use more than one AS number (ASN), and there has been recent interest in various geopolitical organizations to structure AS boundaries along national borders for security, stability, or legislative reasons. This would undoubtably affect path-performance characteristics. Another possibility follows from the recent availability of 4-byte AS numbers [26] and movement toward the use of IPv6, which could ease the requirements for obtaining an ASN, in turn changing the size and nature of the AS graph. Pathsift easily permits automated configuration of AS boundaries based on node, edge and graph properties, making it simpler to study the impact of topology abstractions on computed paths.

2. RELATED WORK AND TOOLS

In order to investigate the types of questions described in Section 1, using a process of simulating route computation and quantifying metrics on the resulting output, one needs elements of three types of tools: (i) graph libraries, which permit abstract modeling of network topologies and calculation of metrics from a network-wide perspective; (ii) discrete-event simulators, which implement the combination of IP-routing protocols used for route selection; and (iii) visualization or plotting tools, which help summarize patterns in the results across all endpoint pairs of interest in the network. Unfortunately, there does not exist a simple workflow that easily combines these three methodologies to study IP routing. In this section we discuss the gaps in existing tools that motivated the development of Pathsift.

2.1 Topology-Protocol-Policy Workflow

As a context for examining available workflows in current tools, consider the following network-design question: Given a particular topology, how tolerant are different sets of routing policies to failure? This question may be answered through simulation in several ways, including: (i) quantifying the number of paths per endpoint pair (satisfying some property), *e.g.*, a "degree of reachability," given the constraints imposed by a routing policy; or (ii) identifying intermediate nodes or edges that are heavily used by the routing solution induced by a policy; or (iii) analyzing the routing solution produced after various sets of nodes and links are removed from the original topology to simulate various failures. Answering these types of questions involves working with a representation of both the topology being tested and the policy being implemented, and using those as input to a tool that implements IP routing protocols to perform the route computation.

Thus, we have a workflow involving a topology, protocols, and policies, and look for tools to provide its components. It seems natural to store the target topology in a graph file format and use a graph library to perform some basic analysis on the topology, including quantifying numbers of paths. It also seems natural to take that topology (perhaps with modifications to simulate failure) and provide it as input, along with the policies being tested, to a routing simulator, using the outputs to evaluate the sets of computed routes. However, the separate tools that exist for these tasks in the workflow have shortcomings; we now discuss these.

2.2 Graph Libraries

Analysis of a topology based on node and link attributes—*e.g.*, finding lowest-latency paths, counting the number of endpoint pairs with a path between them exceeding some minimum bandwidth, finding a network's diameter, *etc.*—is easy when the topology is represented centrally in a convenient graph data structure. Several programming libraries offer routines for manipulation and analysis of graphs in this way, including the Boost Graph Library [24], iGraph [6], and NetworkX [11]. These libraries offer efficient representation of graphs and their corresponding attributes along with implementation of many standard graph algorithms, including random graph generation, path algorithms, and structural analysis (like clustering and diameter). They also support input and output from many standard file formats that can be used to store topologies, such as GraphML [3].

In our example workflow, one could design the target network topology using a visual graph editor and store it, along with its link attributes (like latency and bandwidth), in a file; then, one could write a program to import the topology, iteratively fail selected subsets of nodes and compute paths, and then calculate statistics for those paths, noting any cases when endpoint pairs become disconnected (as a sign of failure susceptibility). However, none of these libraries implement IP-routing algorithms. A major shortcoming is that path algorithms in these libraries are not aware of the levels of hierarchy imposed by the IP-routing protocol stack; in particular, although it's relatively easy to compute lowest-latency paths in a graph by setting link weights to correspond to latency and running an all-pairs shortest-path algorithm (*e.g.* Floyd-Warshall), these paths may not be loop-free paths at the Autonomous System (AS) level of abstraction. Therefore, they may be precluded from the final routing solution by the BGP decision process. A similar problem occurs when the topology is not only divided into subnetworks corresponding to ASes, but when ASes are further divided into multiple routing areas that are communicating using different protocol processes or region designations (*e.g.*, OSPF areas [15]). In addition, path algorithms on graphs tend to assume a global notion of a cost metric, *e.g.*, that "shortest" between two different subsets of nodes should be measured similarly. This assumption is not required for discrete-event simulators or for IP routing in general, which supports autonomy among different subnetworks to define their local-area notions of "best" independently. Although this can be achieved by projecting different functions of metrics onto weights before the computation, existing graph libraries make this complexity of IP routing difficult to model.

Ideally, our workflow would take advantage of the programming flexibility, efficient topology representation, and access to graph algorithms provided by graph libraries, as those components are missing from IP-routing simulators (as discussed below). Graph libraries allow a customized investigation of network properties and development of a set of evaluation functions on paths, but, unfortunately, existing path algorithms lack IP-routing protocol logic. Pathsift was developed on top of NetworkX, providing the additional IP-routing logic needed while maintaining the flexibility of a programming library.

2.3 Discrete-Event Simulators

Network simulators attempt to re-create the control-plane and data-plane dynamics of IP networks; their main achievement is efficiently reproducing in a centralized environment the complicated interactions normally distributed across many devices, while providing the ability to monitor events as they occur. In particular, routing computation is performed by simulating the routing-protocol packets exchanged by participating routers; furthermore, the input, current state, and output of the computation is maintained at each device being simulated. Performance characteristics of the network can be calculated by simulating traffic flow on the network and monitoring packet-level statistics. There are good tools for this type of analysis, both open-source and commercial, including ns-2 [16], ns-3 [17], OPNET [18], and SSFNet [5]. OPNET contains software models for various commercial routers, allowing a user to simulate very realistic network dynamics based on the capabilities of the devices chosen for the network; it contains a GUI for configuring and running simulations and uses a proprietary file format for storing a topology configuration (although this information can be exported to an XML-based file format). The ns-2 simulator (through BGP++ [7]) and SSFNet contain implementations of IP-routing protocols that can be configured on a particular topology through OTcl and DML files, respectively; they produce output traces in text format that can be analyzed through scripts. And, although it is still in development, the ns-3 simulator looks to significantly improve upon ns-2, but it shares many of the fundamental characteristics cited above (including BGP functionality). It does replace OTcl with Python bindings.

While the packet-level simulation offers a view into the dynamic properties of routing algorithms, the encoding of the topology and its attributes into device-specific configurations, and the maintenance of state across many simulated devices, makes tedious the type of analysis needed to answer questions in our example workflow. For example, to obtain sets of paths we'd like to evaluate—even those that could be computed from the graph directly—we would have to develop a mapping from our topology and routing policy to device-specific configurations, creating instructions for each device corresponding to a node in our network. In Figure 1, the top-level workflow illustrates this approach: In order to compute a set of output paths, a mapping M from high-level policies to configuration directives must be applied, which can then be used as input to the simulator; the actual logic of how policies interact to produce the output is a bit of a "gray box" that cannot be observed easily. Then, because simulators maintain the state of the computation across the various devices as protocols are executed, we must (1) determine, from monitoring updates in the network, at what time the route computation is complete; and (2) gather the routing-table or forwarding-table states from all devices and assemble them into a set of paths to analyze. Furthermore, we may have to return

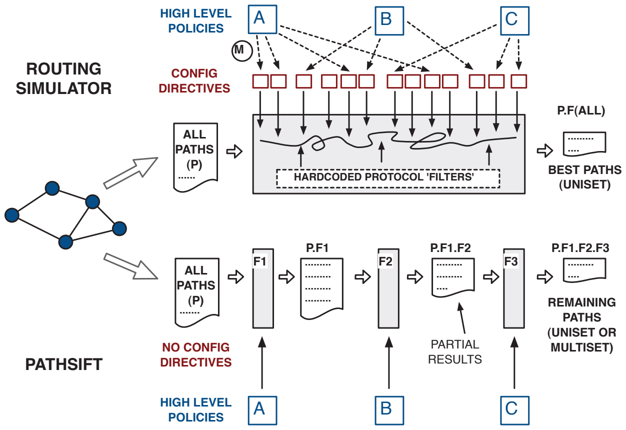


Figure 1: Comparison of network-simulator and Pathsift workflows.

to an alternate representation of node and link attributes, as metrics that depend on attributes that do not have an encoding in a device-specific configuration would not be retained by the network simulation. This is also a problem when answering a question like workflow approach (ii), when trying to analyze utilization of network components—doing so requires more information than the device-specific view on the routing solution provided by a simulator. Finally, to perform analysis like our workflow approach (iii), where we change the topology by removing nodes and links, we would have to re-encode the topology and re-run the simulation for each variant; if we had a more global view of the sets of viable paths, we could more efficiently remove paths containing failed components from our pathset than recomputing a routing solution from scratch. Although the approach used by simulators and routing protocols of keeping just enough information to get to “best” paths quickly is highly efficient, too much information is lost to quantify certain metrics of interest for this type of study.

Fundamentally, because state of the routing computation is maintained for the purpose of protocol execution, which produces a *best* path for each endpoint pair, it becomes difficult to answer questions like workflow approach (i) which seeks to quantify the number of viable paths offered by a particular policy. Furthermore, the intermediate state that can be observed is only enough for the protocol to continually make choices of best paths; we cannot easily extrapolate what alternate paths might be available given failures, changes to tie-breaking rules, or the like. Thus, the types of high-level questions that can be explored with existing tools are limited, even though these tools provide an efficient, accurate way to observe the dynamics of IP-routing computation.

2.4 BGP Solvers

BGP [22], the Internet’s interdomain-routing protocol, is not a lowest-cost-path routing algorithm, but rather a path-vector routing algorithm that supports expressive policies and autonomy among constituent networks. Thus, its routing logic is quite complex, and several tools exist to help simulate BGP’s route computation without the overhead of packet-level simulation between virtual IP protocol stacks. C-BGP [21] (implemented in C) and simBGP [20] (implemented in Python) are two such examples. These BGP solvers are the most similar in spirit to Pathsift, in that they permit testing the effects of routing policy configuration on computed paths at a level of abstraction higher than individual devices. However, these tools still require a mapping of a topology and high-level policy goals (*e.g.*, “do not carry transit traffic” or “use latency to determine IGP weight”) into device-specific instructions; in particular, the configuration files used as input require specification of IP addresses, routing-table entries for BGP sessions, and pattern-match rules to implement route preferences. In addition, querying the tool for output still requires examining individual devices’ routing tables. Thus, these tools do not appreciably simplify the workflow of examining, at a high level, the relationship among pathsets that are induced by different topologies and polices. These tools are primarily designed to check the configuration of a single BGP autonomous system (AS), rather than the interaction among, or studying the effect of boundaries among, multiple ASes. BGP solvers do, however, provide a good way to check the results of a mapping from high-level policy to device configuration, and to examine different ways for a high-level policy specification to be implemented successfully; therefore, we hope to include a mapping from a network configuration in Pathsift to a BGP solver in the future.

2.5 Graph Visualization

The number of potential paths between endpoint pairs grows very quickly as the size of the network increases. These paths, as well as the nodes and edges comprising them, have numerous properties of interest which can be compared, correlated, and measured for various routing approaches—all leading to a massive number of data-points. Effective data visualization is therefore critical to gaining insights into these large, multi-dimensional data-sets. We have chosen to focus on two types of visualization: (i) marking up network diagrams to expose notable subgraphs, attributes of interest, and pathset overlays; and (ii) statistical plots (*e.g.*, heatmaps of pathset metrics) that condense numerical data computed on large pathsets. These types of visual representations make it easier to compare properties of pathsets derived from different routing inputs and to find patterns within the data.

Existing discrete-event simulators, for the most part, do not make overlays of a network visualization easy. For example, OPNET [18] allows a user to highlight a computed best path between one source-destination pair, but does not allow arbitrary visualizations of multiple paths at once. Some graph tools, such as CytoScape [25], Gephi [1], and yEd [28] implement advanced graph-layout algorithms for visualizing network topologies; while their graphical interfaces can be used to generate, read, write, and view IP-routing topologies (even though some, like CytoScape, were originally designed with other applications—in particular, biological analysis—in mind), they do not permit rich visualization of *path* data over the main network visualization.

Since Pathsift is written in Python, and it interfaces with NetworkX [11] for its underlying graph data structure, it is able to take advantage of an existing Python interface to Graphviz [2] for network visualization. We have extended basic Graphviz topology rendering with custom functions, permitting multiple layers of subgraph visualization, which allows the user to markup a network topology with a selected pathset in a manner that highlights properties of interest. In addition, Pathsift takes advantage of matplotlib [12], a Python plotting library, to generate statistical graphs and visualize pathset metrics based on users’ definitions.

3. PATHSIFT LIBRARY

Pathsift is a Python library for generating and comparing pathsets. It is built on top of NetworkX [11], and contains enough routing protocol semantics to allow investigation of IP networks, while still permitting node and edge attributes, and corresponding routing policies, to be defined at a high level of abstraction.

3.1 Architecture and Components

The Pathsift library operates on annotated graphs, computed pathsets, and pathset statistics called *statsets*. Graphs are stored on disk as GraphML [3] files or represented in memory as NetworkX objects (generally directed multi-graphs). Nodes and edges in the graph are marked up with user-defined attributes relevant to the area of study. The library includes built-in support for common properties, such as AS numbers for nodes, and latency and capacity values for edges. Pathsets are computed either from the graph directly, or derived from an existing pathset through the application of a filtering function. Filtering functions are written in terms of the high-level attributes, and passed to a pathset parser.[1] A pathset can be categorized as either a *uniset*, which has at most one path per source-destination pair, or a *multiset*, which may have more than one path per source-destination pair. Note that once routing policy is applied, it is possible for a given pair to have zero valid paths, even in graphs that are connected at the node level (*e.g.*, due to AS loops). Due to resource constraints, especially when working on large topologies, Pathsift's routines support working with pathsets and statsets stored in files on disk rather than fully in memory.

Pathsift supports several algorithms for generating an initial pathset from a graph. Our baseline algorithm computes all loop-free paths for all node pairs, referred to here as an *allpaths* pathset. Because allpaths grows quickly as the size of the network increases, the library also supports setting constraints on the allpaths computation, namely maximum hops and maximum path length based on an edge attribute representing cost. These pathset-generation functions return a multi-pathset, which can then be further reduced through the application of additional filtering functions. Other pathset-generation algorithms directly compute uni-pathsets, such as the library routine for simulating the route computation produced by running BGP and OSPF on the network.

Because the NetworkX library is used for storing and manipulating graphs, existing graph algorithms that operate on topology attributes can be leveraged for route computation, *e.g.*, computation of all-pairs shortest paths. The shortest-paths pathset can serve as a useful reference to assess how far a given policy-induced pathset diverges from at least one definition of optimal. Arguably more important to the study of IP routing, though, is our native implementation of simplified BGP logic, where path segments for each AS are computed using a lowest-cost algorithm (to mirror OSPF [15]), and AS-level paths are computed based on BGP's decision process [22]. This algorithm is particularly useful for comparing BGP best paths to other BGP-constrained multi-pathsets, such as a pathset of all BGP *feasible* paths (*i.e.*, AS-loop-free paths). At present, the BGP pathset generator implements only a subset of the BGP decision tree; other filters, such as removing paths based on the presence a particular AS in the path, are performed by applying reduction functions to a multi-pathset, rather than included in the initial pathset generation. Because route computation via pathset generation does not model packet-level dynamics of routing protocols—instead, policies that describe pathsets are written as high-level functions in Python—no encoding of policy into low-level device configuration is necessary, and intermediate stages of applying policy can be observed. A diagram comparing this workflow to that of packet-level network simulators is shown in Figure 1.

Pathsift contains analysis functions for parsing a pathset and generating path statistics, or statsets. During statset generation, the parser also references the input graph to retrieve node and edge attributes (*i.e.*, most attributes are not stored in the pathset itself).

Objects:

- graph (normally a NetworkX `DiGraph`)
- pathset (list/dictionary or a file)
- statset (list/dictionary or a file)

Library functions:

- AS-setting functions (input: graph, output: graph)
- general / other attribute-setting functions
(input: graph, output: graph)
- pathset generation functions (input: graph, output: pathset)
Note: this includes BGP and OSPF route computation
- pathset filtering functions
(input: pathset, path-filtering function, output: pathset)
- predefined path-filtering functions
(input: path, output: boolean)
- statset generation function
(input: pathset, graph, output: statset)
- visualization functions
(input: graph, pathset or statset, output: visualization)

Table 1: Major components of the Pathsift library.

Like filtering functions, statset-generation logic is also parameterized by user-defined functions to quantify metrics of interest to the user. Statsets are then used to create high-datapoint visualizations; we use Graphviz [2] to render network-style graph markups, and matplotlib [12] for generating statistical graphs (*e.g.*, heatmaps or boxplots). This workflow enables the visual comparison of large pathsets, often derived from a common starting input topology, but with different topology changes (such as failures) or routing policies applied. In this context, a policy could be a complicated set of filters based on abstract node and edge attributes, or a simpler graph-oriented property, such as shortest paths by hop count. Table 1 summarizes the major components of the library.

A final type of function in Pathsift is used to automate the setting of AS boundaries. Due to the way BGP prohibits paths with AS loops by default, the act of selecting AS boundaries is, in itself, an important policy choice (although one that in many networks is more driven by organizational forces or scalability concerns). In fact, a first filter that is often applied in practice is to remove all paths with AS loops, because this artifact of BGP operation alone vastly reduces the number of viable IP-forwarding paths.

Pathsift is written in Python, a language particularly well-suited for this kind of library. As mentioned, Pathsift leverages NetworkX, also written in Python, for its base graph structures, and subsequently can make use of existing bindings to both matplotlib and graphviz (via pygraphviz) for generating custom visualizations. Python also has good support for call-outs to C code when needed for performance (*e.g.*, all-paths computation on large topologies) and libraries for reading XML/GraphML files. Moreover, the Python language itself has several properties that make it ideal for this kind of analysis, independent of the aforementioned benefits: First, Python has powerful built-in functions for easily manipulating lists and dictionaries, which map well to pathsets (lists of lists) and their

[1] Functions are first-order objects in Python; see Sections 3.2–3.3 for examples.

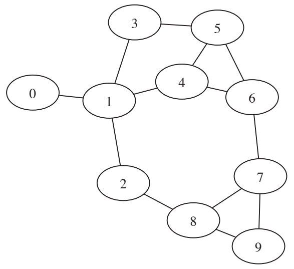


Figure 2: Example topology for Section 3.2.

annotated nodes and edges (dictionaries); second, Python supports using functions as first-order objects, allowing functionality to be parameterized in expressive ways, which we leverage in both pathset filters and stats file generation; finally, as is often claimed, we have found Python to be excellent for rapid prototyping compared to coding similar functionality in C/C++ or Java.

3.2 Using the Library

In this subsection we walk through examples of using Pathsift for analysis of various pathset metrics on an example topology. Consider the simple 10-node graph shown in Figure 2; assume this graph is available as a NetworkX `Graph` object `G`, created using a script or in the Python interpreter. As a baseline pathset, we can create a file containing the set of all simple source-destination paths:

```
allpaths(G, ’G.paths’)
```

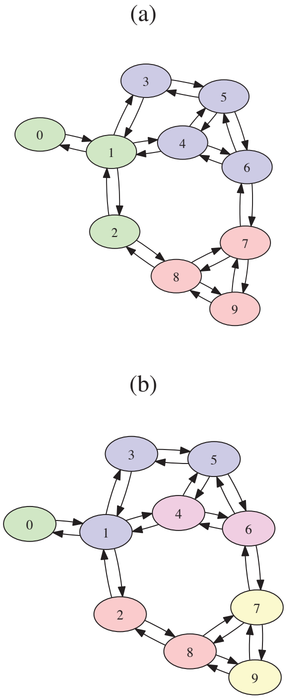
As described above, assignment of nodes to AS numbers (ASNs) is an important, but often overlooked, element of protocol configuration that limits the set of available IP-forwarding paths. In Pathsift, we can manually assign AS numbers to nodes by setting attributes in a script or topology file, or we can programmatically assign AS numbers based other topological attributes using predefined or custom-defined functions. Figure 3 shows our topology after two different ASN-assignment functions are applied: (a) reflects a manual assignment to create a topology `G1`, while (b) reflects executing the following function that sets AS boundaries to encompass connected even- or odd-numbered nodes to create a topology `G2`:

```
def even_odd_as(G):
   cur_asn = 0
   for n in G:
      if ’asn’ not in G.node[n]:
         assign_asn(G, n, cur_asn)
         cur_asn += 1
      for m in G[n]:
         if ’asn’ not in G.node[m] \
           and m % 2 == n % 2:
            assign_asn(G, m, int(G.node[n][’asn’]))

G2 = G.to_directed()
even_odd_as(G2)
```

Although perhaps a trivial (re)assignment, note that topology (b) contains more interdomain links, which could increase the control-plane overhead of BGP, and that the two topologies differ in the locations of edges whose failure would cause connectivity loss due to the creation of discontiguous ASes. We can investigate these differences using Pathsift's pathset generation and analysis routines.

First, we can quantify the number of paths excluded because of our two imposed ASN schemes. To do this, we first define a multiset filter function that excludes paths containing AS-level loops:


(a)

(b)

Figure 3: Pathsift-generated plots (via pygraphviz) of our example topology, marked-up to show AS membership according to two different assignment functions.

```
def bgpff(G, p):
   asp = as_path(as_list(G, p))
   return not detect_path_repeat(asp)
```

The first line translates a node-level path p into its corresponding AS-level path using Pathsift routines that use graph attributes to lookup ASNs. The second line returns a boolean value instructing paths with ASNs repeated to be filtered.

We can compare the number of viable IP-forwarding paths using the heatmap visualization provided by Pathsift. Figure 4 shows the number of potential source-destination paths when (a) BGP is not used, and thus no paths are precluded because of AS-level loops; (b) BGP is used under the manual ASN assignment; and (c) BGP is used under the even-odd ASN assignment. Rows and columns correspond to source and destination nodes, respectively; color intensity strengthens with a greater number of potential paths. In this graph, the maximum number of paths per pair in (a) is 14, but only 10 in (b) and 7 in (c). Note how the even-odd ASN scheme significantly limits the number of paths among nodes in the network because of the addition of AS boundaries (*i.e.*, node-level paths that exit and re-enter the same AS are removed). These heatmaps were produced from multi-pathsets obtained by applying the BGP AS-loop filter defined above to the all-paths pathset for the topology:

```
PS = filtercount(G, ’G.paths’, lambda G, p: True)
PS1 = filtercount(G1, ’G.paths’, bgpff)
PS2 = filtercount(G2, ’G.paths’, bgpff)
heatmap(G.nodes(), PS)
heatmap(G1.nodes(), PS1)
heatmap(G2.nodes(), PS2)
```

The `filtercount` function returns an enumerated pathset containing routes from the input pathset (given here by the filename for the all-paths pathset, `’G.paths’`) that survive the filter function. The

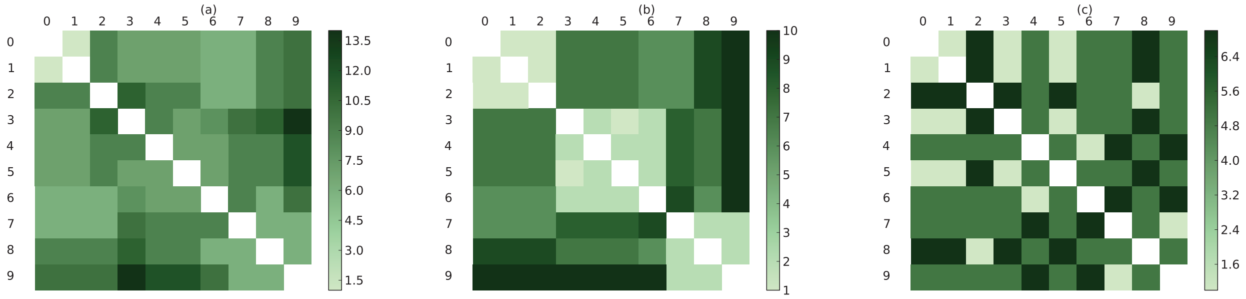


Figure 4: Pathsift-generated heatmaps showing number of viable IP-forwarding paths in our example topology under different conditions: (a) reflects the set of all paths; (b) reflects AS-loop-free paths when ASNs are set as in Figure 3(a); (c) reflects AS-loop-free paths when ASNs are set as in Figure 3(b).

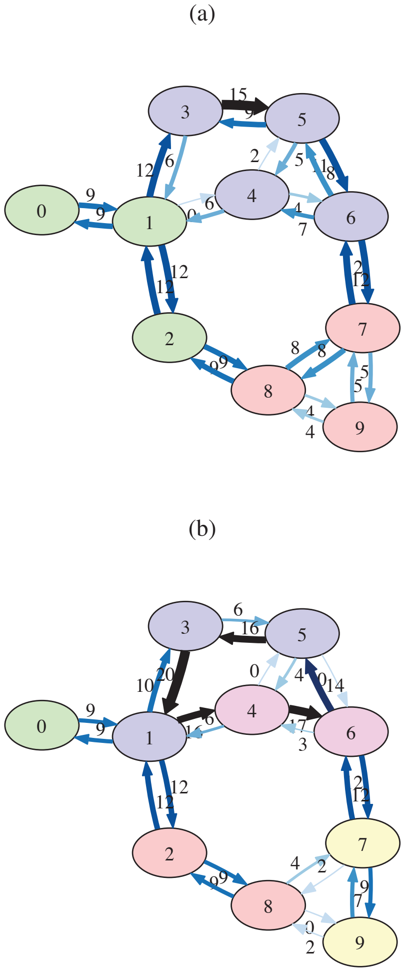


Figure 5: Pathsift-generated markups of the example topology showing edge utilization, *i.e.*, each edge's label, width, and intensity indicates the number of source-destination BGP/OSPF paths that use the edge under the two ASN functions shown in Figure 3.

first line using Python's **`lambda`** syntax to declare a trivial function inline that does no filtering; this is used to produced the data, `PS`, for the all-paths heatmap in Figure 4(a). The second and third lines provide our BGP AS-loop filter as an argument, removing these paths to generate the data for the remaining two heatmaps. Note that the information for these heatmaps, in particular, multi-pathsets that satisfy particular properties, are not readily available in packet-level network simulators; furthermore, AS membership, used here to define pathsets, is generally represented only in configuration settings for BGP sessions between individual devices. This is one example of how Pathsift's high-level approach to network modeling simplifies this type of analysis.

We can quantify the effects of our two ASN schemes on network resiliency. The `bgp_path_dict` function takes a topology with an ASN assignment and simulates the computation of BGP [22] and OSPF [15], namely, lowest-cost paths are selected between endpoints in the same AS, with BGP's decision process (involving shortest AS-path length, in the absence of local-preference settings) used to select paths between endpoints in different ASes. This function produces a uni-pathset, reflecting "best" paths chosen by the protocols, just like the aggregated output of a protocol computation in a packet-level simulator. Figure 5 shows the topology under the two different ASN schemes marked-up using the results of this computation; in particular, the following code to generate the *edge-prominence graphs* uses Pathsift routines that invoke the BGP/OSPF algorithm to indicate the number of paths using each edge in the graph, and then adjust color and line weight in the diagrams:

```
def prominence_plot(G, outfile):
    drawG = G.copy()
    col_edges_bgp_paths(drawG)
    edge_width_by_bgp_paths(drawG)
    networkx.to_agraph(drawG).draw(outfile)
```

We can just as easily run the BGP/OSPF pathset algorithm on m variations of the topology, where m is the number of edges in the original graph G, and each variation is G with a single (different) edge removed, to compute the effect of failures:

```
def test_edge_failure(G):
    ps = bgp_path_dict(G)
    data = []
    for e in G.edges():
        H = G.copy()
        H.remove_edge(e[0],e[1])
        psh = bgp_path_dict(H)
        data.append(compare_pathset_dicts(ps,psh))
    change_boxplot(data)
```

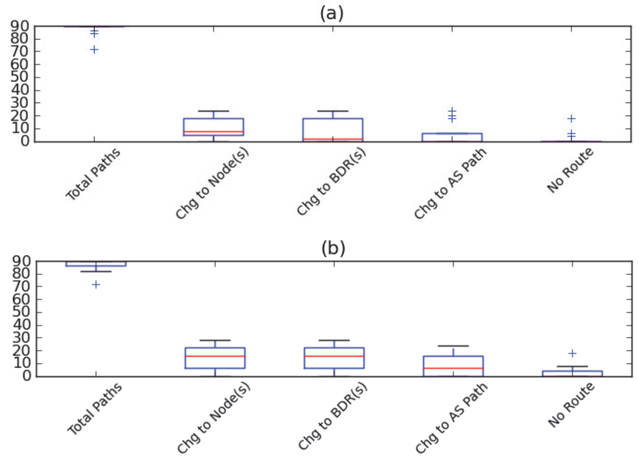


Figure 6: Pathsift-generated boxplots showing the effects of single-edge failures on reachability when BGP/OSPF paths are computed under the two ASN functions shown in Figure 3.

This code produces the boxplots in Figures 6(a)–(b), showing the 5-number summary (median, 1st quartile, 3rd quartile, min, max) of changes to pathsets over all single-edge failures under the two ASN schemes. Red lines show the mean number of paths affected. As expected, the even-odd ASN scheme, with additional interdomain links in the topology, shows a higher average number of paths with changes to border routers or AS paths, resulting in potentially greater BGP update churn due to link failure than the manual ASN-assignment scheme. Obtaining the pathsets and their visualizations involves little code and time with Pathsift, while performing a similar study with a packet-level simulator would involve repeating the simulation of protocol computation from scratch on different input topologies.

3.3 Example: Internet Routing Policy

Gao and Rexford [9] describe a class of routing policies justified by the economics of AS peering on the commercial Internet: It is assumed that neighboring pairs of ASes have either *customer-provider* or *peer-peer* relationships, depending on the settlement agreement between them, and that preferences over routes depend on the economic incentives associated with carrying traffic over these classes of relationships. Gao and Rexford show that this class of policies, though implemented locally, guarantees global stability of the routing system. An important consequence of the policy is that network paths are *valley-free*: The AS-level path computed between any two endpoints consists of some number (possibly zero) of customer-to-provider links, at most one peer-peer link, and some number (possibly zero) of provider-to-customer links. Caesar and Rexford [4] explain how to implement a BGP policy that filters non-valley-free paths.

The Gao-Rexford constraints form a sufficient condition on network stability; in particular, this means that deviations from the rules may still be safe. The constraints may be overly restrictive for a given topology, and changing economics or non-commercial settings may remove the natural incentives for the constraints altogether; *e.g.*, insisting on valley-freeness may preclude paths that have better performance or provide better options in the case of failure. Thus, it's reasonable to ask, given different metrics, what is lost by implementing the Gao-Rexford constraints. Pathsift makes it easy to explore the answer to this question.

Assume that our network topology is stored in a file `T.graphml`, and we have an initial pathset of all AS-loop-free paths in a file `T.AV.paths`. Furthermore, assume we have a dictionary `reln` that describes AS relationships, so that `reln[10][20]` is $-1, 0, 1$ if AS 10 is a customer, peer, or provider of AS 20, respectively. The following path-filter function corresponds to variants of valley-freeness:

```
def grex_vf(G, path, reln, peers=1):
   asp = as_path(as_list(G, path))
   rl = [reln[i][i+1] for i in range(len(asp)-1)]
   vl = [rl[i+1] - rl[i] for i in range(len(rl)-1)]
   return min(vl) < 0 or rl.count(0) > peers
```

For each node-level path on which the function is called, the first line gets the corresponding AS-level path; the second line gets the value of adjacent relationships in that AS path; the third line computes the change in relationship for each pair of adjacent links in the AS path; the fourth line simply checks if those changes are non-monotonic (corresponding to a valley) or whether more than `peers` number of peer links are used. Thus, when `peers` is set to 1 (the default), the filter corresponds exactly to valley-freeness; alternately, we may decide to allow up to k peer links. These two variants can be instantiated and analyzed as follows:

```
T = read_graphml('T.graphml')
ff_vf1 = lambda G, p: grex_vf(G, p, reln)
ff_vfk = lambda G, p: grex_vf(G, p, reln, k)
num_vf1 = filtercount(T, 'T.AV.paths',
         ff_vf1, outfile='T.vf1.paths')
num_vfk = filtercount(T, 'T.AV.paths',
         ff_vfk, outfile='T.vfk.paths')
```

The `filtercount` function will selectively apply the policy filters to the set of viable paths, eliminating paths that fail to meet the defined variations of valley-freeness. At the end of these four lines of code, `num_vf1` and `num_vfk` will have the number of BGP-viable forwarding paths satisfying the single- and multiple-peer definitions, respectively, and the files `T.vf1.paths` and `T.vfk.paths` will contain the filtered pathsets. These pathsets can then be analyzed further based on other metrics, *e.g.*, average latency per hop of paths in the pathset:

```
lat_vf1 = [total_latency(G, p)/len(p) \
   for p in get_list_paths('T.vf1.paths')]
lat_vfk = [total_latency(G, p)/len(p) \
   for p in get_list_paths('T.vfk.paths')]
avg_vf1 = sum(lat_vf1)/len(lat_vf1)
avg_vfk = sum(lat_vfk)/len(lat_vfk)
```

Here, the `total_latency` function represents a path metric defined on attributes of the links defined in the network topology; it can be defined using one line using a workhorse function in the Pathsift library parameterized by the name of the attribute being used for analysis, `latency`:

```
def total_latency(G, p):
   return pwise_edge_attr(lambda x, y: x+y,
     G, p, 'latency')
```

Similar pathset evaluations can be based on metrics predefined in the Pathsift library or that can be defined analogously.

Contrast this with a similar analysis using existing tools: Separate encodings of our valley-free properties, using different pattern-match rules on optional BGP attributes and the AS path, would need to be provided to a simulator; then, the set of routes chosen—limited to one per endpoint pair, based on any tie-breaking rules in the computation—would then be assembled from the routing tables output by the simulator, and a supplemental script with access to link-latency data could perform an analysis over the pathset. Pathsift not only provides a high-level method of describing properties such as valley-freeness, but also maintains a global enough view of the route computation to easily allow aggregate analysis of computed routes.

4. CONCLUSION AND FUTURE WORK

As we have demonstrated, Pathsift allows for rapid experimentation and analysis of a wide range of routing policies written in terms of common and custom node, edge, and graph attributes. By exposing intermediate results of the route computation function, thereby isolating the effects of different topology, policy, and protocol factors, Pathsift facilitates the study of questions not easily answered by other tools.

As noted above, Pathsift was originally developed to study multi-organizational military networks, and so many of the path characteristics we have investigated so far have been particular to that domain. Going forward, we intend to use the library to further study properties and potential policies for Internet routing. We acknowledge that even if a beneficial set of abstract policies were to be discovered, they would not see operation unless they are mappable to specific BGP mechanisms. This drives another area of our future work: building an interface between Pathsift and a BGP solver like C-BGP or simBGP. As mentioned throughout, we expect the translation from high-level policy to low-level configuration to be non-trivial in many cases, and so did not attempt this in our first phase of work. Finally, although we generally run Pathsift functions from a Python shell, visualizations are a key component of our work flow, and so building a GUI to help "organize" rendered images, rather than shuffling through numerous individual graphic files, would be beneficial.

5. ACKNOWLEDGMENTS

The authors would like to thank Dr. Santanu Das (U.S. Office of Naval Research) for funding that partially supported this work. The authors would also like to thank Dr. Glenn Carl and Mr. Terry Gibbons (MIT Lincoln Laboratory) and Dr. Allen Shum (SPAWAR, U.S. Navy) for many helpful technical discussions.

6. REFERENCES

[1] M. Bastian, S. Heymann, and M. Jacomy. "Gephi: an open source software for exploring and manipulating networks." In *Proc. AAAI 3^rd^ Int'l Conf. Weblogs and Social Media (ICWSM'09)*, May 2009.

[2] A. Bilgin, D. Caldwell, J. Ellson, E. Gansner, Y. Hu, and S. North. "Graphviz—Graph Visualization Software." `http://www.graphviz.org`

[3] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. "GraphML Progress Report: Structural Layer Proposal," In *Proc. 9^th^ Int'l Symp. Graph Drawing (GD'01)*, LNCS 2265, pp. 501–512, Sep. 2001.

[4] M. Caesar and J. Rexford. "BGP Routing Policies in ISP Networks." *IEEE Network* **19**(6):5–11, Nov. 2005.

[5] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. "Towards Realistic Million-Node Internet Simulations." In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pp. 2129–2135, Jun. 1999.

[6] G. Csárdi and Tamás Nepusz. "The igraph library for complex network research." `http://igraph.sourceforge.net`

[7] X. A. Dimitropoulos and G. F. Riley. "Efficient large-scale BGP simulations." *Computer Networks* **50**(12):2013–2027, Aug. 2006.

[8] C. E. Fossa and T. G. Macdonald. "Internetworking Tactical MANETs," In *Proc. IEEE Military Comm. Conf. (MILCOM'10)*, Nov. 2010.

[9] L. Gao and J. Rexford. "Stable Internet Routing without Global Coordination." *IEEE/ACM Trans. Net.* **9**(6):681–692, Dec. 2001.

[10] T. G. Griffin, F. B. Shepherd, and G. Wilfong. "The Stable Paths Problem and Interdomain Routing." *IEEE/ACM Trans. Net.* **10**(2):232–243, Apr. 2002.

[11] A. Hagberg, D. Schult, P. Swart, D. Conway, L. Séguin-Charbonneau, C. Ellison, B. Edwards, and J. Torrents. "NetworkX: High productivity software for complex networks." `http://networkx.lanl.gov`

[12] J. Hunter. "Matplotlib." `http://matplotlib.sourceforge.net`

[13] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. "Internet Inter-Domain Traffic." In *Proc. ACM SIGCOMM'10*, pp. 75–86, Aug. 2010.

[14] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. "Shedding Light on the Glue Logic of the Internet Routing Architecture." In *Proc. ACM SIGCOMM'08*, pp. 39–50, Aug. 2008.

[15] J. Moy. "OSPF Version 2." RFC 2328, Apr. 1998.

[16] ns-2. "The Network Simulator." `http://www.isi.edu/nsnam/ns/`

[17] ns-3. "The Network Simulator." `http://www.nsnam.org/`

[18] OPNET Technologies, Inc. "OPNET Modeler." `http://www.opnet.com/solutions/network_rd/modeler.html`

[19] J. Pulliam, Y. Zambre, A. Karmarkar, V. Mehta, J. Touch, J. Haines, and M. Everett. "TSAT Network Architecture." In *Proc. IEEE Military Comm. Conf. (MILCOM'08)*, Nov. 2008.

[20] J. Qiu. "simBGP: Simple BGP Simulator." `http://www.bgpvista.com/simbgp.php`, Apr. 2006.

[21] B. Quoitin and S. Uhlig. "Modeling the routing of an Autonomous System with C-BGP." *IEEE Network* **19**(6):12–19, Nov. 2005.

[22] Y. Rekhter, T. Li, and S. Hares. "A Border Gateway Protocol 4 (BGP-4)." RFC 4271, Jan. 2006.

[23] S. Sangli, D. Tappan, and Y. Rekhter. "BGP Extended Communities Attribute." RFC 4360, Feb. 2006.

[24] J. Siek, L. Lee, and A. Lumsdaine. "The Boost Graph Library." `http://www.boost.org/doc/libs/release/libs/graph/`

[25] M. Smoot, K. Ono, J. Ruscheinski, P. Wang, and T. Ideker. "Cytoscape 2.8: new features for data integration and network visualization." *Bioinformatics* **27**(3):431–432, Feb. 2011.

[26] Q. Vohra, E. Chen. "BGP Support for Four-octet AS Number Space." RFC 4893, May. 2007.

[27] T. Yuan, Y. Chen, and M. LeTourneau. "Joint Tactical Radio System Common Network Services," In *Proc. IEEE Military Comm. Conf. (MILCOM'07)*, Oct. 2007.

[28] yWorks. "yEd Graph Editor." `http://www.yworks.com/en/products_yed_about.html`