An Algorithmic Approach to Identifying Link Failures

Mohit Lad, Akash Nanavati
Computer Science Dept.
University of California
Los Angeles, California 90095
Email:{mohit,akash}@cs.ucla.edu

Dan Massey
USC/ISI
Arlington, Virginia 22203
Email: masseyd@isi.edu

Lixia Zhang
Computer Science Dept.
University of California
Los Angeles, California 90095
Email:lixia@cs.ucla.edu

Abstract

Due to the Internet's sheer size, complexity, and various routing policies, it is difficult if not impossible to locate the causes of large volumes of BGP update messages that occur from time to time. To provide dependable global data delivery we need diagnostic tools that can pinpoint the exact connectivity changes. In this paper we describe an algorithm, called MVSChange*, that can pin down the origin of routing changes due to any single link failure or link restoration. Using a simplified model of BGP, called Simple Path Vector Protocol (SPVP), and a graph model of the Internet,* MVSChange *takes as input the SPVP update messages collected from multiple vantage points and accurately locates the link that initiated the routing changes. We provide theoretical proof for the the correctness of the design.*

Fault Tolerant Algorithms, Fault Diagnosis, Routing and Graph Theory

1. Introduction

Diagnosis tools are essential in monitoring global Internet routing infrastructure for dependable packet delivery. The Internet is divided into a large number of Autonomous Systems (AS) and the Border Gateway Protocol (BGP) [15] is used to exchange reachability information between these Autonomous Systems (AS); a new BGP update message is generated whenever a border router's reachability to any destination prefix is changed. Although the BGP specification is relatively simple, understanding the the global routing dynamics has proven to be great challenge. Due to the Internet's sheer size, complexity, and various routing policies, it is difficult if not impossible to identify the causes behind large volumes of BGP update messages that occur from time to time in today's Internet.

Most Autonomous Systems monitor their own local data and some even provide snapshots to the public. In the last few years, a number of passive BGP monitoring/vantage sites, such as RIPE [16] and RouteViews [14], have been established to collect BGP update data from BGP routers residing in multiple ASes. The resulting BGP log data is potentially useful for diagnosis purpose. However without a set of effective tools, the large volume of raw data obscures the actual routing changes. For instance, in April of 2003, the data collection points at RouteViews logged over 10 gigabytes of unprocessed data from over 30 routers in different geographic locations. An ad-hoc combination of intuition, experience, and informal analysis is often used to speculate the causes of large swings in BGP updates. In addition to the challenge of scale, the underlying Internet topology is not known precisely and the monitoring points provide views from only a limited set of geographic locations. In such an environment, ad-hoc techniques are limited by the expertise of the administrator and it is not easy to identify the underlying events that cause BGP changes. With the increased importance of fault tolerance and survivability, the ability to infer the cause of routing changes would immensely help in diagnosing the failures and estimating the impact. To provide dependable global data delivery, we need analysis tools that can help us understand the BGP system and pinpoint the exact cause of connectivity changes.

In this paper we present a formal approach for analyzing routing data to identify the origin of routing changes. Using a simplified model of BGP, called Simple Path Vector Protocol (SPVP), and a graph model of the Internet, we present the *MVSChange* algorithm set. Note that even a single link failure or link addition can result in large number of BGP path changes, but finding this failure without knowledge of the underlying topology can be a challenge. As a first step toward designing a formal set of algorithms for understanding Internet route changes, we focus on identifying the single link failure or addition that caused an observed set of changes. Our approach takes snapshots of routing tables collected from multiple vantage points, at two different time instances, and without knowing the underlying topology, locates the link that initiated the routing changes, as precisely as possible. We provide theoretical proof confirming the correctness of our design. While previous work has been done in BGP event analysis, the lack of formal algorithms that can automatically take route data as input and output results with verifiable properties, has remained an open challenge. This work is a step in the direction of formal algorithms which combine views from multiple vantages points to identify possible causes in the underlying network connectivity.

The paper is organized as follows. Section 2 explains the current BGP data collection methodology and abstracts the practical problem into an analytic model. Section 3 then takes the view from a single monitoring point and identifies *one* possible set of link changes that could explain the routing dynamics observed from that site. Section 4 shows how this possible set of failures is used to prove whether a single link failure or single link addition could have triggered the changes. If a single link change could have triggered the change, we identify *all* possible failure or addition scenarios that could have triggered the observed changes. Section 5 builds on the single viewpoint faults to combine views from different monitoring points to present a global consistent explanation of the routing event. Finally section 6 reviews related work and Section 7 concludes the paper.

2. BGP and Routing Model Description

BGP is a *path vector routing protocol* used to exchange reachability information between Autonomous Systems. BGP routers in neighboring Autonomous Systems exchange routing updates. A BGP updates lists the path of Autonomous Systems, "$AS_1, AS_2, AS_3 \ldots AS_n$", used to reach the destination prefix. Update messages are transmitted reliably using TCP and once the initial routing table has been exchanged, further updates are sent only if a route changes. Thus in principle each update message should convey new route information and should be triggered by some underlying event such as a link failure, link addition, or policy change.

Projects such as University of Oregon's Route Views [14] and RIPE [16] collect BGP updates from a number of Autonomous Systems. The monitoring points set up peering sessions with collaborating routers and passively collect all the updates generated by the routers. To the router being monitored, the monitoring point appears to be simply another BGP peer router. The monitoring point logs update data and does not advertise any paths to other ASs. Figure 1 shows an example where AS7018 and AS 1239 are being monitored by a single monitoring point. All updates received over the peering session between 7018 and the monitoring point are collected and written to a BGP log. The updates reported to the monitoring point provide a complete view of all Internet prefixes reachable from AS 7018[1] and the path information in the updates provides a glimpse into AS 7018's view of the Internet topology. Similarly a path announcements from AS1239 are also logged.

2.1. Formal Routing Model

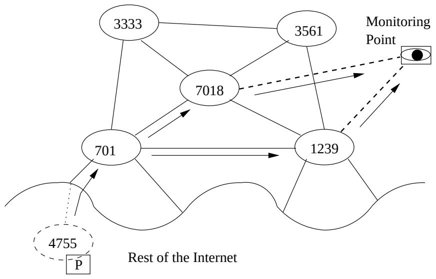
We model the Internet as a simple directed connected graph $G = (V, E)$, where $V = V_D \cup V_N$ and $E = E_D \cup E_N$. V_D represents the set of destinations and roughly corresponds to Internet prefixes. The nodes in V_N are not considered destinations in network G, and roughly correspond to the Internet ASes. Nodes in V_N are connected by links in E_N and each edge in E_N has the form (a, b) where $a, b \in V_N$. The destinations are attached to the nodes in V_N through edges in E_D and each edge in E_D has the form $[d, n]$ where $d \in V_D$ and $n \in V_N$. We assume that each destination d attaches to at least one node in V_N and some destinations may attach to multiple nodes in V_N, i.e. some destinations may be multi-homed.

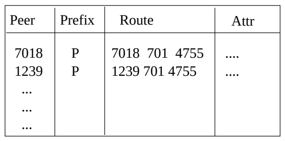
We model the BGP routing protocol as a Simple Path Vector Protocol (SPVP). SPVP is a single path routing protocol, in which each node advertises *only* its best path to neighboring nodes. A path from node v to destination d is a sequence of nodes $Path_v(d) = (v_k v_{k-1} \ldots v_0 d)$ where $v_k = v$, $(v_i, v_{i-1}) \in E_N$ for all $0 \leq i \leq k$, and $(v_0, d) \in E_D$. After receiving and storing a route learned from one of its neighbors, node v selects its best path to destination d according to some routing policy (i.e. ranking function). After the initial path exchanges, further updates are sent *only* if a node's best path changes (i.e. there are no periodic route announcements).

We assume that links can fail and new links may be added to the network. If a link fails, the nodes adjacent to the link detect the failure and all destinations using the link must switch to alternate path (or declare the destination unreachable). Similarly, the addition of a link is detected by the adjacent nodes and as a result, destinations may switch to a new shorter path. The link failures and link additions are not directly reported to any central database or monitoring site. However, by observing the path changes reported by some nodes in V_N, one may be able to estimate the number of changes, locations of the changes, and how the SPVP protocols behaves as a result of the changes.

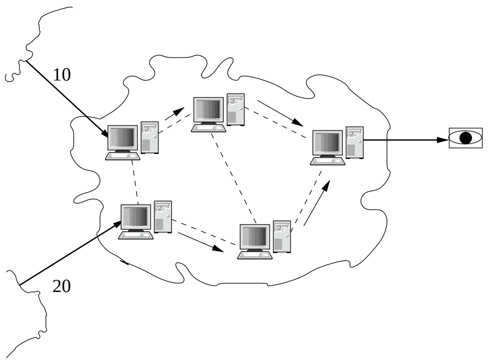
Let $V_M \subset V_N$ denote the set of monitored nodes. At any time, we are able to obtain the routing tables from these monitored nodes, just as in real BGP data. More precisely, for each node $v \in V_M$, we are given the shortest path tree rooted v. Just as the Internet topology is not known, we also

1 Provided that AS 7018 is configured to send full routing tables to the monitoring point.





Peer	Prefix	Route	Attr
7018	P	7018 701 4755	....
1239	P	1239 701 4755	....
...			
...			
...			



assume the topology of graph $G = (V, E)$ is not known by the monitoring process. Given only the shortest path trees from time t_0 and t_1, we would like to explain the cause of any observed routing changes.

2.2. Distinctions Between BGP Monitoring and Our Model

The general problem of understanding BGP behavior based on observed updates is an open challenge. As a first step, we have made a few simplifying assumptions in our formal model.

- We model the edge between two nodes in V_N is a single link E_N and we assume this link is either available or has failed. But in actual BGP operations, the link between two AS can be many physical connections. For example, networks of large ISPs are connected at many places (packets can be exchanged at any of these connection points) and only some of these many physical connections are likely to fail at once.
- We model an AS as a node in V_N and assume each node has one "best" path. In practice, a large AS node is not a single atomic entity and different contiguous portions of the AS, may select and advertise different best paths.
- We assume a shortest path routing policy. The BGP routing protocol allows arbitrary best path selection policies, but some policies can lead to persistent route oscillation [10]. Although our SPVP model can work with any routing policy, this paper considers only a shortest-path policy, which has been proven to converge [9].

Despite these simplifications, the formal model still presents an interesting challenge. The techniques used to find faults in our formal model can be applied to the actual BGP monitoring data, taking the above assumptions into consideration when reviewing the results.

3. Identifying Faults Using a Single Vantage Point

We first consider only the view from a single monitoring point, M, and provide an algorithm that gives *a possible* explanation for the routing changes observed at M. More formally, we are given two shortest path trees rooted at M: $T_0 = (V_0, E_0)$ is the shortest path tree at time t_0 and $T_1 = (V_1, E_1)$ is the shortest path tree at time t_1. Both T_0 and T_1 were computed in some unspecified graph $G = (V, E)$ and if $T_0 \neq T_1$, some link(s) in $G = (V, E)$ must have failed (or recovered). Our objective to identify some scenario of failed (and/or recovered) links that explain the change from T_0 to T_1. Toward this end, we present an algorithm that assigns labels to each edge and identifies *one* possible scenario for the route change event. We later show how the output of Algorithm 1 can be easily extended to identify *every* possible single-event scenario and we show how to use the view from multiple monitoring points to identify, as precisely as possible, the failed (or recovered) link.

Algorithm $FindChange()$ takes tree $T_0 = (V_0, E_0)$ and $T_1 = (V_1, E_1)$ as input and labels each edge in $E_0 \cup E_1$ as either *unchanged*, *vanished*, or *appeared*. A *vanished* link is present in T_0, but not present in T_1. This can occur due the failure of the link or the link may *v*anish as a consequence of some other change. We distinguish the actual failed edges from the other *vanished* edges by labeling failed edges as *failed*. Similarly, edges that recovered or whose additions caused route changes are marked *added*. Note that edges marked *appeared* may not have changed state (from down to up), but simply became part of the shortest path tree as a consequence of some other event.

Conceptually, the algorithm is relatively simple. To identify a link failure, the algorithm combines T_0 and T_1 and then starts by calculating a shortest path tree, T_{fail} in this combined graph. Note that if only a link failure has occurred, $T_{\text{fail}} = T_0$ and the algorithm progressively transforms T_{fail} by removing links that are absent in T_1. The failed links are those whose removal changes T_{fail} and we adjust T_{fail} after each change. Similarly in the case of a link recovery, $T_{\text{add}} = T_1$ is transformed to get T_0, to identify the restored link. We are not given whether a link failure or link recovery (or both) have occurred, but we observe that the two steps can be run in parallel as shown in Algorithm 1.

To maintain T_{fail} and T_{add}, we define a function called $SPT(V, E)$. In our model, the edges are unweighted and this function can be implemented to run in linear time

$O(|E|)$ as a *Breadth First Search* with appropriate tie-breaking. One important property of the shortest path tree is that leaf nodes can only be nodes from V_D (destination set), because in real life the monitoring point gets paths only to *prefixes* and *not to network nodes*. Thus some nodes of V_N (and even from V_D) might disappear over time, hence in general $V_0 \neq V_1$ (recall V_i = node appearing in T_i).

Algorithm 1: FindChange(T_0, T_1)

```
Input: $T_0 = (V_0, E_0)$ : The SPT from $M$ at $t_0$ ;
       $T_1 = (V_1, E_1)$ : The SPT from $M$ at $t_1$;
Output: Marked edges: unchanged, vanished, failed, added;
Let $V = V_0 \cup V_1,\ E_{\mathrm{add}} = E_{\mathrm{fail}} = E = E_0 \cup E_1$;
Let $T_{\mathrm{fail}} = T_{\mathrm{add}} = SPT(V, E)$;
for each $e \in E$ in BFS order do
    if $e \in E_0 \cap E_1$ then
        $e = unchanged$;
    else if $e \in E_0$ then
        /* $e$ has vanished from $M$'s SPT due to either
        the failure of $e$ itself or some other change in
        $G$ */;
        $e = vanished$;
        if $e \in T_{\mathrm{fail}}$ then
            $e = failed$;
            $E_{\mathrm{fail}} = E_{\mathrm{fail}} - e$;
            $T_{\mathrm{fail}} = SPT(V, E_{\mathrm{fail}})$;
    else if $e \in E_1$ then
        /* $e$ has appeared in $M$'s SPT due to either
        the recovery of $e$ itself or some other change
        in $G$ */;
        $e = appeared$;
        if $e \in T_{\mathrm{add}}$ then
            $e = added$;
            $E_{\mathrm{add}} = E_{\mathrm{add}} - e$;
            $T_{\mathrm{add}} = SPT(V, E_{\mathrm{add}})$;
```

3.1. Example

Figure 3 provides an example showing the execution of the algorithm. Figure 3-a, shows T_0, the shortest path tree at time t_0. Three prefixes, $P1$, $P2$, and $P3$ are advertised from AS-4, AS-6 and AS-5 (respectively). Figure 3-b shows T_1, the shortest path tree at time t_1. Note that the path to prefix $P3$ has not changed, but an event has changed the shortest paths from M to prefixes $P1$ and $P2$. The algorithm will label each edge as *unchanged*, *vanished*, or *appeared* and will select one edge as the cause.

The algorithm first combines the two trees T_0 and T_1 to obtain the graph shown in Figure 4-a and computes a new shortest path tree in this combined graph. Following a

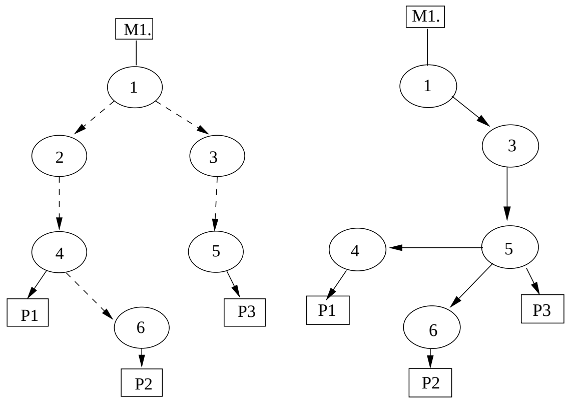


Figure 3. Trees T_0 at time t_0 and T_1 at t_1

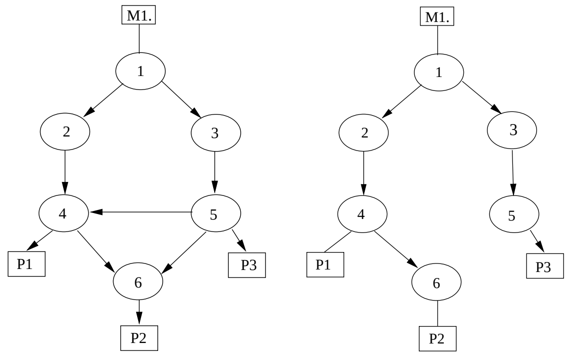
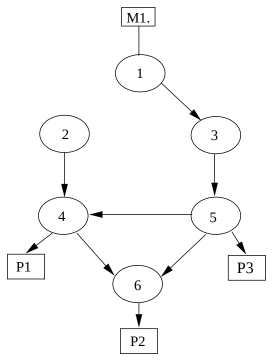


Figure 4. Initial Graph G with $E = E_0 + E_1$ and Initial $T_{\mathrm{fail}} = T_{\mathrm{add}}$ on G

a breadth first search on edges, each edge is tested and labeled accordingly. Edge $(M, 1)$ appears in both graphs and is labeled *unchanged*. Edge $(1, 2)$ appears only in T_0 and is labeled as *vanished*. Furthermore, edge $(1, 2)$ is in the SPT T_{fail} and thus edge $(1, 2)$ is marked as a *failed* link and a new SPT T_{fail} is computed after excluding edge $(1, 2)$, as shown in Figure 5-b. Each of the additional edges are tested in BFS order and are either present in both trees and labeled as *unchanged* or not present in T_{fail} (or T_{add}) and labeled as *vanished* or *appeared*. Note that for edges $(5, 4)$ and $(5, 6)$, since they are present in T_1, their membership has to be checked in T_{add} and not T_{fail}. The resulting labels are shown in Figure 6, where F denotes *failed*, A denotes *appeared*, V denotes *vanished*, and U denotes *unchanged*.

In this example, the algorithm identified link $(1, 2)$ as failed link. All other links did not change state, though they moved into or out of the shortest path tree as a consequence of this single failure. While this scenario is a feasible explanation for the change observed at M, it is not the only possible explanation. The failure of only link $(2, 4)$ could have also caused the change from T_0 to T_1. In addition, a



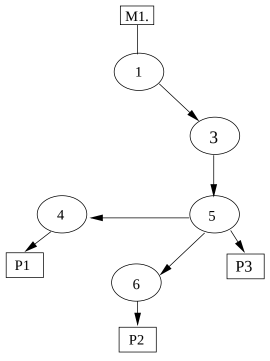


Figure 5. E_{fail} after removal of $(1, 2)$ and resulting T_{fail}

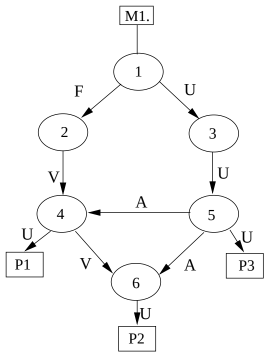


Figure 6. Final labeled graph

variety of multi-failure/recovery events could have caused the change, such as the recovery of link $(4, 5)$ combined with failure of links $(4, 6)$ and link $(2, 4)$. Any one of these events is a plausible explanation and the exact cause cannot be determined from only T_0 and T_1. Since we consider edges in BFS order, our algorithm selects the edge closest to the vantage point as failed. We first show the algorithm always produces some feasible explanation and then show how we can easily extend this feasible solution to identify *all* single-fault explanations.

3.2. Correctness

While no algorithm could produce a unique solution given only this limited information, it is clear the algorithm will always provide an feasible explanation for any input T_0 and T_1.

Let E^{t_0}, E^{t_1} be the set of links available at time t_0, t_1 respectively in the graph. At a monitoring point M, let T_0, T_1 be to Shortest Path Trees rooted at M, i.e. let $T_0(V_0, E_0) = SPT(V, E^{t_0}), T_1(V_1, E_1) = SPT(V, E^{t_1})$. Let $E = E_0 \cup E_1$.

We first claim that the algorithm 1 produces a correct explanation of the change. This can be formalized as follows.

Theorem 1 (Correctness of $FindChange$). *Given T_0, T_1, suppose algorithm FindChange() labels set $F \subseteq E_0$ as failed and $R \subseteq E_1$ as added, i.e. the explanation produced is: $E^{t_0} = E_0 + E_1 - R$ and $E^{t_1} = E^{t_0} - F + R$. Then this explanation is correct, i.e. $T_0 = SPT(V, E_0 + E_1 - R), T_1 = SPT(V, E_0 + E_1 - F)$.*

We will need following simple claim in this proof.

Claim 1.1. *T_{add} does not use any edge labeled* appeared *(it can use edges of $E_1 - E_0$ that are not labeled at the time T_{add} is constructed).*

Proof. We shall show this by induction on the iterations of the `for loop`. Initially, before the `for loop` begins, the statement is true since no edges are labeled. Suppose the statement is true upto $(i-1)$th iteration and now we are considering the ith iteration. The only way T_{add} changes is that we label edge $e = (u, v)$ as *added* and delete it from E_{add}. Suppose for contradiction that edge $f = (x, y)$ labeled *appeared* now enters T_{add}.

Since the `for loop` considers the edges in the BFS order over $E_0 + E_1$, and f was considered before e, we know that x cannot be in the subtree rooted at v (in the tree T_{add}). In a shortest path tree, deleting an edge can only change paths to nodes in the subtree rooted at that edge. Thus if y is not in the subtree rooted at v, path to y does not change and hence $f = (x, y)$ cannot be added. Let us consider then the case when y is there in the subtree rooted at v. If $y = v$, then both f, e are in E_1, so y has two parents $\{x, u\}$ and that is not possible in a tree, therefore $y \neq v$. Hence $d(M, y) > d(M, v)$, and because f was considered before e, $d(M, u) \geq d(M, x)$, i.e. $d(M, v) = d(M, u) + 1 \geq d(M, x) + 1$, i.e. $d(M, y) > d(M, x) + 1$. But in shortest path calculation, for any edge (x, y), $d(M, y) \leq d(M, x) + 1$. Hence this is a contradiction.

Thus T_{add} will not use any edges labeled *appeared* after the recomputation in the ith iteration of the `for loop`, and now the proof follows. [end claim] □

Proof of Theorem 1. We will prove that $T_0 = SPT(V, E_0 + E_1 - R)$, the proof for T_1 is similar.

In the algorithm 1, we label all edges in $E_1 - E_0$ (the last `else if` block) as either *appeared* or *added*. We initialized $T_{\text{add}} = SPT(E_0 + E_1)$. Since we keep deleting edges that are labeled *added*, at the end of the algorithm, $T_{\text{add}} = SPT(E_0 + E_1 - R)$. Thus it is sufficient to prove that at the end of the algorithm, $T_{\text{add}} = T_0$. But if two trees have the same sets of edges, the arrangement of edges is also same. Hence we will only show that the set of edges used in T_{add} is exactly E_0.

To see this, note that all the edges of $E_1 - E_0$ are labeled *appeared* or *added* by the end of the algorithm. The edges labeled *added* (R) have already been deleted from E_{fail} and so they cannot be in T_{add}. And by claim 1.1, edges labeled *appeared* are not used in T_{add} at any time. Hence at the end of algorithm, T_{add} can use only edges from E_0, in fact it will use all the edges of E_0 as they are never deleted from E_{fail}. □

4. Dealing with Single-Fault Scenarios

Although there may be many possible explanations for the routing changes observed by M, we first seek the simplest possible explanation. In the best case, a large number of route changes can be caused by a single link failure or recovery. In the remainder of the paper, we call the failure or recovery of a single link as a *single-event explanation*. We have shown that algorithm $FindChange()$ finds one explanation. In this section, we show that the algorithm in fact always finds a single-event explanation, if such a explanation exists. We then show how, given this one single-event explanation, we can easily list *all* possible single-event explanations.

More formally, given two shortest path trees $T_0 = (V_0, E_0), T_1 = (V_1, E_1)$, we define an explanation for the change in trees as two sets of edges (F, R) where F is the set of failed edges and R is the set of recovered edges. In other words, $T_0 = SPT(V, E^{t_0})$ and $T_1 = SPT(V, E^{t_1})$ where $E^{t_1} = E^{t_0} - F + R$.

Theorem 2 (Solution-Space Structure Theorem). *Suppose $E^{t_1} = E^{t_0} - e$ (i.e. a single link failure). Then any other single event explanation of $T_0 \longrightarrow T_1$ change must have $R = \emptyset$ (i.e. any other single event explanation is also a link failure).*

Proof. There is a change from T_0 to T_1 only if $e \in E_0 - E_1$. We already given one explanation: $E^{t_0} = E_0 + E_1, E^{t_1} = E^{t_0} - e$, i.e. $T_0 = SPT(V, E_0 + E_1), T_1 = SPT(V, E_0 + E_1 - e)$.

If some other single event explanation has $R \neq \emptyset$ then $F = \emptyset$ and $|R| = 1$, say $R = \{f\}$, for some $f \in E_1 - E_0$. Thus $T_0 = SPT(V, E^{t_0})$ and $T_1 = SPT(V, E^{t_0} + f)$ for suitably chosen E^{t_0}. Thus $E_0 \subseteq E^{t_0}$, and $E_1 \subseteq E^{t_0} + f$, so $E_0 + E_1 \subseteq E^{t_0} + f$. Now Shortest Path Tree over $E^{t_0} + f$ uses only edges from $E_0 + E_1$, so $SPT(V, E^{t_0} + f) = SPT(V, E_0 + E_1)$. In this equation, the latter quantity by our pre-ordained explanation is T_0, while the former is explained as T_1. Hence we arrive at the contradiction, $T_0 = T_1$. Therefore we must have $R = \emptyset$. □

Now we show that in the single link failure case, algorithm $FindChange()$ in fact outputs an explanation with $|F| = 1$ and $R = \emptyset$. It is possible that F may include an edge other than the actual failed link, but it is important to note that $|F| = 1$.

Theorem 3 (Optimality of $FindChange$). *Suppose $E^{t_1} = E^{t_0} - e$ (single link failure). Then $FindChange()$ labels exactly one edge as* failed, *and no edge as* added.

Proof. From theorem 2 about admissible explanations and the correctness proof in theorem 1, algorithm $FindChange()$ will never mark any edge as *added*. Thus it remains now to show only the first part of the theorem, i.e. $FindChange()$ labels exactly one edge as *failed*.

Let $e = (u, v)$ be the actual failed edge. Then $Path^{T_0}(v)$ changes. On this path, let x be the node closest to v such that $Path^{T_0}(x)$ does not change and let (x, y) be the edge on $Path^{T_0}(x \longrightarrow v)$. We start with $T_{\text{fail}} = T_0$ and we consider edges in the BFS order of $E_0 + E_1$ (Shortest Path Tree = Breadth First Search Tree for unweighted graphs). Thus we find (x, y) to be the first edge such that $(x, y) \in E_0 - E_1, (x, y) \in T_{\text{fail}}$ and we mark it *failed*. By deleting (x, y) we will change paths of only the nodes in the subtree rooted at y (that includes nodes u, v). When we recompute $T_{\text{fail}} = SPT(E_0 + E_1 - (x, y))$, call it $\tilde{T}$, we claim that this tree uses edges from E_1 only so that all other edges of $E_0 - E_1$ will be marked *vanished* and not *failed*.

If $x = u, y = v$, we have identified the correct fault and we are done. Otherwise the only reason $Path^{T_0}(i)$ changes for $i \in Path^{T_0}(y \longrightarrow u)$ because of (u, v) failure is that there is a destination node $p \in V_D$ such that $Path^{T_0}(p)$ is along $Path^{T_0}(v)$ and there is no other destination $q \in V_D$ such that $Path^{T_0}(q)$ uses node i but not edge (u, v). Hence in T_1, all these nodes must be absent. This is true also when we delete only (x, y) and so in $\tilde{T}$, all these nodes (and hence the edges) are absent. In other words, deleting (x, y) has the same effect as deleting (u, v), i.e.

$$SPT(V, E_0 + E_1 - (x, y)) = SPT(V, E_0 + E_1 - e) = T_1.$$

Now since new T_{fail} is same as T_1, it does not have any edges from $E_0 - E_1$, so it will not mark anymore edges as *failed*, i.e. the algorithm marks exactly one edge as *failed*. □

In the case of single link recovery, results analogous to theorems 2 and 3 can be proved along the same line of reasoning. Combining this with theorem 1, we have proved correctness and optimality of algorithm $FindChange()$ in the case of single event explanations.

Thus in order to determine if a single-fault event could explain the changes at monitoring point M, we simply run algorithm $FindChange()$ and count the number of links labeled *failed* or *added*. If there is more than one link in this set, the change could not have been caused by any single-fault event. If there is exactly one such link, then we have a single-fault explanation. Furthermore, if the link is *failed*, then all single-event explanations involve only one failed link. Similarly, if the link is *added*, then all single-event explanations involve only one recovered link.

4.1. Identifying All Possible Single-Faults

Given a single plausible failure (recovery) identified using $FindChange()$, we now show how to easily identify *all* possible single faults. Recall the example in Figures 3–6 identified link $(1,2)$ as the only failed link. Link $(2,4)$ could also be selected as the only failed link, but the failure of only link $(4,6)$ would not explain the change in route to prefix $P1$. As the previous theorem shows, links $(5,4)$ and $(5,6)$ are marked *appeared*, but their recovery alone could not explain the changes in the route to either $P1$ or $P2$. Thus the only possible single fault events in the example are the failure of link $(1,2)$ or the failure of link $(2,4)$.

At the conclusion of $FindChange()$, every edge is labeled as either *unchanged*, *vanished*, or *appeared* and one edge is labeled either *failed* or *added*. $FindPath()$ starts with this edge and moves down the tree until all edges that could have failed are marked *failed* (or in the case of single recovery, all edges that could have recovered are marked *added*). $FindPath()$ returns the entire set of links that constitute this path. The failure of any of these links alone would explain the path changes from T_0 to T_1.

Algorithm 2: FindPath(T_0, T_1)

```
Input: $T_0(E_0), T_1(E_1)$: $SPT$s from a single View
Output: $P$: A Path of candidate edges for failure
Let $e = (u,x)$ be the only edge marked failed in $FindChange(T_0,T_1)$;
Initialize $P = \{e\}$;
Let $p = x$;
if tag$(e)$ == vanished then
    while p has exactly one outgoing edge $(p,q) \in E_0 - E_1$ do
        Add $(p,q)$ to $P$;
        Set tag$(p,q)$ = failed;
        Set $p = q$ ;
else if tag$(e)$ == appeared then
    while p has exactly one outgoing edge $(p,q) \in E_1 - E_0$ do
        Add $(p,q)$ to $P$;
        Set tag$(p,q)$ = added;
        Set $p = q$ ;
return $P$;
```

5. Combining Views from Multiple Monitoring Points

Having dealt with route change explanations from single view point, we now focus on the problem of combining the views from multiple vantage points to better identity the single fault scenario. The main objective is to derive a globally accepted link or set of links, each of whose failure (addition) alone, could explain the change in routes between times t_0 to t_1. Even if all monitoring points offer a single fault explanation, some conflict resolution problems remain. In the previous steps, a monitoring point assigned a label to each edge visible from that monitoring point and different monitoring points may have assigned conflicting labels to an edge. Combining views primarily requires some form reconciling these differences.

Note that it is possible to have a single fault explanation from one monitoring point, yet have only multiple fault explanations from the combined view. In this case, the combined view perspective proves a single fault could not explain the changes observed globally. For example, it may be the case that monitoring point M_1 identified link (a,b) the only failed link, but monitoring point M_2 identified link (x,y) as the only failed link. This can easily occur since link (a,b) may have never appeared in the view from M_2 and similarly (x,y) never appeared in the view from M_1.

Algorithm $MVSFault()$ details the procedure for finding a globally consistent single fault (or single addition) explanation. The first part of the algorithm involves finding the set of links common to possible failed (added) links from all the vantage points in the case of a single failure. The second part involves updating the edge labels of the algorithm and uses a procedure *updateLabel()* to achieve this. The algorithm returns the empty set if a single fault explanation is not possible in the combined view.

Algorithm 3: MVSFault($\mathcal{M}$)

```
Input: Set of SPTs at time $t_0$ and $t_1$ from vantage
       points in $\mathcal{M}$
Output: $Causes$: A set of single-change failed (re-
        covered) links
Let $Causes = \emptyset$;
for $i = 1, \ldots, M$ do
    $P$ =FindPath($SPT^{t_0}_{\mathcal{M}_i}, SPT^{t_1}_{\mathcal{M}_i}$);
    /* a few $\mathcal{M}_i$s may not be affect by this failure,
    ignore them */
    if $P \neq \emptyset$ then
        if $Causes == \emptyset$ then
            Initialize $Causes = P$;
        else
            $Causes = Causes \cap P$;
    for each edge $e \in M_i$ do
        if conflict$(e, G)$ then
            updateLabel$(G, e)$;
return $Causes$;
```

If at every step, $P == \emptyset$, then no monitoring point detected any change in routes and the failed (added) edge set is empty. If many monitoring points detected changes, then the actual failed link is present in paths P from each of the monitoring points who observed a change. Thus taking the intersection of these paths will highlight one or more links. The intersection cannot be empty if only a single fault explanation exists. The algorithm outputs a set of links denoted as $Causes$ and each link in this set is a possible single failure explanation. Note that even with the view from multiple monitoring points, our algorithm may still identity several possible links whose failure could have caused the change. In this case, the actual failed link belongs to the set $Causes$, but given the limited views, we cannot pin-point the exact failure. The other task achieved by MVSCauses() is to re-label the graph edges to achieve a more accurate label for each edge as views from different vantage points are added.

5.1. Re-labeling of Edges

When combining views, edge labels assigned for an individual vantage point, may not be valid when we add another view. For example, an edge e labeled as *vanished* from one point M_1, could be labeled *unchanged* from another point. We address this by maintaining a global labeled graph, that contains edge labels based on already resolved views from vantage points. For every view added to this global labeled graph, we relabel the conflicting edges. The following rules follow from our labeling scheme, when adding a view to the global graph.

1. An *unchanged* edge would always remain unchanged.
2. Any *failed* or *vanished* edge creating a conflict (with an *unchanged*, *appeared*, or *added* edge) is re-labeled *unchanged*. Note that from a point M, an edge can be labeled *vanished*, only when it is present in T_0, and not present in T_1. This edge can cause a conflict only when it is present in T_1, viewed from another point, and thus this edge can only be labeled *unchanged*. A *failed* edge can never be relabeled *added* or *appeared*
3. An *added* or *appeared* edge creating a conflict (with an *unchanged*, *vanished*, or *failed*) has to be relabeled *unchanged*. An *added* link can never be relabeled *failed* or *vanished*

Figure 7 shows the possible state transitions while combining views. Procedure *UpdateLabel()* works on the rules in this state transition diagram. The states in this diagram are the current labels associated with the global graph. The labels on the edges are the labels associated with the view from $\mathcal{M}_i$ that is being added. The transition from *vanished* to *failed* can only occur in *FindPath()*.

Figure 8 shows how the entire process of combining views and edge relabeling works with an example. Figure 8-a shows the view from M_1, based on the example in Figure 4. For clarity, the SPTs at times t_0 and t_1 are indicated by dashed and solid lines respectively. Similarly, the view from M_2 is presented in 8-b. Executing *FindPath()* on these two graphs generates Figures 8-c and 8-d. *FindPath()* also produces a path containing the possible single link failure candidates, shown in figures 8-f and 8-g for M_1 and M_2 respectively. The intersection of these two paths (as calculated by $MVSFault$) results in the failed link as

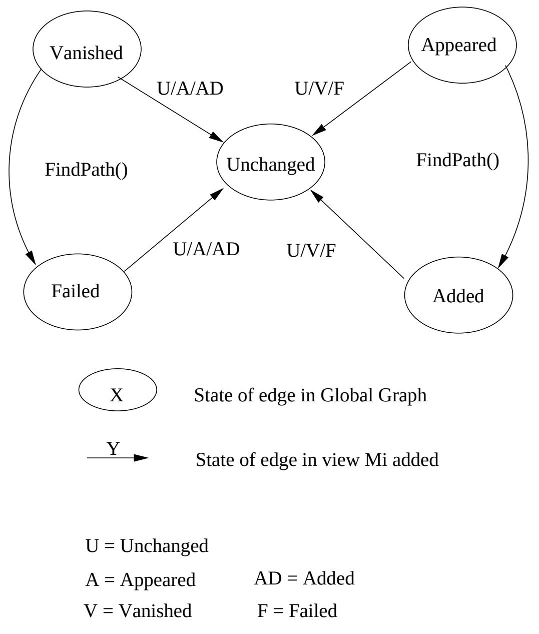
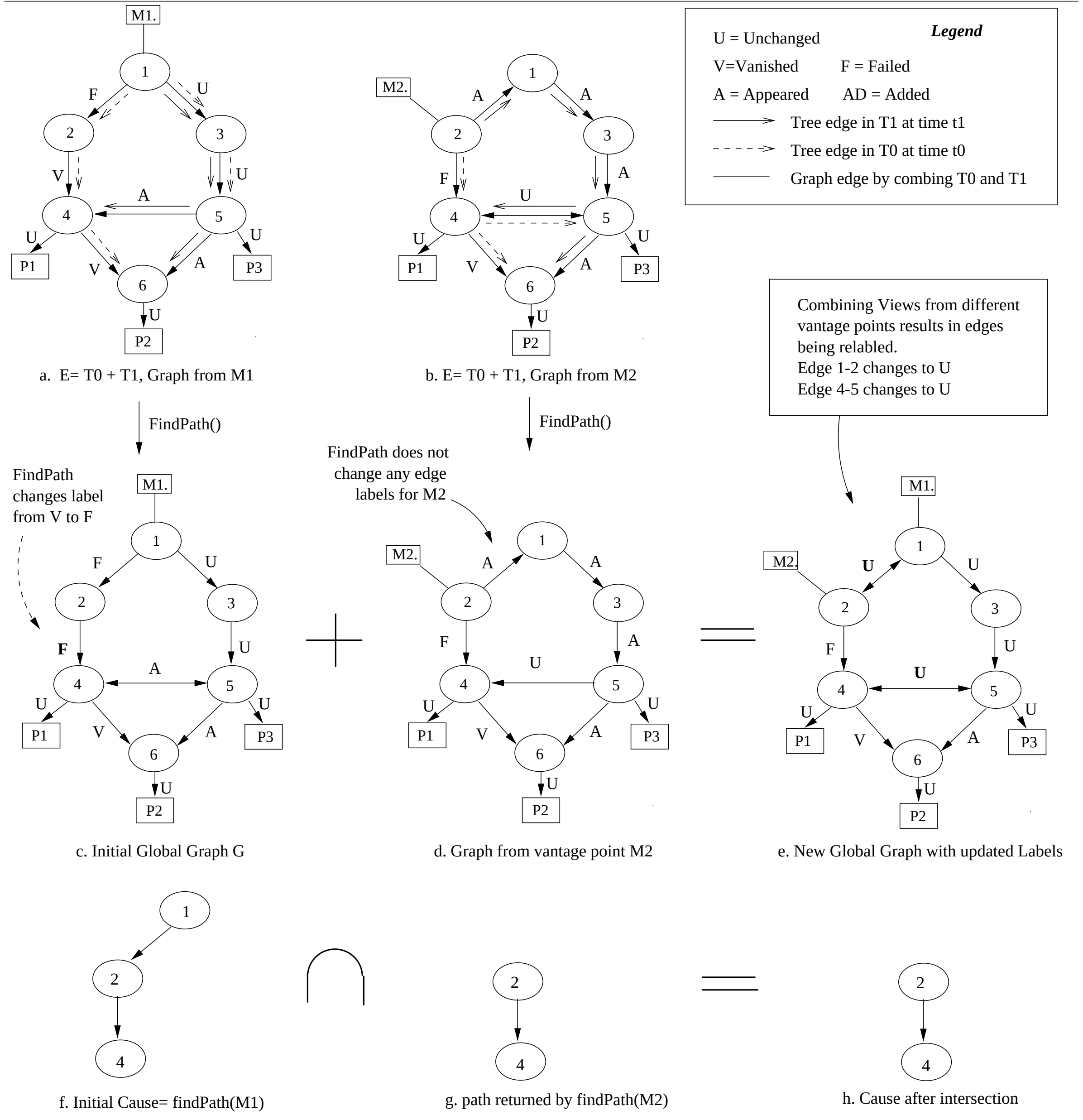


Figure 7. State Transition Diagram for labeled edges

$(2, 4)$. The graph for M_1 initially forms the global graph, and the edge relabeling is done by updating the labels in the global graph from 8-c to 8-e. Note how the labels for $(1, 3)$ and $(3, 5)$ from M_2, did not have any impact on the corresponding labels of global graph which were already *unchanged*. However, $(1, 2)$ and $(4, 5)$ in the global graph were relabled *unchanged*. Thus, we see how we can combine views to identify the single-change links that could explain the route changes.

6. Related Work

This paper presents a first step towards an analytical system for detecting faults at the BGP level. Previous work has studied the impact of various stress events on BGP routing, including studies of Code-Red [1] and SQL Slammer [3], as well as infrastructure failures. While some of the previous work was able to provide explanations for these events, the explanations relied on human insight along with extensive data processing. Code-Red worm and its impact on BGP routes was first presented in [2], who reported abnormalities in BGP behavior during the worm attack. [19] reported the presence of monitoring artifacts in the code-red data and provided a second look at the event. According to the study in [19], the worm had a large impact on some edge networks, and weaknesses in BGP's design and implementation substantially amplified the impact. The impact of SQL Slammer worm on BGP routing was also studied in [12] and efforts for visualizing BGP updates to get a sense of the location of change has been done in [11]. The approach in [11] is geared towards summarizing large scale



M1.
M2.
U = Unchanged
Legend
V=Vanished
F = Failed
A = Appeared
AD = Added
Tree edge in T1 at time t1
Tree edge in T0 at time t0
Graph edge by combing T0 and T1
a. E= T0 + T1, Graph from M1
b. E= T0 + T1, Graph from M2
FindPath()
FindPath changes label from V to F
FindPath does not change any edge labels for M2
Combining Views from different vantage points results in edges being relabled.
Edge 1-2 changes to U
Edge 4-5 changes to U
c. Initial Global Graph G
d. Graph from vantage point M2
e. New Global Graph with updated Labels
f. Initial Cause= findPath(M1)
g. path returned by findPath(M2)
h. Cause after intersection
P1
P2
P3

Figure 8. Example of Combining Views from Two Vantage Points

changes to understand events like major ISP failures. Work in [18] is directed at visually capturing the BGP routing information, and contains guidelines for gaining insights. However, formal algorithms to address the BGP dynamics are lacking in previous studies.

Other studies have used BGP route data to statically infer the Internet's underlying AS level topology. [7], [8] and [4] used AS path information from BGP routing tables to map and derive characteristics of the Internet interdomain topology. [5] further examined the relationships between ASes using AS path information and classified AS relationships into three classes: customer-provider, peering and sibling, based on the observation that each AS path contains first a sequence of customer-provider and sibling edges, then one or zero peer-peer edge, and finally a sequence of provider-customer and sibling edges. Subramanian et. al. [17] also inferred inter-AS relationship, but their approach is based on the consistency among multiple routing tables. These studies have led to a better understanding of the structure of the Internet topology and the commercial relationship between ASes. However, each uses snapshots of the Internet and tells us little about dynamics. The results cannot be trivially applied to pin point failures.

Algorithm 1 FindChange can also be viewed as determining *tree-edit distance* when the operations allowed are delete and insert and trees are restricted to be Shortest Path Trees. But in general finding how to convert one tree into the other is NP-hard and many have considered approximate solutions [6]. While it is enough for us to compute SPT everytime a change occurs, many efficient algorithms

exist that dynamically update Shortest Paths Trees, see for example [13]. Their average case complexity is $O(\log n)$ for each update, and can be used for graphs involving weighted edges, to reduce the complexity of SPT.

7. Conclusions and Future Work

In order to build diagnostic tools for the global routing infrastructure, this paper developed an algorithmic set, called *MVSChange*, to identify the origin of routing changes caused by a single link. Utilizing the information carried in a path vector routing protocol and collected from multiple vantage points, *MVSChange* builds graphs of snapshots of the network routing connectivity over time, and applies basic graph theory approaches to pin down the exact location of the link change. Along with the theoretical proof, our preliminary simulation experiments have also confirmed the correctness of the design.

We believe that *MVSChange* represents an important first step towards applying formal methods to inter-domain routing diagnosis. Our plan for future work includes extending *MVSChange* to cover increasingly more complex failure cases, such as node failure and multiple simultaneous failures. We also plan to incorporate considerations of routing policies in order to make *MVSChange* more applicable to Internet deployment.

References

[1] C. A. CA-2001-19. ”Code Red” Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html.

[2] J. Cowie, A. Ogielski, B. J. Premore, and Y. Yuan. Global routing instabilities triggered by Code Red II and Nimda worm attacks. Technical report, Renesys Corporation, Dec 2001.

[3] D. M. et. al. The spread of the Sapphire/Slammer worm. http://www.cs.berkeley.edu/ nweaver/sapphire/.

[4] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the ACM SIGCOMM ’99*, August/September 1999.

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

[6] M. Garofalakis and A. Kumar. Correlating xml data streams using tree-edit distance embeddings. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 143–154. ACM Press, 2003.

[7] R. Govindan and A. Reddy. An analysis of inter-domain topology and routing stability. In *Proceedings of the IEEE INFOCOM ’97*, Apr. 1997.

[8] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM 2000*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE.

[9] T. Griffin and G. T. Wilfong. A safe path vector protocol. In *Proceedings of INFOCOM*, pages 490–499, 2000.

[10] T. G. Griffin and G. T. Wilfong. An analysis of BGP convergence properties. In *Proceedings of SIGCOMM*, pages 277–288, Cambridge, MA, August 1999.

[11] M. Lad, D. Massey, and L. Zhang. Link-Rank: A Graphical Tool for capturing BGP Routing Dynamics. In *UCLA CSD Technical Report*, Aug. 2003.

[12] M. Lad, B. Zhang, X. Zhao, D. Massey, and L. Zhang. Analysis of BGP Update Surge during Slammer attack. In *Proceedings of 5th International Workshop on Distributed Computing*, Dec. 2003.

[13] P. Narvaez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic SPT algorithm based on a ball-and-string model. In *INFOCOM (2)*, pages 973–981, 1999.

[14] U. of Oregon. The Route Views Project. http://www.routeviews.org.

[15] Y. Rekhter and T. Li. A border gateway protocol (BGP-4). *Request for Comment (RFC): 1771*, Mar. 1995.

[16] RIPE. Routing Information Service Project. http://www.ripe.net/ripencc/pub-services/np/ris-index.html.

[17] L. Subramanian, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proceedings of the IEEE INFOCOM ’02*, New York, NY, June 2002.

[18] S. T. Teoh, K.-L. Ma, and S. F. Wu. A Visual Exploration Process for the Analysis of Internet Routing Data. In *Proc. IEEE Visualization*, 2003.

[19] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, and L. Zhang. Observation and analysis of BGP behavior under stress. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, Nov. 2002.