BGP Routing Dynamics Visualization for Root Cause Analysis

Luca Cittadini, Tiziana Refice, Alessio Campisano, Giuseppe Di Battista and Claudio Sasso

2007
Abstract

The policy-oriented nature of BGP provides network operators with great flexibility and control over the interdomain routing, nevertheless researchers have shown that these features come at the cost of stability and predictability. In particular, policy interactions often separate, both in time and space, the effects from causes of network events, making it hard to assess and debug network configurations.

To better trace the complex routing dynamics, we developed BGPATH, a system that records, process and visualizes large amounts of BGP data. Our tool provides the user with both separated and aggregated information from multiple vantage points. Namely, BGPATH monitors the number of prefixes routed through each inter-AS link both from single and multiple vantage points. The system also allows the user to deal with path exploration and data sources outages/unreliability, which are common BGP data analysis problems. The implementation relies on several algorithms targeted to process large streams of BGP data. Such algorithms are described and analyzed in details, underlining the potential to evolve towards a completely real-time system for BGP data mining. We show the usefulness and the effectiveness of our approach by means of use cases from BGP real world data.

1 Introduction

The interdomain routing protocol BGP [22] delivers to Internet Service Providers (ISPs) the flexibility to properly control traffic, at least to some extent, via policy specification. Unfortunately, unrestricted policy routing is renowned to be fault-prone, difficultly predictable, and to possibly exhibit undesirable behaviors like lack of convergence and robustness ([25, 16, 24]). Thus, the availability of distributed observation points becomes crucial from an ISP’s perspective in order to check the effectiveness of its own routing policies. In fact, monitoring the interdomain routing evolution from multiple vantage points helps verify whether specific prefixes are correctly reachable from the whole Internet, and helps assess the impact of traffic engineering configurations. Also, detecting and debugging misconfigurations or faults would be extremely hard without the support of a such an infrastructure. Further, customer ISPs want to evaluate the quality of service offered by their own providers. Finally, forensic analyses usually take advantage of remote monitors in order to identify interdomain routing events and their causes.

Those needs led ISPs to place observation points inside their own network, making them (at least partially) available for public usage. For example, route servers and looking glasses are very widely spread. Unfortunately, such vantage points provide limited benefits since they can only show the current state of the network, without storing such information.

In order to investigate routing dynamics over time, RIS [4] and ORV [6] projects spread worldwide many collection boxes which continuously gather routing data from the network. Collected data are permanently stored and made publicly available. Those collection boxes currently receive an average of about 1,500 updates per minute, with peaks of more than 50,000 updates per minute.

Manually analyzing such data is unfeasible because of their huge amount. Thus, several tools have been proposed to support the analysis. [19] proposes LinkRank [3], a tool that visualizes the routing activity of the whole Internet within a user-specified time window. LinkRank allows to assess the scope and the impact of major network events. However, large portions of the Internet can experience routing changes even in a small time window. Hence, the resulting graphs can be very large and thus difficult to analyze at a glance. Adopting a different approach, [11] devised BGPlay [1, 2] to look
at routing changes only affecting a single user-specified destination prefix. In fact, [11] argues that a network operator is usually interested in what happens to a specific portion of the network (e.g., his provider’s network), rather than monitoring changes of the whole Internet. On the other hand, this approach does not relate the routing of the input prefix to the overall Internet dynamics. Finally, BGP-Inspect ([7]) only considers a limited number of collector peers, and only fulfills a restricted number of well-known queries. Summarizing, BGPlay and BGP-Inspect can effectively trace specific changes, but do not help an ISP to investigate the cause of those changes. Conversely, [19] allows to spot major events (e.g., link faults or router resets) but does not help the user understand how such events affected specific prefixes.

To overcome such limitations and bridge the gap between the approaches described above, we developed BGPath. This tool features a stream-based back-end system, which (i) collects data from many monitors, (ii) performs data cleaning processes that cope with both long-lasting un reliabilities of data sources and short-lived dirty data bursts, (iii) computes metrics over input data within strict time constraints. As shown in Section 4, BGPath also provides a user with an effective graphical interface that, given a routing change $c$:

- places the ASes involved in $c$ according to the customer-provider hierarchy of the ISPs
- visualizes both aggregated and disaggregated information for each interdomain link appearing in $c$
- traces different paths to the prefix involved in $c$ over time.

BGPath is publically available at http://nerodavola.dia.uniroma3.it/rca/.

We provide basic background and notation in Sections 2 and 3 respectively. Our data sources are described in Section 6. Section 5 gives an overview of the architecture of BGPath, while the algorithms the tool relies on are detailed in Section 7.

## 2 Background

The Internet is divided into administrative domains called Autonomous Systems (ASes). The Border Gateway Protocol (BGP) [21, 22] is the routing protocol used to exchange reachability information between ASes. Two ASes whose routers exchange routing information using BGP are said to have a peering between them. A BGP router stores in its Routing Information Base (RIB) the prefixes it can reach, and for each of them an AS-path. An AS-path, also called route, is the sequence of ASes used to reach the destination prefix. Routes are propagated by BGP messages called updates. BGP is an incremental protocol: once two BGP routers establish a peering, they exchange their whole RIB each other; this process is called table transfer. Further updates are sent only if a route changes, in response to network events (e.g., link failure, router reset, or policy change).

To obtain information about the Internet routing dynamics, projects - such as the RIPE NCC’s Routing Information Service (RIS) [4] and the University of Oregon’s RouteViews Project (RV) [6] - spread around the world several passive collection boxes, called (Remote) Route Collectors (RRCs). Each route collector peers with several BGP routers, called Collector Peers (CPs), belonging to various ASes. The routing tables of all RRCs and the updates they receive are periodically dumped, permanently stored, and made publicly available. Some collector peers provide information about all the prefixes on the Internet, while others only provide information about a subset of them. We call the former full collector peers, the latter partial collector peers.

## 3 Preliminaries and Notation

Previous research works proposed several models of the BGP protocol to capture its most important features. Most of those models represented an AS with a single router, while in other models (e.g., [20]) an AS can contain multiple routers. The first approach has the clear advantage of simplicity, but can be too coarse-grained to describe the impact of intra-AS dynamics on interdomain routing. In fact, [17, 26] showed that complex internal structure of an AS can lead to unexpected protocol interactions that exceed the borders of the AS itself. On the other hand, the second approach has the major drawback that there is no currently available dataset that can provide a reliable fine-grained description of the internal structure of an AS. Such information could be inferred, but the inference mechanisms proposed so far are neither complete nor accurate.

In this section we introduce our notation and define new metrics which rely on the model defined in [9]. These metrics are valuable regardless of the capability of the underlying model to capture ASes’ internal structures.

We consider the following sets. $\text{ASes}$ is the set of all known ASes, $\text{ASes} = \{1, \ldots, 65535\}$. $\text{Times}$ is the set of all instants of time when any route collector receives a BGP update from any collector peer. A total order $<$ is defined on $\text{Times}$. $\text{CollectorPeers}$ is the set of all collector peer identifiers. Usually, each CP peers with only one route collector. Hence, we can use its IP address as identifier. When the same CP peers with more than one route collector, we assign a different identifier to each of its peering sessions. $\text{Prefixes}$ is the set of all known prefixes.

2
An AS-path (or simply path) π is a sequence of ASes, π = (a0, ..., ai), where ai ∈ ASes. a0 is called origin. The empty path is denoted by φ. ASPaths is the set of all known AS-paths. A pair (ai+1, as) of ASes that are adjacent in some AS-path is an edge. We consider the edges as directed, i.e. (v, w) ≠ (w, v). We say that a path contains an edge.

An update u is a quadruple (cp, p, π, t), where u.cp ∈ CollectorPeers is the collector peer that collected the update, up ∈ Prefixes is the prefix the update refers to, u.π ∈ ASPaths is the route announced by the update, and u.t ∈ Times is the time when the update is collected. If u.π = φ then u is an announcement, otherwise it is a withdrawal. Updates is the set of all known updates. The last update u that collector peer cp received for prefix p, before time t, is denoted ℓcp(p, t). Formally, ℓcp(p, t) is such that ℓcp(p, t), t ≤ t and ∃u ∈ Updates | u.cp = cp ∧ u.p = p ∧ ℓcp(p, t), t < u.t ≤ t.

Every update u causes a routing change. A change c is a quintuple (cp, p, πold, πnew, t) where c.cp = u.cp, c.p = u.p, c.t = u.t, πold = ℓcp(p, c.p, c.t), πnew = ℓcp(p, t). We define local rank and global rank, two metrics that describe the status of a specific BGP peering, respectively, from a single vantage point perspective and from a cross vantage point perspective. While the former has been introduced in [19], the latter is, as far as we know, a novel concept. Given a collector peer cp, the local rank of an edge e at time t is defined as the number of prefixes whose path at time t contains e, as observed by cp. Namely, lrank(cp, e, t) = |P(cp, e, t)|, where P(cp, e, t) = \{p ∈ Prefixes | e ⊆ ℓcp(p, t), πold = (cp, p, π, t) ∈ Updates | u.cp = cp, u.t = t, e ⊆ u.π\}. We define the global rank of an edge e at time t as

\[ grank(e, t) = |P(e, t)|, P(e, t) = \bigcup_{cp ∈ CollectorPeers} P(cp, e, t). \]

Figure 1 illustrates values of local and global ranks of edges of a portion of Internet at a certain time t. For example, the label (1, 2, 3) on edge (as1, as2) states that lrank((as1, (as1, as2), t) = 1 since cp1 sees just the green dashed prefix traversing (as1, as2). Also, grank((as1, as2), t) = 3 since (as1, as2) is traversed by all three prefixes. Note that grank((as1, as2), t) ≠ \sum_{i=1}^{3} lrank(cp_i, (as1, as2), t). According to definitions of local and global rank, we label the pair (cp2, as2) only once, even if these ASes are connected by multiple peerings.

Intuitively, while the local rank measures the number of prefixes that are observed passing through an edge by a single collector peer, the global rank measures the number of distinct prefixes that are observed passing through an edge by any CP ∈ CollectorPeers. We stress that both measures are extremely valuable for a network operator, since they can provide insights on, for example, the soundness of an interdomain peering or on the impact of traffic engineering choices.

As shown in [9], local rank variations exhibit a flow-like behavior. Although this is an interesting property, it is out of the scope of this paper. The reader is encouraged to check [9] for a complete description of the model.

4 BGPATH in Action

Because of the huge size of BGP routing data and the high complexity of the protocol behavior, visual support for analyzing routing information is as important as the information itself, in order to effectively monitor and diagnose Internet dynamics. Moreover, selecting the most relevant information from the large amount of available routing data is far from trivial and depends on specific needs.

According to the BGPlay approach, we developed BGPATH, a tool that visualizes a user-specified path change and provides useful information to help identify its root cause. We integrated our BGPATH within BGPlay, in order to allow the user to graphically select a particular path change.

Through the following scenario, we describe the user interface of BGPATH, in order to show the effectiveness
of its visualization approach.

Consider a network operator interested in monitoring prefix $p = 159.14.0.0/16$ on January 1st 2007. He can query BGPlay, and observe how routes to $p$ change over time during this time interval. Figure 2 is a snapshot of the resulting BGPlay plot. AS7328, that originates $p$, is highlighted in red and placed in the center of the window. Available paths that do not change within the time interval are represented by dashed lines, while routing changes are visualized by colored solid lines moving from one path to another (see [11] for further details about BGPlay). The operator selects a specific path change $c$, with the old path $c.\pi_{old} = (15837, 8881, 2914, 10910, 7328)$ and the new path $c.\pi_{new} = (15837, 8881, 3356, 12178, 7328)$ displayed during the BGPlay animation and asks BGPATH for further information on it.

Figure 3 shows how BGPATH displays the input path change $c$. Edges belonging to the old path are represented by dashed lines, while edges in the new path are drawn with solid lines. Data about $c$ are reported in the top of the window. In order to provide a network operator with a familiar representation of this topology, we place ASes in the graph according to the customer-provider hierarchy (see e.g., [15]). Namely, tier-1 ISPs are represented by the top nodes in the graph, and customers are placed just below according to their position in the hierarchy. In Figure 3, for example, the two top ASes are well-known tier-1 providers (NTT and Level3, namely). Besides being natural from an operator’s perspective, this representation is also helpful to understand the relevance of network events related to the path change $c$. In fact, [28] shows that events located in different areas of the hierarchy usually have significantly different impact on the network. In Section 7.3 we explain in details how we draw the topology.

Before further analyzing the path change, the user can check which data sources provide insights about $c$ and whether they can be considered trustworthy. In fact, BGPATH presents in the left panel a list of collector peers related to $c$, along with the ASes they belong to and grouped by remote route collectors they peer with. ASes containing these collector peers are marked on the graph with an eye-shaped icon. Also, the system warns the user about collector peers that are potentially unreliable for the analysis of $c$. Section 7.1 and 6 provide further details about how we identify unreliable collector peers. The ability to spot potentially dirty data gives the user a high level of trust on the analysis he is carrying out. Thus, we believe it is a very important feature of BGPATH and, as far as we know, no other tools provide such an information.

To assess the scope of path change $c$, the network operator can visualize the history of all routes chosen by $c.cp$ to reach $p$ within a fixed time window around $c.t$. As shown in Figure 4(a), the tool displays a chart where time is on the $x$ axis and a set of distinct paths, including the empty path, on the $y$ axis. For example, the transition from $(15837, 8881, 2914, 10910, 7328)$ to $(15837, 8881, 3356, 12178, 7328)$ at time $t$ on the plot represents the input path change. In our example, $c.cp$ switches path to the prefix $p$ only twice (from $c.\pi_{old}$ to $c.\pi_{new}$ and back) within a short time interval. This suggests that the cause of the change $c$ could be a short-lived event with a very limited impact on the network. Instead, Figure 4(b) shows an unstable prefix which keeps flapping between path $(6067, 3549, 137)$ and $(6067, 174, 137)$. Hence, looking at the path evolution, it is possible to verify whether $c$ experiences any specific pattern of changes, e.g. if it is just a temporary oscillation, or if it belongs to a persistent dynamics. The interface of the tool also allows to select other prefixes belonging to the same origin AS, and to draw their path evolution within the same chart. This way an ISP can monitor the behaviors of different prefixes it announces on the network, and the impact of per-prefix routing policies (e.g., interdomain traffic engineering configurations). Finally, the visualization of path evolutions is extremely useful to detect sequences of routing changes due to BGP path exploration.

Once assessed the data sources’ reliability and the scope of the path change $c$, the network operator can look at the evolution of the number of prefixes routed through every edge affected by $c$ (i.e. belonging to either $c.\pi_{old}$ or $c.\pi_{new}$) in order to have a rough estimate of the traffic load born by this interdomain link. BGPATH can plot both $\text{grank}$ and $\text{lrank}$ evolutions for all visualized edges. In the following, we focus on the edge $e = (10910, 7328)$. Figure 5(a) shows that, according to both $\text{grank}$ and $\text{lrank}$, $e$ carries a steady quantity of prefixes over time, but exhibits a discontinuity right at time $c.t$: for a couple of minutes, in fact, all the prefixes that passed through $e$ moved somewhere else. This is a clear evidence of some kind of problem affecting $e$, since more than 20 collector peers contribute to the global rank. At this point, the network operator can guess that the path change $c$ he selected has been probably triggered by some minor event happened on link $(10910, 7328)$. Moreover, he could continue his analysis by extending the time interval to check if the shortage occurred again some time later, in order to identify potentially troubled links (Figure 5(c) exemplifies $\text{grank}$ and $\text{lrank}$ of a probably faulty link). Zooming in the $\text{lrank}(e)$ plot (Figure 5(b)) we can observe the lack of synchronization between $\text{lrank}$ and $\text{grank}$. Note that, in general, $\text{grank}$ experiences a delay in recording a “negative” variation, while it is much more reactive in recording a “positive” variation, with respect to $\text{lrank}$. 

Figure 2: A path change displayed by BGPlay.

Figure 3: A path change displayed by BGPATH.
5 A Stream-Based Architecture

The user interface briefly described in Section 4 relies on a back-end system that gathers a huge amount of BGP raw data (i.e., BGP RIB dumps, updates, and session logs) and efficiently computes local and global ranks of all the Internet interdomain links and reliability information for all the available collector peers. Figure 5 outlines the main components of the back-end system.

Moreover, the back-end handles an input data stream as follows:

- BGP updates, table dumps and session logs (if available) are collected by the Retriever from different data sources and converted in a common format. Section 6 describes the data stream.

- Input data undergo a Reliability Screening, which discards collector peers that provide inconsistent information within a specific time interval. The Reliability Screening process is explained in Section 6.

- Then the Ranker processes “reliable” BGP routing data (both rib dumps and updates), and computes local and global ranks for all the collector peers and for all the edges, using the algorithms described in Section 7.2.

- In parallel, the Table Transfer finder combines “reliable” routing data with session state logs, and identifies table transfers together with their alleged duration. Section 7.1 details the algorithm implemented by the Table Transfer finder.

In order to deal with streams of BGP updates, we designed the architecture of the back-end system to be compliant with very strict time constraints. Namely, available data sources currently provide BGP updates grouped into chunks, so we have to fully process a chunk before another one comes out of the stream.
Figure 5: Local and Global ranks. (a) A stable link with a brief service discontinuity. (b) An insight on the discontinuity. (c) A link which is experiencing some malfunction.
According to the average per-chunk performance reported in Section 7, the current implementation of BGP-ATH is able to process a 15-minute data chunk in less than 3 minutes, including the time required to download data from the collection boxes. This implies that the output is ready for queries long before the next chunk becomes available.

We evaluated the performance of the system using the testbed described in Table 1. Note that the testbed is an average platform. Therefore, any common machine could effectively run our tool.

<table>
<thead>
<tr>
<th>Project</th>
<th>all CPs</th>
<th>reliable CPs</th>
<th>full &amp; reliable CPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIS</td>
<td>425</td>
<td>381</td>
<td>72</td>
</tr>
<tr>
<td>RV</td>
<td>101</td>
<td>101</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 2: Statistics on CPs.

6 The Data Stream

Our tool relies on BGP data obtained from both RIS [4] and RV [6]. Overall, those projects currently provide 526 collector peers, 30% of which are full collector peers (see Table 2 for details).

Examples and statistics presented in this paper refer to a portion of the data stream collected from 12/26/2006 to 01/02/2007, and we call this time interval reference week. The reference week data contains 320,678,893 updates (nearly 46M updates per day on average) with 7,537,378 distinct AS-paths on 70,078 distinct peerings and 24,493 distinct ASes. The number of observed prefixes is 235,725.

In order to discard data coming from faulty or misconfigured collector peers, once a week we perform a data cleaning step, called Reliability Screening. Several reasons can lead to the unreliability of a collector peer, including bugs in routing or collection software (see [18] for details), major asynchronies between the collector peer and its route collector, and poor standard compliance (e.g., some vendors do not implement highly recommended optional timers).
The screening of a collector peer $cp$ over a time interval $[t_{start}, t_{end}]$ is executed as follows: (i) we make a local copy of the RIB of $cp$ at $t_{start}$, (ii) we modify the copy according to the updates collected by $cp$ during $[t_{start}, t_{end}]$, (iii) we compare the modified copy to the RIB dumped by $cp$ at $t_{end}$, (iv) we decide if $cp$ is reliable by evaluating the ratio between number of mismatches and the average size of its RIB.

Reliability Screenings performed during several experiments led to the detection of a major problem that affected RIS route collectors since May 2005. Overall, the problem affected 44 collector peers, 12 of which were full collector peers (see Table 2 for details). Contacting the RIS maintainers resulted in that problem being fixed by Jan 2nd, 2007.

7 Algorithms

We devised efficient algorithms which are able to fulfill the time constraints outlined in Section 5. In this Section, we describe the algorithms BGPATH exploits for the Table Transfer Finder and the Ranker modules. We also describe one of the algorithms used by the layout facility embedded in the graphical user interface.

7.1 Identification of Table Transfers

The analysis of BGP data is very sensitive to various types of “noise”. The Reliability Screening discards collector peers that are unreliable in the long term, but some collector peers provide “noisy” data only within short periods of activity. In particular, we are concerned with session resets between any collector peer and its route collector, because, after a session reset, a collector peer sends its whole BGP table to the route collector. This huge set of updates only refers to a local event, and possibly hides other events happened in Internet. When available, BGP session state messages provide an evidence of such events, recording transitions from/to the BGP state “6=Established” [22]. Unfortunately, only RIS collector peers supply their state messages. Thus, we devised Algorithm 1 in order to identify, with a reasonable accuracy, all the BGP table transfers occurred between collector peers and route collectors in a given collection of BGP updates. We stress that our algorithm is valuable even when state messages are available, because it estimates the duration of a table transfer. This information cannot be extracted from state messages alone and it is necessary for pinpointing updates caused by resets in order to avoid taking them into account during the analysis.

Real world data illustrates that BGP session resets happen with a non negligible frequency in the Internet. Namely, RIS collectors reported 71 session resets during the reference week, despite the fact that we only considered collectors that successfully passed the Reliability Screening.

Although [27] has already described an approach to identify table transfers, we found out that it does not scale well over a large set of collector peers. Namely, [27] does not give any computational complexity analysis of the proposed algorithm, however it is easy to find that it requires $O(nwσ)$ time and $Ω(ωσ)$ space, where $n$ is the number of processed updates, $σ$ is the maximum number of updates per second, and $ω$ is the number of seconds of a time window that is used to scan the updates (whose maximum size spans over two hours). Time complexity is still feasible if $ωσ$ is $o(n)$. However, processing a huge set of data requires a lot of memory. Using secondary memory instead significantly affects performance. Also, performing several scans, for example one for each collector peer, would not be acceptable, since each scan can take up to a dozen minutes.

Algorithm 1 was designed to tackle the above space complexity problem. Given a stream of BGP updates, this algorithm can pinpoint a table transfer from any collector peer, with a reasonable approximation of the start time of the transfer and its duration.

We distinguish between pumping and vacuum table transfers. The first corresponds to the typical large set of announcements from a router that notifies a neighbor about its own full routing table. The latter refers to an explicit withdrawal of a large set of prefixes, potentially the whole routing table. Vacuum transfers should never happen, according to the RFC specifications ([22]). In fact, no state transition in the BGP finite state machine can cause the withdrawal of all the known prefixes. On the other hand, experience shows that they actually occur with non negligible frequency. Such abnormal table transfers are symptomatic of something unusual happening between the route collector and the collector peer.

We define a threshold function $ρ(cp,t)$, $ρ: (CollectorPeers × Times) → ℝ$ that, given the RIB size of the peer $cp$ at time $t$, provides the threshold number of prefixes to check against to decide if a table transfer occurred. Different choices of the threshold function can lead to quite different results in terms of alleged session resets detected. Our current $ρ$ function is the weighted average of the RIB size of $cp$ until time $t$, where the weights are the amounts of time within which the RIB had a certain size. This choice has at least two advantages. First, it is computable in $O(1)$ time, without memory penalties (only one value per collector peer needs to be retained in memory). Second, it is hijacking-insensitive (i.e. even large prefix hijackings have a low influence on $ρ$), because hijackings usually last for a short time. Namely, let $t_i$ be the time of the $i$-th change of the RIB size of $cp$, let $R(t_i)$ be the size of that RIB at time $t_i$, and let $t_0$ be the first considered
time, then
\[
\rho(cp, t) = \delta \sum_{i=1}^{k} R(t_{i-1}) \cdot (t_i - t_{i-1}) / (t - t_0).
\]

For each collector peer, we keep in memory a set of known prefixes \((RIB_{tt})\), along with two dictionaries, \(P_W\) and \(P_{RA}\), that map a prefix to the latest time it was withdrawn and re-announced\(^1\), respectively. \(P_W\) (\(P_{RA}\)) only retains prefixes that were withdrawn (re-announced) in the last \(\omega_w\) (\(\omega_r\)) seconds. Whenever an update \(u\) is received, we classify it as a new announcement, a withdrawal, a re-announcement, or a path change. In the first case, the prefix is simply inserted into the current \(RIB_{tt}\). In the other cases, the time value associated with \(u.p\) in \(P_W\) or in \(P_{RA}\) is updated with \(u.t\). Then, if \(P_W\) (\(P_{RA}\)) contains almost all the prefixes in the \(RIB_{tt}\) (\(\rho(u.cp, u.t)\) defines the threshold), we say that a table transfer occurred. Algorithm 1 illustrates the approach with a pseudo-code notation.

In our experiments we set \(\delta = 0.99\) according to the results in [27], and we chose \(\omega_w = \omega_r = 500\) seconds.

Using Algorithm 1, we are able to identify occurring table transfers in a single sweep of all the updates \(O(np)\) time, where \(n\) is the number of updates and \(p\) is the number of prefixes, only using \(O(p)\) memory space (considering constant the number of collector peers). In the following we show the complexity analysis of the algorithm. First of all, we assume \(O(1)\) time for all the operations on hash tables (e.g. insertion and deletion of elements in a set, retrieval of the value associated to a specific key, evaluation of the cardinality of sets). As shown before, evaluating \(\rho(u.cp, u.t)\) takes \(O(1)\) time. Without loss of generality, we only describe the complexity for processing input withdrawals. Similar arguments hold for path changes or re-announcements. Note that \(\min_{p} P_W(p)\) is never actually computed, as this value can be kept in a proper variable \(\min_W\). Hence, the whole complexity is due to the step that drops outdated entries from the map \(P_W\). In the worst case and without further performance optimization, we scan all prefixes in \(\text{keys}(P_W)\). Observe that \(\text{keys}(P_W)\) can never contain more than \(p\) distinct prefixes. Thus, the worst case time complexity is \(O(np)\). Moreover, using \(\min_W\), the cleaning step is only necessary when \(\min_W < u.t - \omega_w\). Space complexity is trivially \(O(p)\) since we keep in memory \(RIB_{tt}\), \(P_W\), and \(P_{RA}\).

Thanks to the performance of Algorithm 1, we can account for table transfers of all collector peers at once. Our experiments show that we are able to process the data in the reference week in less than half the time spent

---

\(^1\)In this context, a re-announcement is a change \(u\) such that both \(u.\pi_{new}\) and \(\ell_{u.cp}(u.p, u.t)\) are non-empty.

---

**Algorithm 1** Identify Table Transfers of a Collector Peer

**Require:** an update \(u\) is received from \(u.cp\) at \(u.t\)

1. if \(u.p \notin RIB_{tt}\) then \{new announcement\}
   2. \(RIB_{tt} \leftarrow RIB_{tt} \cup u.p\)
   3. else \{In this case, \(u\) can be a withdrawal, a path change or a re-announcement\}
      4. if \(u.\pi = \emptyset\) then \{withdrawal\}
         5. \(RIB_{tt} \leftarrow RIB_{tt} \setminus u.p\)
         6. \(P_W(u.p) \leftarrow u.t\)
         7. for all \(p\) such that \(P_W(p) < u.t - \omega_w\) do
            8. remove \(p\) from \(P_W\)
      9. end for
     10. if \(|\text{keys}(P_W)| > \rho(u.cp, u.t)\) then
         11. output a vacuum TT in \([\min_{p} P_W(p), u.t]\)
         12. clear \(P_W\)
     13. end if
     14. else \{re-announced prefix\}
        15. \(P_{RA}(u.p) \leftarrow u.t\)
        16. for all \(p\) such that \(P_{RA}(p) < u.t - \omega_r\) do
           17. remove \(p\) from \(P_{RA}\)
        18. end for
        19. if \(|\text{keys}(P_{RA})| > \rho(u.cp, u.t)\) then
           20. output a pumping TT in \([\min_{p} P_{RA}(p), u.t]\)
           21. clear \(P_{RA}\)
        22. end if
        23. end if
     24. end if
using the approach in [27]. On the other hand the algorithm in [27] can be in some cases more accurate in identifying the exact beginning of a table transfer, because it performs several backward scans.

Table 3 summarizes the most relevant performance information about our implementation of Algorithm 1, ran over the reference week. Considering only the 169 full collector peers Algorithm 1 identified 90 table transfers, each one corresponding to a session reset.

<table>
<thead>
<tr>
<th>Execution (wall clock) time</th>
<th>11:30:32 (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory used (peak)</td>
<td>1.2 Gbyte</td>
</tr>
<tr>
<td>15’ chunk avg processing time</td>
<td>62 sec</td>
</tr>
</tbody>
</table>

Table 3: Algorithm 1 - Time and memory performance

7.2 Computation of Local and Global Ranks

Monitoring the evolution of local and global ranks over time allows to check the soundness of interdomain links and peerings, and helps pinpoint macro-events which affect either the physical or the logical network topology (e.g. interdomain link faults/restorations, BGP router faults/restorations, or BGP peering shutdowns/setups). Such network events can have a dramatic impact on Internet routing. [9] describes a methodology to infer interdomain macro-events by analyzing local and global ranks.

Computing rank values can seem straightforward, but the real challenge is doing it efficiently for hundreds of collector peers at once. This huge amount of data requires very careful tuning of algorithms and data structures.

[19] provides a publicly available tool for visualizing local ranks, but it uses data only from a limited set of collector peers. Moreover, [19] does not describe in detail how local ranks are computed on the server-side, so no performance comparison with their approach is possible.

We define Algorithms 2 and 3 to compute local and global ranks of an edge, respectively. Both these algorithms require to retrieve the last path received by a collector peer for a specific prefix, i.e. \( \ell_{u, cp}(u.p, u.t) \). To store this information, we use, for each collector peer \( cp \), a running table \( RIB_e \) that maps each prefix \( p \) observed by \( cp \) to the AS-path that \( cp \) currently uses to reach \( p \).

For each collector peer \( cp \), Algorithm 2 keeps track of the running local rank of each edge observed by \( cp \). Whenever a new update is received, Algorithm 2 increases local ranks of all edges in the new AS-path (if any, i.e. if the new update is not a withdrawal), while it decreases local ranks of all edges in the previously used path (if any).

Likewise, Algorithm 3 uses a running counter \( \lambda(\text{edge}, u.p) \) that indicates of the number of distinct collector peers whose current route to prefix \( u.p \) includes \( \text{edge} \). The global rank of \( \text{edge} \), \( \text{grank}(\text{edge}) \), is decreased whenever this counter reaches 0, i.e. whenever 0 collector peers are currently using \( \text{edge} \) to reach \( u.p \). On the other hand, whenever \( \text{edge} \) is used again to reach \( u.p \) by any collector peer, \( \text{grank}(\text{edge}) \) is subsequently increased.

```
Algorithm 2 Compute Local Ranks
Require: an update \( u \) is received from \( u.cp \) at \( u.t \)
1: find \( \ell_{u, cp}(u.p, u.t).\pi \) \{previous path for \( u.p \}\n2: if \( \ell_{u, cp}(u.p, u.t).\pi \neq \phi \) then
3: for all edge in \( \ell_{u, cp}(u.p, u.t).\pi \) do
4: \( \text{lrank}(\text{edge}, u.cp) \leftarrow \text{lrank}(\text{edge}, u.cp) - 1 \)
5: OUTPUT \( \text{lrank}(\text{edge}, u.cp), u.t \)
6: end for
7: end if
8: if \( u.\pi \neq \phi \) then
9: for all edge in \( u.\pi(u.p, u.t).\pi \) do
10: \( \text{lrank}(\text{edge}, u.cp) \leftarrow \text{lrank}(\text{edge}, u.cp) + 1 \)
11: OUTPUT \( \text{lrank}(\text{edge}, u.cp), u.t \)
12: end for
13: end if
```

Algorithms 2 and 3 compute \( \text{lrank} \) and \( \text{grank} \) in \( O(nl) \) time and \( O(ep) \) space, where \( n \) is the number of input updates, \( l \) is the maximum length of an AS-path, \( e \) is the number of edges in Internet, and \( p \) is the number of all known prefixes. In fact, implementing \( RIB_e \), \( \lambda \), \( \text{lrank} \) values, and \( \text{grank} \) values with hash tables and supposing \( O(1) \) time for hash table operations, the time complexity of both Algorithms 2 and 3 only depends on the number of input updates and on the number of edges contained in the path \( \ell_{u, cp}(u.p, u.t).\pi \). Observe that no update is considered twice, i.e. these algorithms perform no backward scans when elaborating a stream of updates. Thus, both algorithms requires \( O(nl) \) time. Note that, because of the high connectivity of the Internet, we consider \( l \) as a constant value. About space complexity, Algorithm 3 keeps in memory a running \( RIB_e \) (\( O(p) \)), a counter \( \lambda \) for each distinct edge and prefix (\( O(ep) \)), and the running grank value (\( O(e) \)). A similar argument applies to Algorithm 2, since the number of collector peers is constant.

For the sake of efficiency, we implemented the two algorithms within the same program using shared hash tables. Our experience shows that this optimization can efficiently handle the whole input data stream. Also, note that the elaboration can run in parallel on multiple computers by simply partitioning the prefix space, assigning a subset of prefixes to each machine, and, finally, summing up the partial \( \text{lrank} \)s and \( \text{granks} \) computed by
Algorithm 3 Compute Global Ranks

Require: an update $u$ is received from $u.cp$ a $u.t$
1: find $\ell_{u.cp}(u.p, u.t), \pi \{\text{previous path for } u.p\}$
2: if $\ell_{u.cp}(u.p, u.t), \pi \neq \phi$ then
3: for all edge $\in \ell_{u.cp}(u.p, u.t), \pi$ do
4: $\lambda(edge, u.p) \leftarrow \lambda(edge, u.p) - 1$
   \{u.cp does not use edge any more to reach $u.p$\}
5: if $\lambda(edge, u.p) = 0$ then
6: $grank(edge) \leftarrow grank(edge) - 1$
   \{no cp uses edge to reach $u.p$\}
7: end if
8: OUTPUT $grank(edge), u.t$
9: end for
10: end if
11: if $u.p \neq \phi$ then
12: for all edge $\in \ell_{u.cp}(u.p, u.t), \pi$ do
13: $\lambda(edge, u.p) \leftarrow \lambda(edge, u.p) + 1$
   \{u.cp now uses edge to reach $u.p$\}
14: if $\lambda(edge, u.p) = 1$ then
15: $grank(edge) \leftarrow grank(edge) + 1$
   \{first cp starts using edge to reach $u.p$\}
16: end if
17: OUTPUT $grank(edge), u.t$
18: end for
19: end if

each processes. This setting clearly improves the scalability of our approach.

Table 4 reports the performance of Algorithms 2 and 3, running the single-program implementation over the reference week. Those results show that the computation of local and global ranks has reasonable time and memory requirements, so it can be executed on an average machine.

<table>
<thead>
<tr>
<th>Execution (wall clock) time</th>
<th>13:48:43 (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory used (peak)</td>
<td>2.7 Gbyte</td>
</tr>
<tr>
<td>15' chunk avg processing time</td>
<td>74 sec</td>
</tr>
</tbody>
</table>

Table 4: Algorithms 2 and 3 - Time and memory performance

7.3 Layout for Routing Changes

As shown in Section 4, an effective visualization of a single path change helps understand routing dynamics related to the path change.

Figure 7(b) is the graphical representation of a very simple path change according to our approach. Namely, we draw the path change on an Internet customer-provider hierarchy, where the tier-1 ISPs are the nodes at the top of the graph. We currently assign customer-provider relationships according to the valley-free property ([15]), and relying on the set of well known tier-1 computed in the same way as [20]. However, peer-refering relationships can be inferred using arbitrary algorithms (e.g., [12, 13, 15]) and can be easily fitted into our tool. Moreover, we draw ASes and links in the new path right of the ASes and links in the old path, according to the common plotting convention of time flowing from left-to-right. Note that we cannot use any well known algorithm for visualization of hierarchical structures (e.g. [23]), on account of the specific requirements we discussed above. Thus, we defined an ad-hoc algorithm (Algorithm 4) that, given a path change $c$, returns bidimensional coordinates for the ASes in paths $c.\pi_{old}$ and $c.\pi_{new}$ satisfying the above properties.

First, we assign each AS its layer in the customer-provider hierarchy, defining the vertical coordinates (lines up to 10). In order to compute the horizontal coordinates, all the edges are directed according to their orientations inside either $c.\pi_{old}$ or $c.\pi_{new}$, and all the nodes are categorized as follows:

- **type 1**: nodes in the “uphill” portion of $c.\pi_{old}$
- **type 2**: nodes in the “uphill” portion of $c.\pi_{new}$
- **type 3**: nodes in the “downhill” portion of $c.\pi_{old}$
- **type 4**: nodes in the “downhill” portion of $c.\pi_{new}$

Then, we add extra nodes and edges whenever an edge spans over multiple layers (Procedure 1). Moreover, we add extra edges (Procedure 2) on the same vertical layer, depending on the type of nodes, in order to ensure that nodes in the new path are drawn right of nodes in the old one (see Figure 7(a)). Finally, the topological sorting of this augmented graph provides us with the horizontal coordinates (line 14). Figure 7 exemplifies the main idea of the algorithm.

We now discuss the complexity analysis of Algorithm 4. Procedure 1 iterates over edges, performing constant time operations, and possibly adding extra nodes and extra edges. Note that no more than $|V|$ nodes and $|V| + 1$ edges can be added by Procedure 1, so it takes time $O(|E|)$. Topological sort is known to take $O(|E|)$ time. Procedure 2 iterates over $|V|^2 = O(|E|)$ nodes. Thus, the overall time complexity is $O(|E|)$. Space complexity is also $O(|E|)$, since Algorithm 4 keeps in memory $G, G'$ the type label for each node, $\pi_o$, and $\pi_n$. Notice that $|E|$ is expected to be quite small in our context, given that the average length of a path is 7 ([15]). Unlike identification of table transfers and computation of ranks, visualization is executed on-demand on a specific path change. Thus, measuring the performance of the implementation of Algorithm 4 over the reference week is meaningless as it does not affect the ability of BGPATH to efficiently process the data stream.
Figure 7: (a) A portion of the augmented graph $G'$ after Algorithm 4 has computed both vertical and horizontal coordinates. Dashed edges and nodes represent extra elements added during the execution. Sample nodes are labeled with their type (b) Final result. Dashed edges belong to the old path, solid edges belong to the new path.

Algorithm 4 Draw a Path Change

Require: an input routing change $c$
1: $G(V, E) \leftarrow c.\pi_{\text{old}} \cup c.\pi_{\text{new}}$ \{undirected graph\}
2: $G' \leftarrow G$ \{new graph\}
3: direct all $(a, b) \in E'$ s.t. $a$ is provider of $b$
4: if $\text{top}(c.\pi_{\text{old}}) > \text{top}(c.\pi_{\text{new}})$ then
5: $E' \leftarrow E' \cup \{(\text{top}(c.\pi_{\text{old}}), \text{top}(c.\pi_{\text{new}}))\}$
6: else if $\text{top}(c.\pi_{\text{new}})$ is a provider of $\text{top}(c.\pi_{\text{old}})$ then
7: $E' \leftarrow E' \cup \{(\text{top}(c.\pi_{\text{new}}), \text{top}(c.\pi_{\text{old}}))\}$
8: end if
9: topologically sort $G'$ {The order of node $n$ is $y(n)$}
10: $E' \leftarrow E$ \{drop new edges and remove directions\}
11: direct edges in $E'$ according to $c.\pi_{\text{old}}$ and $c.\pi_{\text{new}}$
12: splitEdgesSpanningMultipleLayers($G', y, \text{type}$)
13: addSameLayerEdges($G', y, \text{type}$)
14: topologically sort $G'$ {The order of node $n$ is $x(n)$}
15: for all $n \in V$ do \{disregard extra edges and nodes\}
16: output $x(n), y(n)$
17: end for

Procedure 1 splitEdgesSpanningMultipleLayers($G, y, \text{type}$)

Require: $G = (V, E)$
1: for all edges $(a, b) \in E$ do
2: if $|y(a) - y(b)| > 1$ then
3: $V \leftarrow V \cup \{u_1, \ldots, u_i, \ldots, u_n\}$, $n = |y(a) - y(b) - 1|$ \{add extra nodes\}
4: $y(u_1) \leftarrow y(a) + 1$
5: $y(u_i) \leftarrow y(u_{i-1}) + 1$
6: $y(u_n) \leftarrow y(b) - 1$
7: $\text{type}(u_i) \leftarrow \text{type}(a)$
8: $E \leftarrow E \cup \{(a, u_1), \ldots, (u_i, u_{i+1}), \ldots, (u_n, b)\}$ \{add extra edges\}
9: end if
10: end for

Procedure 2 addSameLayerEdges($G, y, \text{type}$)

Require: $G = (V, E)$
1: for all $(u, v) \in V \times V$ s.t. $y(u) = y(v), u \neq v$ do
2: if $\text{type}(v) > \text{type}(u)$ then
3: $E \leftarrow E \cup \{(u, v)\}$
4: end if
5: end for

8 Conclusions and Future Work

As the Internet grows far beyond the initial expectations, interdomain routing becomes more and more complex, and so distributed monitors collect larger and larger amounts of data. Thus, (semi)automatic tools for BGP routing data analysis and visualization become increasingly important. Previous works proposed several techniques and tools to extract relevant information from the huge BGP dataset. Namely, they either choose to have a global perspective, looking at major events affecting the whole Internet, or assume the point of view of an ISP which is mainly interested in monitoring its own prefixes without further understanding of the dynamics they undergo.

In this paper, we propose a new approach, which tries to combine the advantages of both these perspectives. In fact, we developed BGPATH, a publicly available system that analyses a user-specified prefix and assesses routing dynamics the prefix undergoes over the global Internet activity. Our tool combines data collected by multiple distributed monitors, checking the reliability of available data sources. Then, it estimates the usage of each interdomain edge, both from a local and global view, taking into account all routed prefixes. Finally, BGPATH graphically displays detailed and aggregated information about path changes experienced by a single user-specified prefix. We depicted the architecture of BGPATH and showed that the system can process in a streaming fashion a huge amount of input data (collected
by all the RIS and RV monitors). Furthermore, we detailed the underlying algorithms, and discussed their time and space complexity.

As future work, we plan to further extend the information computed and visualized by BGP ATH, in order to increase the understanding of the user on routing dynamics and, as a consequence, his ability to pinpoint network events. On the other hand, we are working on defining new approaches for root cause analysis in interdomain routing, to overcome limitations of previous works (e.g., [14, 10, 8]). Our long term goal is to equip the system with pattern recognition modules, in order to automatically infer alleged causes of the input path change. A preliminary study of common interdomain routing patterns is described in [9].

We believe that our work improves the state-of-the-art approaches for BGP data visualization and allows a deeper understanding of interdomain routing dynamics.

References