Improvisation of Netplumber on Ryu Controller

Adarsh Srinivasa, Shashank Ravindranath, Yuan-An Liu

ABSTRACT— ***Networks in the modern era are getting more bigger and debugging them is getting more complicated. It is getting harder to observe and analyze the forwarding state of packets, understand the overall system behavior and find the network wide invariants.To handle this, there is a need for a tool that can check the network wide invariants in real time.***

Our paper improvises a real time policy checking tool called NetPlumber which is based on Header Space Analysis. Netplumber can incrementally check for compliance of state changes, using a novel set of conceptual tools that maintain a dependency graph between rules. We are improvising the Netplumber by porting the previously used mininet controller to Ryu controller to reap the benefits of Openflow 1.3. Also, NetPlumber cannot handle the exempt the violation, which currently needs to be done at the Router level. We have added this feature as an enhancement to the current architecture.

I. PROBLEM STATEMENT

Handling the network was all simple which integrally involved indexing into a forwarding table using a destination address and decide where to send the packet next until the network grew multiple folds and got more complex and error prone. Managing and troubleshooting the network was a tedious and manual process which requires manually logging into every box in the network and understanding every protocol and then fixing the ambiguities. Existing tools that check network configuration files and the data-plane state operate offline. These operate at timescale of seconds to hours and cannot prevent bugs in real-time. Implementing a tool that will check network-wide invariants in real time and to achieve the extremely low latency during the checks which will not degrade the network performance was a necessity.

Hence a tool 'Netplumber'[1] was built which is simple, protocol independent and has forwarding functionality of packets that can be used as foundation for systematic verification of networks. Our goal is to improvise the tool to make it more efficient and effective. We are improvising the Netplumber by porting the previously used mininet[7] controller to Ryu[6] controller to reap the benefits of Openflow 1.3 which has a lot of advantages such as expanded IPv6 support, supports per-flow meters which can limit the packets sent to the controller, provides Backbone Bridging tagging, supports encapsulation and de-capsulation of packets.

Because NetPlumber is a real-time policy checking tool which based on the header space analysis framework. NetPlumber incrementally checks for the compliance of state changes using a novel conceptual representation, called the Rule Dependency Graph, that maintain a dependency graph between rules. Since it provides the convenience of adding and deleting rules, we want to provide network administrators a new additional way to determine editing the rules by providing reliable statistical data. With the data, the higher value of the rule indicates that it would have more impacts to the entire network. This would eliminate the manual tedious process of making the changes at the router level.

II. PRIOR WORK

Netplumber Architechture:

NetPlumber has a much faster update time than Hassel because instead of recomputing all of the transformations each time the network changes, it incrementally updates only the portions of the results affected by the change. Underneath, NetPlumber still uses HSA.

Figure 2.1 shows NetPlumber checking policies in an SDN. An agent sits between the control plane and switches and sends every state update (installation or removal of rules, link up or down events) to NetPlumber. Internally, NetPlumber creates and maintains a model of the network, which is used to verify policies in real time. In response to network state changes, NetPlumber updates its network model and reevaluates the policy checks affected by the update. If a violation occurs, NetPlumber performs a user-defined action such as removing the violating rule or notifying the administrator.

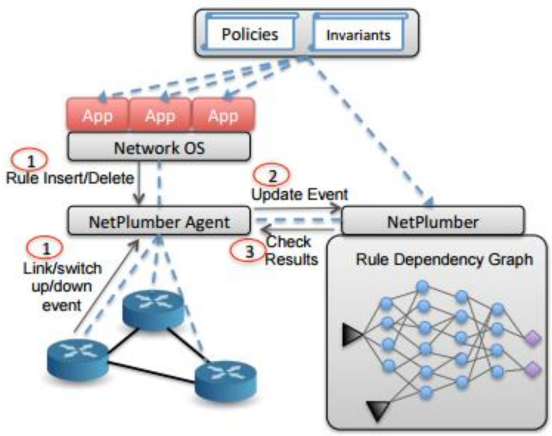


Fig 2.1 .Deploying NetPlumber as a policy checker in SDNs

The Rule Dependency Graph:
NetPlumber creates and maintains a network model in the form of a forwarding graph called the rule dependency graph. The nodes of this graph are the forwarding rules, and directed edges represent the next-hop dependency of these rules. We call these directed edges pipes because they represent possible paths for flows. A pipe from rule a to b has a pipe filter that is the intersection of the range of a and the domain of b. When a flow passes through a pipe, it is filtered by the pipe filter. Conceptually, the pipe filter represents all packet headers at the output of rule a that can be processed by b.

A rule node corresponds to a rule in a forwarding table in a switch. Forwarding rules have priority; when a packet arrives to the switch, it is processed by the highest priority matching rule. Similarly, the NetPlumber needs to consider rule priorities when deciding which rule node will process a flow. For computational efficiency, each rule node keeps track of higher priority rules in the same table. It calculates the domain of each higher priority rule, subtracting it from its own domain.

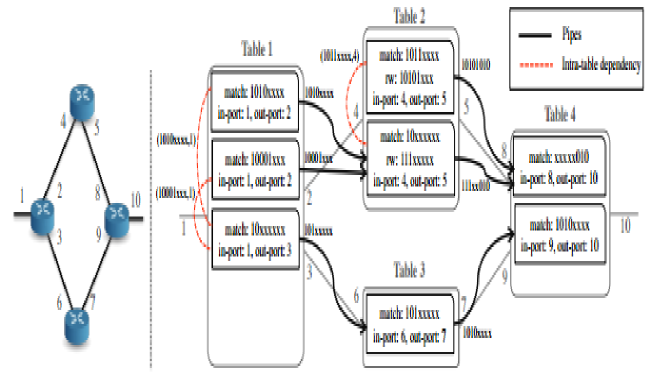


Figure 2.2: Rule dependency graph of a simple network

Source and Sink Nodes:
NetPlumber converts policy and invariants to equivalent reachability assertions. To compute reachability, it inserts flow from the source port into the rule dependency graph and propagates it toward the destination. This is done using a “flow generator” or source node. Just like rule nodes, a source node is connected to the rule dependency graph using directed edges (pipes), but instead of processing and forwarding flows, it generates flow.

Sink nodes are the dual of source nodes. A sink node absorbs flows from the network. Equivalently, a sink node generates “sink flow,” which traverses the rule dependency graph in the reverse direction

Probe Nodes:
Probe node, is used to check policy or invariants. Probe nodes can be attached to appropriate locations of the rule dependency graph and can be used to check the path and header of the received flows for violations of expected behaviour.

There are two types of probe nodes—source probe nodes and sink probe nodes. The former check constraints on flows generated by source nodes, and the latter check flows generated by sink nodes. We refer to both as probe nodes.

Implementation:

Figure 2.3 shows a simple block diagram of the implementation of NetPlumber and its dependency on Hassel. The two systems share the foundation layer, that is, the wildcard expression and header space objects, but they create different models of the network and policies. Hassel uses transfer function while NetPlumber uses rule, probe, and source nodes and flowexps language. Also, the two systems have different ways of checking policies and invariants: Hassel provides the basic functionality for checking reachability. Custom policy checks can be implemented by invoking the basic reachability function and writing extra code to check the specific policy on the result of the reachability check. On the other hand, NetPlumber uses the rule dependency graph and the flow routing techniques to run all of the checks

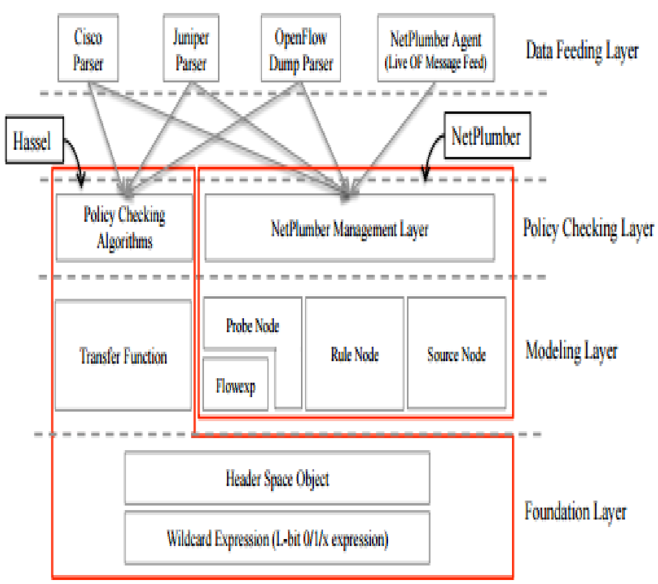


Fig. 2.3 NetPlumber Software Block Diagram

The NetPlumber management layer is the object that manages and controls different nodes and the rule dependency graph. It provides the following API for updating the NetPlumber state and checking policies:

Add Table: Adds a new table to NetPlumber with a list of input/output ports belonging to the table

Remove Table: Removes a table with a given table ID.

*Add New Rule :*Adds a rule with the specified matching wildcard expression, input ports, action, and output port and returns an ID for the created table.

Delete Rule: Deletes a rule with the given ID.

Add Link: Adds a unidirectional link from a source port to a destination port.

Remove Link: Removes a unidirectional link from a source port to a destination port.

Add Probe Node: Attaches a probe node to a particular port in the rule dependency graph.

Remove Probe Node :Removes a probe node with a given ID.

Add Source/Sink Node: Attaches a source/sink node to a given port in the rule dependency graph.

Remove Source/Sink Node: Removes a source/sink node with a given ID.

Set Loop Detection Callback Function: Sets a global callback function for the loop detection check.

Set Black Hole Detection Callback Function: Sets a global callback function for black holes detection.

Limitations:

Implementing Netplumber on the default mininet controller might not be as advantageous as implementing over the external controller such as Ryu for the factors listed above such as support of Openflow 1.3 which inturn has a lot of advantages such as expanded IPv6 support, supports per-flow meters which can limit the packets sent to the controller, provides Backbone Bridging tagging, supports encapsulation and de-capsulation of packets. Also, NetPlumber cannot handle the exempt the violation, which currently needs to be done at the Router level. We want to provide network administrators a new additional way to determine editing the rules by providing reliable statistical data. With the data, the higher value of the rule indicates that it would have more impacts to the entire network. This would eliminate the manual tedious process of making the changes at the router level.

III. Research approach

Section 1: Research Approach - Improvising

We have performed lot of research to understand the NetPlumber [12] and Header Space analysis [16] in depth. We read the paper that are available in the internet, the reference is [9], [10], [11], [13], [14], [17] and [18]. We have also gone through the base source code available in [15] and try to understand the code by considering each of the module and we went through the thesis paper of the original author [19]. We have debugged almost all the modules and tried to understand the link between each of the modules.

Each of the modules are written in different programming language for different purpose and are linked together to run as a NetPlumber.
The original source code has five different modules as Hassel-c, has-python, mahak, mininet and net_plumber.

Each module has its own functionality and interconnectivity lets see how each of them are connected and what are the functionalities of each module in brief.

Hassel-c - is an optimized version of the header space library written in C. It is a separate library that can be plugged into the NetPlumber. This Hassel-c itself can be used to statically check network specifications and configurations to identify important class of failures such as Reachability Failures, Forwarding loops, Traffic Isolation and Leakage problem.

HSA-python – In this module it has configuration parser that parses into transfer function *.tf files. This module is written is python. It contains different parser for different router manufacturers like Cisco Parser, Juniper Parser and Openflow protobuf parser for Openflow virtual switches, graph to xml parser.

Mahak – In this module it has bundle that downloads the Openflow configuration files into Mininet. This module has been implemented in java.

NetPlumber - This module contains the major part of our project. It has all the rules defined and the code to check the rules and configuration policies. It also has graph importing and code to import topology from mininet and parse the data.

All the Rules and Policies are return using JSON object. these JSON object will be parsed and used in NetPlumber to check the rules. So adding and deleting a rule is very simple and easy. It saves all the rules and policies in JSON object. So processing will be very fast in real time

It is written both in C and C++ programming language. It also has code to check real time invariants in the network. This modules uses Hassel-c. Haccel-C has been exposed as an API in python to NetPlumber. So, NetPlumber uses this to implement the network wide invariant in realtime using HSA.

Mininet_builder – This module contains the mininet related configuration and also a python script that uses Mininet API to construct the Test backbone topology.

We have tried many different approach to find the link between the Mininet builder and the netplumber. We tried to contact the author of the Original paper but we couldn’t make a touch with him. And we have not got any documentation work on the code base, how does it works? How it connects with mininet and how it should be tested. We tried our own approach and successfully able to get the connection between other modules and make it work. But, unfortunately, we failed to find the connection between NetPlumber and Mininet. Also, Mininet was not working as per our expectation due to code maintenance problem.

Some of the approach we tried are below –

1. Module Integration of base version:

We initially started to implement in Ubuntu 16.04.1 as Networks related configuration will be easy in Linux when compared with the other Operating systems. And Our project was on SDN it will be relatively easy to implement on top of Linux system than on other. So to work with SDN we started to install Mininet on top of it and initially run with the default controller of

Mininet for configuration of SDN and run throught the base version of code.

Once we started with integration of all the modules without the documentation, we started to face lot of issues. We fixed some parts of the code that was not working by debugging all the files. So, this clarified the functionality and to understand the flow of the implementation.

Some modules are outdated and it was not in proper place for importing of modules. As the version of python, they have used was old and not compatible with the newer versions, some of them were not working. So, we fixed them placing it in proper place and where it can be reusable, scalable and extensible.

Some of the code was not working due to outdated maintenance. So it was very difficult to understand the functionality and to fix the issues that are coming on our way. Our intention was not fix the code.

We wanted to take out the dependency on the mininet controller and to port on to the RYU Controller which gives great flexibility and it supports Openflow 1.3. Also, some of the new features are supported in Openflow 1.3 [20].
But, we have spent all most half of our time in fixing of the issues and making it work as it was previously.

2. Different versions of Mininet:

We tried to integrate the code into different versions of Mininet as it was not working with the version 2.2.2 Because, the base source code was using beta version of Mininet. Although we have tried with different versions and changing the functions, it solved some interpreter errors. (e.g. add_switch vs. addSwitch). The mininet initially was using the Stanford backbone topology to test the NetPlumber. This has almost 500+ hosts, 100+ switches and 800+ links between them. As the Topology was too big it was taking too much time on the machines that we were testing and, we got so many errors in creating the topology. We tried with our own Topology to make it work. We could build our own topology but, unfortunately it was not connecting with NetPlumber. We tried to find the cause for this but, unable to find it.

3. Porting Netplumber to Ryu Controller from Mininet:

As mentioned in the abstract, our first goal was to port the Netplumber into RYU controller.
RYU is a component based software defined components with well-defined API that make it easy to create new network management and control applications. RYU supports Openflow 1.0, 1.2, 1.3, 1.4, 1.5. We initially started to port the part of the NetPlumber that was using the internal mininet controller. But, since mininet itself was not working we are unable to test it.

The original source code tries to evaluate the network monitor and test agents, it replicates the Stanford backbone network in Mininet, a container-based network emulator which uses Open vSwitch. We have tried using Ryu Controller with OpenFlow protocol API Reference.

4. Finding the link Netplumber with Mininet:

When we are trying to port the NetPlumber to the RYU controller, we have a great challenge to find the application that connects to mininet controller and executes the NetPlumber. We tried every possible way by debugging through all the connecting modules from Hassel-c, HSA-Python, Mahak, NetPlumber and Mininet builder. We were trying all possibilities to find the connection between the NetPlumber and Mininet. There was no specific code that connects it and we didn’t find any documentation of the original source code.

5. Different available Source Code – ONOS:

With the disfunction of the original source code that was using Mininet and was implemented in python script. We found a different source code that implements the NetPlumber in a different programming language.
ONOS Open Network Operating System is a software defined networking OS for service provider that has scalability, high availability, high performance and abstraction to make it eay to create apps and services. The platform is based on a solid architecture.

The source code is a java web application using ONOS. We have tried building the application using Apache Maven, Groovy Grape and Gradle/Grails … etc. However, the application code was also outdated and not maintained by any of the original authors who developed the web application. We are not able to compile the code.

6. Contacting the Author and others:

When we started getting lot of issues with the code and thought of contacting other people if any were/are trying/tried to implement the NetPlumber.
We tried to post in blogs where other developers are trying to fix the issues and develop their own requirement using this base version. But there was no help available from those blogs as all were struggling in some or the other way. And we were the only people who have code base in GitHub and some of the developers started to fork our sorce code and started to raise the issues in our implementation.
So, when we didn’t get any help other than authors. We were only left with one way to approach the original authors.
We have tried to mail Dr. Peyman Kazemian via his Stanford’s email which is already invalid. Then, we tried to connect him via LinkedIn by inMail. Unfortunately, He didn’t get back to us.So, our problem with finding the link between the NetPlumber and the Mininet and also the Mininet error were remained unsolved.

Section 2: Research Approach - Enhancement

1. Goal

Because NetPlumber is a real-time policy checking tool which based on the header space analysis framework. NetPlumber incrementally checks for the compliance of state changes using a novel conceptual representation, called the Rule Dependency Graph, that maintain a dependency graph between rules. Since it provides the convenience of adding and deleting rules.

We want to add one more feature to it by providing an option to exempt the rules whenever the network administration wants. He doesn’t need to go into each and every router and configure to exempt the rules. Instead he just can change an option in the NetPlumber and do whatever he wants.

So, in the way of providing an option we wanted to tell the Network administrator that by the exempting the rule what are the errors or is it opens any backdoor in the network and notify him in advance.

A new additional way to determine editing the rules by providing reliable statistical data. With the data, the higher value of the rule indicates that it would have more impacts to the entire network. Basically, it means that this rule is good or harmful.

2. Approach

We are using **Girvan-Newman Algorithm**. The Girvan–Newman algorithm detects communities by dynamically expelling edges from the original graph (Network). The associated parts of the rest of the network are the groups or communities. Instead of trying to construct a measure that tells us which edges are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities.
The algorithm's steps for community detection are summarized below (Figures come from Mining of Massive Datasets, Jure Leskovec, Anand Rajaraman, Jeff Ullman)

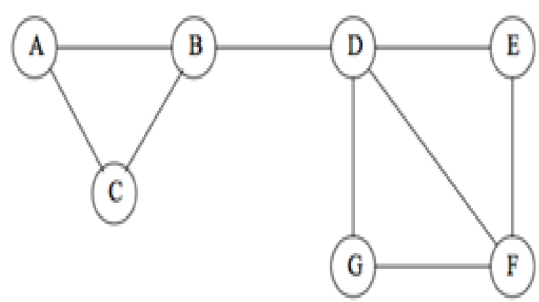


Fig 3.1 Example of a small network
Computing betweenness of all edges:

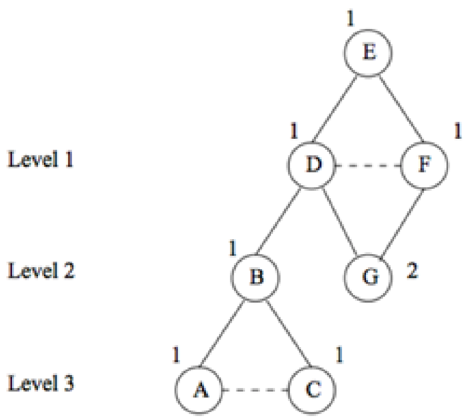


Fig 3.2 First step of the Girvan-Newman Algorithm

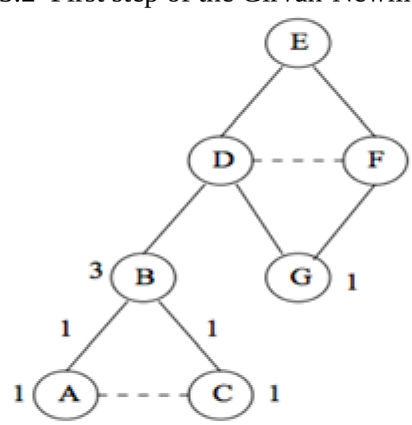


Fig 3.3 Next step of the Girvan-Newman Algorithm

B gets credit 3 from (A+C+1)

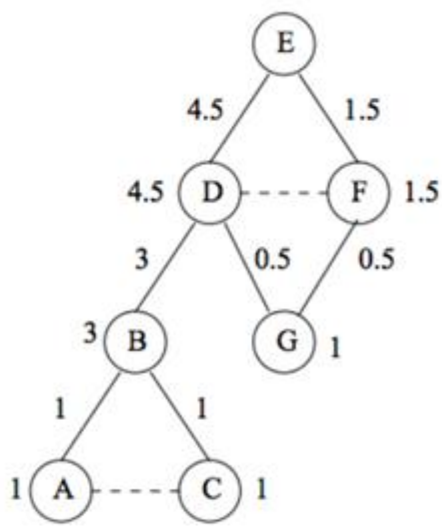


Fig 3.4 Completing the credit calculation

D and F each have credit 1, so shared equally (1 / (1+1) = 0.5)

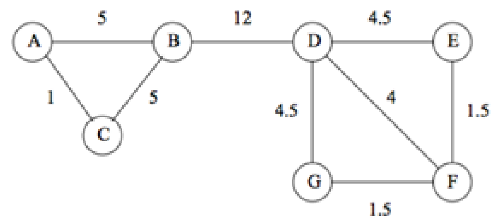


Fig 3.5 Result of the betweenness scores

To test our implementation of the algorithm is working. Also, because of the mininet is not working. We uses a mirror data of twitters followers to create our testing network and write a python code using Networkx and Scikit libraries. In this case, if we put it together with Hassel-C, nodes will represent as routers or hosts and edges will represent as rules. Initially, we tried to use the girvan_newman() function from Networkx. However, the result is not working properly. Then we implemented the algorithm manually from BFS to bottom up and computing betweenness0

IV. EXPERIMENTAL RESULTS

We have got some good results after trying a lot to work with Hassel-c, HSA-Python, NetPlumber and Mininet.

The below results shows how the HAS transforms the packet data between two ports in a router. This transformation happens in controller and the router just sends the packet based on this. The transformation is performed with help of transformation function that are defined in the HSA- Python.

At the end of this it also shows the time it took to transform all the incoming packets.

The below screenshot is just an example between two ports within the Router. But this transformation happens in Controller. And all these things happen in real time when it relates to netplumber

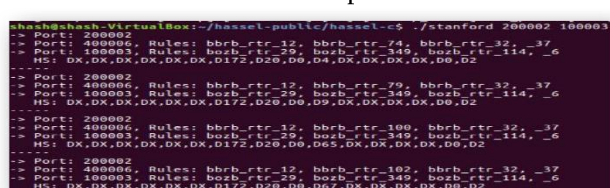


Fig. 4.1 Hassel - C Header transformation

The figure show the Hassel-C Header transformation, by ignoring the protocols and by treating the header bits as a series of 1 and 0 bits.

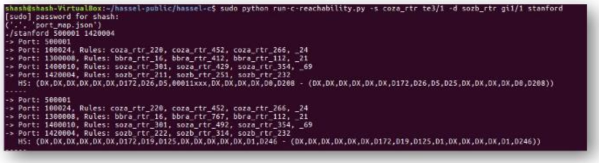


Fig. 4.2 Reachability of the network from Source to given destination

Given N number of source and M number of destinations, we are testing whether the destination is reachable from the source.

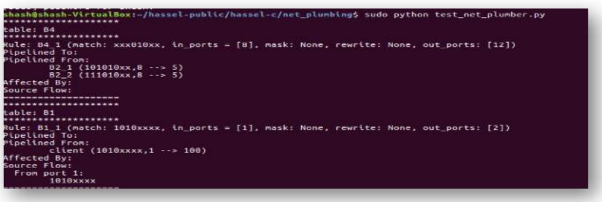


Fig. 4.3 Testing NetPlumber - Header space transformation with matching pattern

Basically, the header space bits are transformed and matched with the nearing pattern in Hassel-C

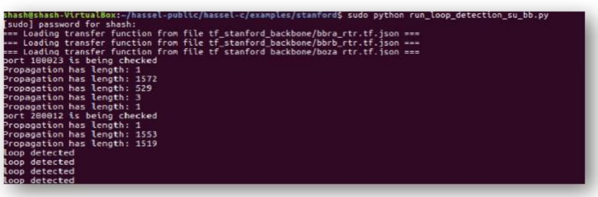
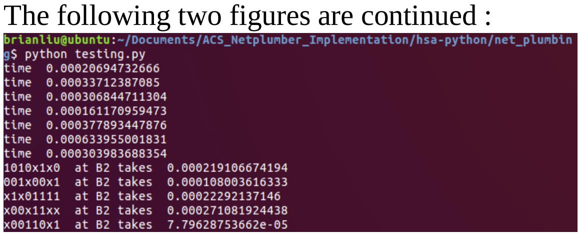


Fig. 4.4 Loop detection using Net-Plumber

The figure explains the existence of any loops in the Net-Plumber. Such a loop can be avoided and a short path can be chosen to transfer the packets

The following two figures are continued :



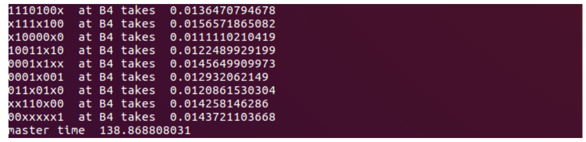


Fig. 4.5 Testing Net-Plumber

In testing.py, basically, we add a few simple links and rules and test in Net-Plumber. The result is in Fig. 4.5.

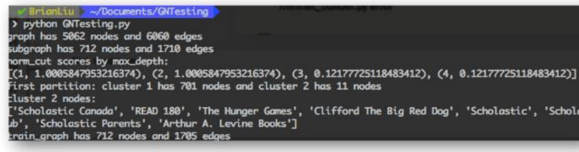


Fig. 4.6 Result of Girvan-Newman Algorithm

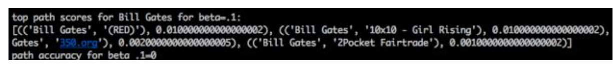


Fig. 4.7 Result of Girvan-Newman Algorithm – Scores

Girvan–Newman algorithm detects communities by dynamically expelling edges from the original graph. Instead of trying to construct a measure that tells us which edges are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities

V. CHALLENGES FACED AND LESSONS LEARNED

Challenges Faced:

Understanding the Hassel-C and Netplumber code base:
The first major issue that we have faced is that it is extremely difficult for us to understand the source code because it is a huge implementation and it is written in C, C++, Python and Java.

Finding the link between Hassel-C and Netplumber:
Hassel - C and Netplumber are implemented separately by using the outdated libraries. We faced issues in trying to establish a link between them and trying to find a way where they interact with each other.

Fitting the tools in between mininet and Ryu controller:

Some efforts were spent on the ways to fit the Netplumber and Hassel-C in between our mininet and the controller so that they interact effectively and filter the network wide invariants.

Defining rules in the Netplumber:
Trying to understand the various packet forwarding rules which was defined by the author took some considerable efforts.

Ways to display the results:
Trying to print some preliminary results by customising according to the original author[1] defined topology. For example, what are the results in each test case and how it shows the information of results.

Build a custom topology in mininet:
Instead of using the topology used by the authors, we planned to build a custom topology with X number of switches and Y number of routers.

Lessons Learned:
Understanding the Hassel-C and Netplumber code base:
After investing time and efforts and going through the code base, we were able to figure out how these tools work in conjunction with each other.

Finding the link between Hassel-C and Netplumber:
Hassel-C has been written in C and are exposed as python APIs. .We have fixed the python files that exposes APIs, that can be used alongside with NetPlumber API.

Fitting the tools in between mininet and Ryu controller:
We have fixed some of the existing libraries which will interact with the mininet topology and the tools and in-turn with the controller.

Defining rules in the Netplumber:
Some of the packet forwarding rules were analysed after backtracking the code and comparing them with the results.

Ways to display the results:
We have grouped our source code in such a way that, we can display some of the intermediate results such as creating a custom topology in mininet, interaction between the custom topology and the mininet controller, reachability of nodes, working of netplumber, Hassel-C, loop detection in network etc. by executing their makefile or by compiling the .py files.

Build a custom topology in mininet:
As the mininet is not working, we ended up using the default Stanford topology.

VI. CONCLUSION AND FUTURE WORK

The author introduced Header Space Analysis (HSA) as a protocol-independent model for forwarding functionality of networks.

Then we are trying to improvise the Netplumber by building an additional way to determine editing the rules by providing reliable statistical data.

For future work there are two phases:
First, although the improvising method has already been tested, it is still tested by mirroring data.

For better and more accurate testing result, we must fix the Mininet issues to test our improvising method with real data. We will try to maintain connections to all the other developers working on Hassel-C. After that, our team should start the Implementation of this component in Ryu Controller and test the performance after transferring to Ryu.

Second, once everything is done. We will like to complete the entire library and update our Github repository by completing documentation. And Other users have already opened issues asking us to provide examples. We will provide sufficient and clear examples for other developer to understand the completely usage.

VII. REFERENCE

[1] Real Time Network Policy Checking using Header Space Analysis, P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte, in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13)

[2] Hassel-Public https://bitbucket.org/peymank/hassel-public/wiki/Home

[3] Header Space Analysis - McKeown Group - Stanford University

[4] NSDI Talk https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[5] Header Space Analysis: Static Checking For Networks, Peyman Kazemian, George Varghese, Nick McKeown in 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12)

[6] Girvan–Newman algorithm https://en.wikipedia.org/wiki/Girvan–Newman_algorithm

[7] Mining of Massive Datasets - Jure Leskovec, Anand Rajaraman, Jeff Ullman

[9]http://www.cs.cornell.edu/conferences/formalnetworks/peyman-slides.pdf

[10]https://cis.temple.edu/~tug29203/16-5590/lectures/netplumber.pdf

[11]https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[12]https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final8.pdf

[13]http://security.riit.tsinghua.edu.cn/mediawiki/images/9/9a/NetPlumber_slides.pdf

[14]http://yuba.stanford.edu/~peyman/research.html

[15]https://bitbucket.org/peymank/hassel-public/branch/hassel-dev

[16]https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final8.pdf

[17]http://tiny-tera.stanford.edu/~nickm/papers/peyman-thesis.pdf

[18]https://www.ietf.org/proceedings/85/slides/slides-85-irtfopen-1.pdf

[19]http://tiny-tera.stanford.edu/~nickm/papers/peyman-thesis.pdf

[20]https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

VIII. APPENDIX

Key functions of GNTesting.py - Girvan-Newman Algorithm testing (The total file contains 500 lines)

```
def bfs(graph, root, max_depth):
    """
    Params:
      graph.......A networkx Graph
      root........The root node in the search graph (a string). We are computing
                  shortest paths from this node to all others.
      max_depth...An integer representing the maximum depth to search.

    Returns:
      node2distances...dict from each node to the length of shortest path from
                       the root node
      node2num_paths...dict from each node to the number of shortest paths from the
                       root node that pass through this node.
      node2parents.....dict from each node to the list of its parents in the search
                       tree.

    >>> node2distances, node2num_paths, node2parents = bfs(example_graph(), 'E', 5)
    >>> sorted(node2distances.items())
    [('A', 3), ('B', 2), ('C', 3), ('D', 1), ('E', 0), ('F', 1), ('G', 2)]
    >>> sorted(node2num_paths.items())
    [('A', 1), ('B', 1), ('C', 1), ('D', 1), ('E', 1), ('F', 1), ('G', 2)]
    >>> sorted((node, sorted(parents)) for node, parents in node2parents.items())
    [('A', ['B']), ('B', ['D']), ('C', ['B']), ('D', ['E']), ('F', ['E']), ('G', ['D', 'F'])]
```

```
    >>> node2distances, node2num_paths, node2parents
= bfs(example_graph(), 'E', 2)
    >>> sorted(node2distances.items())
    [('B', 2), ('D', 1), ('E', 0), ('F', 1), ('G', 2)]
    >>> sorted(node2num_paths.items())
    [('B', 1), ('D', 1), ('E', 1), ('F', 1), ('G', 2)]
    >>> sorted((node, sorted(parents)) for node, parents
in node2parents.items())
    [('B', ['D']), ('D', ['E']), ('F', ['E']), ('G', ['D', 'F'])]
    """
    ###TODO done
    queue = deque()
    num = 0
    queue.append(root)
    node2distances = {}
    node2num_paths = {}
    node2parents = defaultdict(list)
    node2distances[root] = 0
    node2num_paths[root] = 1
    poped = []

    while num <= max_depth and queue:
        node = queue.popleft()
        num = node2distances[node] + 1
        if num > max_depth:
            break
        poped.append(node)
        nei_list = graph.neighbors(node)

        for adjacent in nei_list:
            if adjacent not in queue:
                if node == root or (adjacent not in
node2parents[node] and adjacent not in poped):
                    queue.append(adjacent)
                    node2distances[adjacent] = num
                    node2num_paths[adjacent] = 1
                    node2parents[adjacent].append(node)
            else:
                if num <=
node2distances[adjacent]
        :
                    node2num_paths[adjacent] += 1
                if node not in node2parents[adjacent] and
node2distances[node] < node2distances[adjacent]:
                    node2parents[adjacent].append(node)

    return node2distances, node2num_paths,
node2parents
```

```
def bottom_up(root, node2distances, node2num_paths,
node2parents):
    """
    Params:
      root.............The root node in the search graph (a
string). We are computing
                       shortest paths from this node to all
others.
      node2distances...dict from each node to the length
of the shortest path from
                       the root node
      node2num_paths...dict from each node to the
number of shortest paths from the
                       root node that pass through this node.
      node2parents.....dict from each node to the list of
its parents in the search
                       tree
    Returns:
      A dict mapping edges to credit value. Each key is a
tuple of two strings

    >>> node2distances, node2num_paths, node2parents
= bfs(example_graph(), 'E', 5)
    >>> result = bottom_up('E', node2distances,
node2num_paths, node2parents)
    >>> sorted(result.items())
    [(('A', 'B'), 1.0), (('B', 'C'), 1.0), (('B', 'D'), 3.0), (('D',
'E'), 4.5), (('D', 'G'), 0.5), (('E', 'F'), 1.5), (('F', 'G'), 0.5)]
    """
    ###TODO done
    ret_dict = {}
    dist_list = []
    node_val_dict = {}
    for key, val in node2distances.items():
        dist_list.append((key, val))
        node_val_dict[key] = 1
    dist_list.sort(key=lambda tup: tup[1], reverse =
True)

    for key, val in dist_list:
        if key != root:
            for p in node2parents[key]:
                if p in node_val_dict:
                    node_val_dict[p] += (node_val_dict[key] /
len(node2parents[key]))
                else:
                    node_val_dict[p] = (node_val_dict[key] /
len(node2parents[key]))

    for key, val in dist_list:
        if key != root:
```

```
        for p in node2parents[key]:
            if p < key:
                tup_of_two_node = (p, key)
            else:
                tup_of_two_node = (key, p)
            ret_dict[tup_of_two_node] =
node_val_dict[key]/len(node2parents[key])

    return ret_dict


def approximate_betweenness(graph, max_depth):
    """
    Params:
      graph.......A networkx Graph
      max_depth...An integer representing the maximum
depth to search.

    Returns:
      A dict mapping edges to betweenness. Each key is
a tuple of two strings
      representing an edge (e.g., ('A', 'B')). Make sure
each of these tuples
      are sorted alphabetically (so, it's ('A', 'B'), not ('B',
'A')).

    >>>
sorted(approximate_betweenness(example_graph(),
2).items())
    [(('A', 'B'), 2.0), (('A', 'C'), 1.0), (('B', 'C'), 2.0), (('B',
'D'), 6.0), (('D', 'E'), 2.5), (('D', 'F'), 2.0), (('D', 'G'), 2.5),
(('E', 'F'), 1.5), (('F', 'G'), 1.5)]
    """
    ###TODO done
    ret_betweenness = {}
    for n in graph.nodes():
        node2distances, node2num_paths, node2parents =
bfs(graph, n, max_depth)
        bottom_up_dict = bottom_up(n, node2distances,
node2num_paths, node2parents)
        for e in bottom_up_dict.keys():
            if e in ret_betweenness:
                ret_betweenness[e] += bottom_up_dict[e]
            else:
                ret_betweenness[e] = bottom_up_dict[e]

    for key, val in ret_betweenness.items():
        ret_betweenness[key] = val / 2

    return ret_betweenness


def partition_girvan_newman(graph, max_depth):
    """
    Params:
      graph.......A networkx Graph
      max_depth...An integer representing the maximum
depth to search.

    Returns:
      A list of networkx Graph objects, one per partition.

    >>> components =
partition_girvan_newman(example_graph(), 5)
    >>> components = sorted(components, key=lambda
x: sorted(x.nodes())[0])
    >>> sorted(components[0].nodes())
    ['A', 'B', 'C']
    >>> sorted(components[1].nodes())
    ['D', 'E', 'F', 'G']
    """
    ###TODO done
    graph_c = graph.copy()
    ret_list = []
    ibet_dict = approximate_betweenness(graph_c,
max_depth)
    ib = sorted(ibet_dict.items(), key=lambda i: i[1],
reverse=True)
    components = [c for c in
nx.connected_component_subgraphs(graph_c)]
    while len(components) == 1:
        graph_c.remove_edge(*ib[0][0])
        del ib[0]
        components = [c for c in
nx.connected_component_subgraphs(graph_c)]
    for c in components:
        ret_list.append(c)

    return ret_list


def score_max_depths(graph, max_depths):
    """
    Params:
      graph........a networkx Graph
      max_depths...a list of ints for the max_depth values
to be passed
                   to calls to partition_girvan_newman

    Returns:
      A list of (int, float) tuples representing the
max_depth and the
```

```
    norm_cut value obtained by the partitions returned
by
    partition_girvan_newman.
  """
  ###TODO done
  ret_list = []
  for i in max_depths:
    components = partition_girvan_newman(graph, i)
    norm_val = norm_cut(components[0],
components[1], graph)
    ret_list.append((i, norm_val))

  return ret_list


def make_training_graph(graph, test_node, n):
  """
  Params:
    graph.......a networkx Graph
    test_node...a string representing one node in the
graph whose
            edges will be removed.
    n...........the number of edges to remove.

  Returns:
    A *new* networkx Graph with n edges removed.

  In this doctest, we remove edges for two friends of
D:
  >>> g = example_graph()
  >>> sorted(g.neighbors('D'))
  ['B', 'E', 'F', 'G']
  >>> train_graph = make_training_graph(g, 'D', 2)
  >>> sorted(train_graph.neighbors('D'))
  ['F', 'G']
  """
  ###TODO done
  temp_graph = graph.copy()
  nei_list = sorted (temp_graph.neighbors(test_node))

  for i in range(n):
    temp_graph.remove_edge( test_node ,nei_list[i] )

  return temp_graph


def path_score(graph, root, k, beta):
  """
  Params:
    graph....a networkx graph
    root.....a node in the graph (a string) to recommend
links for.
    k........the number of links to recommend.
    beta.....the beta parameter in the equation above.

  Returns:
    A list of tuples in descending order of score.

  >>> g = example_graph()
  >>> train_graph = g.copy()
  >>> train_graph.remove_edge(*('D', 'F'))
  >>> path_score(train_graph, 'D', k=4, beta=.5)
  [(('D', 'F'), 0.5), (('D', 'A'), 0.25), (('D', 'C'), 0.25)]
  """
  ###TODO done
  node2distances, node2num_paths, node2parents =
bfs(graph, root, math.inf)
  scores = []
  nodes_without_root_nei = set(graph.nodes()) -
set(graph.neighbors(root))
  for n in nodes_without_root_nei:
    if n != root:
      score = (beta ** node2distances[n]) *
node2num_paths[n]
      scores.append(((root, n), score))

  scores = sorted(scores, key=lambda x: x[0])[:k]
  return sorted(scores, key=lambda x: x[1],
reverse=True)[:k]
```