A Dynamic Algorithm for Loop Detection in Software Defined Networks

David Kordalewski and Robert Robere
University of Toronto

December 14, 2012

Abstract

A potential problem in computer networks is the existence of loops, which are cyclical paths through the network’s switches that can cause some packets to never leave the network. We present a dynamic algorithm built on the Header Space Analysis of Kazemian et. al. [1] which allows the detection of loops in software defined networks — such as ones created using OpenFlow [3]— over a sequence of rule insertions and deletions on the network’s switches. A key ingredient in our algorithm is a dynamic strongly connected component algorithm by Roddity and Zwick [5]. After a polynomial time pre-processing stage on the network description, our algorithm detects if the insertion or removal of a particular rule in a software defined network will cause some collection of headers to loop in the network. We present a theoretical and experimental analysis of our algorithm, and conclude that in densely connected networks (i.e. much more highly connected than in practice), it can detect negative network behaviour while sustaining a reasonable amount of rule insertions per second, as long as the rule insertions are batched.

Contents

1 Introduction 2
  1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Definitions 3

3 Header Space and Directed Graphs 6
  3.1 Rule Graphs and the Dynamic Loop Detection Problem . . . . . . . . . 6
  3.2 A Dynamic Strongly Connected Component Algorithm . . . . . . . . 9
  3.3 Combining the Rule Graph and DynSCC . . . . . . . . . . . . . . . . . 11
In a Software Defined Network (SDN), the behaviour of the underlying switches is determined globally, by a controller which dictates the local rules each switch should use to direct packets it encounters. This global control is a significant benefit of SDNs, but also a source of potential pitfalls.

In a SDN framework like OpenFlow it becomes possible to write network operating systems that can accommodate suites of applications which work together on the controller to make the network run smoothly. Using such a system we can write applications that work in tandem to perform different functions in a network such as network visualization, traffic analysis, or traffic simulation. Unfortunately, with an abundance of applications comes the possibility of mutual interference which could adversely impact the network. Thus, there is now a useful place in the SDN toolbox for applications that can provide real-time debugging information about a network.

For example, consider the problem of routing consistency. When the routing instructions sent to switches are generated by complex chains of modules it can become difficult or impossible to provide theoretical guarantees that the network will avoid certain types of bad behaviour. One of the most obvious ways in which a SDN can misbehave is by permitting packets to loop through their switches without ever reaching a client. This sort of behaviour can lead to increased latency, power consumption, and can possibly be targets for attack.

In [1], Kazemian et. al. tackled this problem on static (not software defined) networks using a theoretical model that they call Header Space Analysis. Unfortunately, their framework does not extend to dynamic, software defined networks. Thus, following their work, we have developed algorithms for detecting loops on functioning OpenFlow networks and built an application that can not only detect looping behaviour in networks, but prevent, in real time, flow table entries that would cause such behaviour from being sent to OpenFlow switches.

By monitoring a network from its inception (or taking some time to initialize our data structure from the current state of a network), we can verify that every new flow table entry and every withdrawn entry will not cause any packets to loop in the network for a relatively small amortized cost. Our application can be customized to make
controllers refuse to install flow table entries that would result in looping behaviour or simply log details of the errant flow that can then be monitored by other software or a network administrator to enable interactive development of the network.

1.1 Background

OpenFlow is a specification for SDN defining the behaviour of OpenFlow switches and a protocol for communication between switches and a controller [3]. The objective of OpenFlow is to define a uniform standard for switches that is relatively cheap and easy to implement in hardware and which also permits global control of the switches' behaviour by a single controller. In OpenFlow, the controller has a complete view of the network topology and complete control over flow routing through the instructions it gives the switches in the network.

The framework of Header Space Analysis, presented in [1], provides a geometric interpretation of packet flows in networks. The authors begin by modelling packets as points in a discrete geometric space and define operators characterizing how a packet moves and is transformed through a network. By then constructing these functions for real functioning networks, they can build a model of the network that can be queried for properties like “can user A communicate with user B?” or “is there any packet that user C can send through switch D that will not leave the network?”. In our work, we extend this framework to a graph theoretic one, which allows us to consider dynamic rule insertions and deletions over time much more simply.

2 Definitions

In this section, we will introduce many of the basic notions of header space analysis which fuel our dynamic algorithm. For a set $S$, we will use $\mathcal{P}(S)$ to denote it’s power set. If we have a set operator $\alpha : S \rightarrow S$, and a subset $I \subseteq S$, we will use $\alpha(I)$ to denote the image of $I$ under $\alpha$. We assume an introductory background in complexity theory, and so will freely draw on asymptotic notation such as $O(f(n))$, $\Omega(f(n))$, and the notion that polynomial time is efficient.

We ask that the reader recalls the following algorithms, which we will not reproduce for space considerations. The first is the LCA algorithm, which, when given a forest $F$, can answer queries of the following form: given two leaves $u$ and $v$, what is their lowest common ancestor, if it exists? The algorithm by Harel and Tarjan [6] allows us to perform $O(1)$ queries on leaves, after an $O(n)$ time pre-processing stage on $F$.

Second, an algorithm for strongly connected component detection. If $G = (V, E)$ is a directed graph, recall that a strongly connected component is a subgraph $S \subseteq G$ in which each pair of vertices $u, v \in S$ are connected by a path in $S$. The famous algorithm by Tarjan [7] allows us to construct all the SCCs of a graph $G$ in time $O(m + n)$, where $|E| = m$ and $|V| = n$.

Finally, the disjoint-set data structure, and it’s associated Union-Find algorithm.
The disjoint-set data structure tracks a set of elements partitioned into a collection of disjoint subsets, and allows two operations:

1. **Find**, which given an element \(v\) returns the representative element \(v^*\) of the partition \(v\) is contained in, and
2. **Union**, which, when given two elements \(v, v'\), merges their representative partitions.

The standard Union-Find algorithm that we consider was also developed by Tarjan, in e.g. [8], and runs in amortized \(O(\alpha(m, n))\) time, where \(\alpha(m, n)\) is the inverse Ackermann function.

The next definitions are inspired from Kazemian et. al. [1]. We consider packets in a network to be defined by their headers, which we view as \(L\) length strings \(x \in \{0, 1\}^L\). We therefore refer to the space \(\mathcal{H} = \{0, 1\}^L\) as **header space**. To specify subsets of headers in header space we will use **wildcard expressions**, which formally are strings \(w \in \{0, 1, *\}^L\). We say \(w\) **matches** a string \(x \in \mathcal{H}\) if for all \(1 \leq i \leq L\), either \(x_i = w_i\) or \(x_i \neq w_i = *\), and we will use \(\mathcal{W}\) to denote \(\{0, 1, *\}^L\). Any subset of \(\mathcal{H}\) can be represented as a union of wildcard expressions, and so for the remainder of the paper, we will say that a set of wildcard expressions \(I \subseteq \mathcal{W}\) **matches** a header \(h\) if any wildcard expression in \(I\) matches \(h\), and we will consider subsets of \(\mathcal{W}\) to implicitly be unions of wildcard expressions. A **packet flow** is defined as a subset of \(\mathcal{H}\).

We will often use wildcard expressions to refer to their underlying sets, and so will use set-theoretic operations on wildcard expressions without any special mention — e.g. \(\cup, \cap, \setminus\). We note that there are simple algorithms for determining the union, intersection, difference, and complement of two wildcard expressions, each of which run in time linear in the length of the expression and return the result as a union of wildcard expressions (see [1, Section 4]). Thus, in our notation, the result of any of these operations will be returned as a subset \(I \subseteq \mathcal{W}\).

In Header Space Analysis, a network is modelled as a collection of **switches**, each of which has a unique set of incoming and outgoing ports. We will slightly modify this definition so as to sit closer to the conceptual framework of software defined networks. In OpenFlow [3], for example, when a packet enters a switch, it’s ingress port is concatenated to the header and a wildcard expression on the switch is used to match the port as well as the other elements of the header. We will take our cue from this behaviour, and model a switch as a collection of **rules**, with the network topology and the actions of a switch represented as a function we call the **transfer operator**.

**Definition 1.** A function \(f\) is a **transfer operator** if it maps unions of wildcard expressions \(H \subseteq \mathcal{W}\) to a sequence \((H_1, s_1), (H_2, s_2), \ldots, (H_k, s_n)\), where \(S = \{s_1, \ldots, s_n\}\) is the set of all switches in the network and \(H_1, H_2, \ldots, H_k \in \varnothing(\mathcal{W})\) are each unions of wildcard expressions. We also require that such an operator satisfies the following conditions:

1. If \(h \in \mathcal{H}\) is a header, then for all \((H_i, s_i)\) pairs in \(f(h)\), we have that \(H_i \in \mathcal{H} \cup \emptyset\) (\(f\) maps headers to headers).
2. If $H \subseteq \mathcal{H}$ is a packet flow matched by $I \subseteq \mathcal{W}$ and $H' \subseteq H$ is a subflow matched by $I'$, then we require $f$ to be downward closed in the following sense: if $f(I) = \{(H_1, s_1), \ldots, (H_n, s_n)\}$, $f(I') = \{(H'_1, s_1), \ldots, (H'_n, s_n)\}$, then $H'_j \subseteq H_j$ for all $j$.

The transfer operators determine how packets flow and are re-written throughout the network. We can combine these operators with wildcard expressions to make rules:

**Definition 2.** A rule is a pair $(m, f)$, where $m \in \mathcal{W}$ is a wildcard expression and $f : \wp(\mathcal{W}) \to \bigcup_{i=1}^n \wp(\mathcal{W}) \times \{s_i\}$ is a transfer operator on the space of headers mapping wildcard expressions to wildcard expressions and other switches.

We assume oracle access to the transfer operators, and so we will not concern ourselves with their complexity. If $f$ is a transfer operator, and $s$ is a switch, then we define $f(w)|_s = w' \in \wp(\mathcal{W})$ if $(w', s) \in f(w)$ to be the union of wildcards that $f$ sends to switch $s$, and $f(w)|_w = \{s' \in \mathcal{S} \mid (w', s') \in f(w) \text{ and } w' \neq \emptyset\}$ to be the set of switches to which $f$ can send $w$ after transformation.

We now introduce the formalization of a network which we will use for the rest of the paper.

**Definition 3.** A network $N = (\mathcal{S}, \mathcal{L})$ is defined as follows. The collection $\mathcal{S} = \{\bot, s_1, s_2, \ldots, s_m\}$ is a set of switches, where $\bot$ is defined as the null switch, and represents if a packet was generated at a particular switch in the network. With each switch $s_i \neq \bot$, we associate a totally ordered list of rules $L_i \in \mathcal{L}$. We say that switch $s_i$ is connected to switch $s_j$ if there exists two rules $(m_i, f_i) \in L_i$, $(m_j, f_j) \in L_j$ such that $f_i(m_i)|_{s_j} \cap m_j \neq \emptyset$.

Suppose that a packet with header $h$ has arrived at switch $s \in \mathcal{S}$ in the network. We can model how the header $h$ moves through the network iteratively as follows. For each rule in $s$, let $(m^*, f^*)$ be the first rule in the list $L_i$ for which $m^*$ matches $h$. We consider $f^*(h) = \{(h_1, s_1), (h_2, s_2), \ldots, (h_n, s_n)\}$, which we interpret as sending the packets with header $h_j$ routed to switch $s_j$ (or sending no header if $h_j = \emptyset$). As previously remarked, we assume w.l.o.g. that the ingress port information is stored as a field of packet header, and when the transfer operator routes a packet from one switch to another, that field is re-written to the ingress port on the target switch.

We make some remarks about the connection between our networks and software defined networks. Suppose, for a moment, that we have a packet flow arriving at particular switch. How that packet flow is transformed and re-routed inside of the switch can change from one point in time to another, as the routing rules inside the switch can be changed by a controller sitting outside of the network. We can model this rather simply: the network topology will remain fixed, and the internal routing rules for each switch can change from one point of time to another. Most importantly, the
set of switches to which a transfer operator can send packet flows will not change from one point in time to another.

Throughout our analysis in the rest of the paper, we will consider the controller to be an adversary — that is, we will prove upper bounds on our ability to detect loops in the network over a sequence of arbitrary rule insertions and deletions. In fact, it will be much easier to model our network as a directed graph, and so our next job will be to translate the concepts of header space analysis into the language of graph theory.

3 Header Space and Directed Graphs

3.1 Rule Graphs and the Dynamic Loop Detection Problem

In this section, we will show how to model a network in the sense of the previous section as a directed graph. Through this translation we can introduce notions from dynamic graph algorithms to help us compute port to port reachability in our network over rule updates in an efficient manner. In a sense, one can view these graphs as a generalization of the propagation graphs introduced in [1].

The rough idea is this. Suppose again that we have \( n \) switches in a set \( S = \{s_1, \ldots, s_n\} \), where each switch \( s_i \) has an associated rule list \( L_i \). We will model the network as a digraph \( G_N = (V_N, E_N) \) where vertices are the rules occurring in the rule lists, and there is an edge between rule \( i \) and rule \( j \) if there are headers, which after being matched by rule \( i \) and transformed by the associated transfer operator, can be matched by rule \( j \). Of course, in a software defined network, the network transfer function can change from one point in time to another, and so by applying a dynamic SCC algorithm (which we present in the next section) we can maintain network safety even with this dynamic behaviour.

Definition 4. Consider a network \( N = (S, L) \) specified by a set of switches \( S = \{s_i\}_{i=1}^{n} \), where each switch \( s_i \) has an associated rule list \( L_i \in L \).

We define the rule graph \( G_N \) as follows. The vertices \( V_N \) of \( G_N \) is the collection of all rules in the network, so \( V_N = \{L_i\}_{i=1}^{n} \). The edges \( E_N \) of \( G_N \) are defined as follows. If \( u = (m_u, f_u), v = (m_v, f_v) \) are rules in \( V_N \) on switches \( s_i \) and \( s_j \) respectively, then we add the edge \((u, v)\) if \( f_u(m_u)|_{s_j} \cap m_v \neq \emptyset \).

We can clearly construct a rule graph from a network \( N = (S, L) \) efficiently in a brute-force manner. Define the degree of a switch \( s \) to be number of switches that \( s \) is connected to in the network.

Proposition 1. Given a network \( N \) specified by a collection of \( n \) switches, where each switch \( s_i \) is specified by a list of rules \( L_i \), we can construct a rule graph \( G_N \) in \( O(ndR^2) \) time, where \( R \) is the maximum number of rules on any switch and \( d \) is the maximum number degree of any switch in the network.
Proof. We will give an algorithm which constructs $G_N$ in the time specified. Fix a switch $s_1$, and add each rule from $L_1$ to the graph $G_N$. Now, for each switch $s_i \neq s_1$, and for each rule $(m, f) \in L_i$, perform the following. Let $S$ be the set of all switches connected to $s_i$. Then $|S| \leq d$ by definition. For each switch $s_j \in S$, and each rule $(m', f') \in L_j$, we add an edge from $(m, f)$ to $(m', f')$ if $f(m)_{s_j} \cap m' \neq \emptyset$.

We can consider how a packet header $h$ will move in the network by considering how it travels in the rule graph $G_N$. Suppose, as we did in the previous section, that the header $h$ was sitting on a rule $(m, f) \in V_N$ (and so is implicitly in the switch $sw(m, f)$). We can determine where the header will “move next” — that is, how the header is transformed and sent to a collection of new switches — by considering the action of the transfer operator $f$ on $(h, sw(m, f))$. By then considering the transformations of the new headers on the new switches, we can track the movement of all the packets generated from a single packet $h$ inductively throughout the network. Algorithm 1, PacketSimulation, formalizes this idea.

Algorithm 1: PacketSimulation($G_N, h, r$)

Input : A rule graph $G_N$, a packet header $h$ beginning on a rule $r = (m_r, f_r) \in G_N$ for which $h \in m_r$.
Set $S = \{(h, r)\}$; Set $Visited = \emptyset$;
while $S \neq \emptyset$ do
  for $(h, r) \in S$ do
    Denote $r = (m, f)$;
    Set $s$ to be the switch containing $r$;
    Set $o = f(h)$;
    for $(h', s') \in o$ do
      if $h' \neq \emptyset$ then
        Let $(m', f')$ be the first rule which matches $h'$ on switch $s'$;
        Add $(h', (m', f'))$ to $S$;
      end
    end
    Remove $(h, r)$ from $S$ and add it to $Visited$;
  end
return $Visited$

In the context of SDNs, the vertices in the rule graph $G_N$ defined above will no longer be fixed. As we mentioned before, the routing rules appearing inside the switches can change from time to time as governed by the controller. We therefore view the actions of this controller as dynamic insertions or removals of vertices and edges in the rule graph. We will make the assumption that when a new vertex is inserted into a rule graph, all of the edges described in Definition 4 will be inserted along with that vertex.
Using Algorithm 1 we can formally state the problem we will study.

**Problem 1 (The Dynamic Loop Detection Problem).** We seek an algorithm which, over a sequence of vertex insertions and deletions in a rule graph $G_N$, will maintain the following invariant: for every cycle $C$ in $G_N$, there does not exist a pair of headers $h, h'$ and a rule $r$ on $C$ for which, over the course of the computation of $\text{PacketSimulation}(G_N, h, r)$, the set $\text{Visited}$ will contain both $(h, r)$ and $(h', r)$.

Before we continue, continue the following case. Suppose that $G_N$ is a rule graph with a rule $r$ lying on a loop $C$ as described in the above problem. We say that Algorithm 1 detects $C$ if the pair $(h, r)$ and $(h', r)$ ever lie in the set $\text{Visited}$ at two different points in time of the computation for a pair of headers $h, h'$. We prove a complexity bound on the amount of time for Algorithm 1 to detect $C$. Recall that a graph is strongly connected if there exists a path between each pair of the graph’s vertices. We have the following Theorem:

**Theorem 1.** Let $G_N$ be a rule graph which is strongly connected, with $n$ vertices and $m$ edges, and let $r$ be a rule in $G_N$. If $r$ is contained in a cycle $C$ on which it is possible for a sequence of headers to travel, then $C$ will be detected in $O(n + m)$ time.

**Proof.** Without loss of generality, we consider the computation of $\text{PacketSimulation}(G_N, *^L, r)$. Let $S_i, V_i$ be the rules contained in $S, V \text{ised}$ after $i$ iterations of the outer loop. Clearly the algorithm will halt once $|V| \geq n$.

We can express $|S_{i+1}|$ and $|V_{i+1}|$ recursively in terms of $|S_i|$ and $|V_i|$ by observing that after a single execution of the inner loop of Algorithm 1, at most $a_i$ rules are added to $|S_i|$ and $|S_i|$ rules are removed from $S_i$, for some $a_i > 0$. Therefore,

$$|S_{i+1}| = a_i|S_i|$$
$$|V_{i+1}| = |V_i| + |S_i|$$
$$|S_0| = 1$$
$$|V_0| = 0$$

Note that after each execution of the inner loop the size of $V \text{ised}$ must increase by at least one by the above recursive definitions. It therefore follows that the inner loop can be executed at most $n$ times over the course of the computation without the same rule being inserted into visited twice. Moreover, each edge can only be considered at most once before we add a rule to $V \text{ised}$ that has already been considered. Thus the total running time of the algorithm before it detects a loop is $O(n + m)$. \qed

We end this section with a motivating idea. Recall that in a graph $G = (V, E)$ a strongly connected component (SCC) is a subgraph $G'_v$ of $G$ in which there is a directed path between every pair of vertices $u, v \in G'_v$. Clearly, any cycle must be contained in an SCC. Thus, if in a rule graph $G_N$ we maintained a list $L$ of the SCCs of the graph, we could determine if the graph $G_N$ was loop free by applying Algorithm 1 to each SCC in the graph, and halting if any rule in the SCC is visited twice. In the next section, we
will introduce a *dynamic strongly connected component* algorithm which maintains a list of SCCs over any sequence of edge insertions and deletions, and allows us to query in constant time whether two vertices are contained in the same SCC.

### 3.2 A Dynamic Strongly Connected Component Algorithm

Next we introduce and discuss an algorithm which allows us to dynamically track the SCCs in a graph over a sequence of edge insertions and deletions. Afterwards, we will show how to use this dynamic algorithm to solve the Dynamic Loop Detection Problem outlined at the end of the previous section. The particular algorithm we will use is from [5], where it was originally created as a stepping stone to a more efficient dynamic reachability algorithm. In this section we will give a high-level description of the algorithm, and leave the formal definition in pseudocode for Appendix A.1.

Consider an input graph $G = (V, E)$, and suppose that $E_1, E_2, \ldots, E_k$ are a sequence of sets of edges not appearing in $G$, and each set of edges in the sequence is disjoint — formally, $E \cap E_i = \emptyset$ and for all $i, j \in \{1, 2, \ldots, k\}$, $E_i \cap E_j = \emptyset$. Then we can consider a corresponding sequence of graphs $G_1, G_2, \ldots, G_k$, where each graph $G_i$ in the sequence is generated by adding to $E$ each of the edge sets $E_1, E_2, \ldots, E_i$, so $G_i = (V, E \cup E_1 \cup E_2 \cup \cdots \cup E_i)$.

If we wanted to track the strongly connected components in the graphs $G_i$ for each $i$, then the following is a simple observation that will help us.

**Observation 1.** If $S$ is a strongly connected component of the graph $G_i$ for $i \leq k$, then for all $j \geq i$, there exists a strongly connected component $S'$ of $G_j$ for which $S \subseteq S'$. We call this the *Monotonic Components Property*.

From this, we know that if two vertices $u$ and $v$ ever appear together in some strongly connected component in the graph $G_i$, then they are guaranteed to remain in some strongly connected component together for every graph $G_j$ with $j > i$. By exploiting this, the Monotonic Components Property allows us to greatly decrease the amount of space needed in storing the strongly connected components over a sequence of rule inserts.

A natural way to model this behaviour formally is with what Roddity and Zwick call a *component forest* [5]. The leaves of the component forest are exactly the nodes appearing in the graph $G$, and the internal nodes of the component forest correspond to SCCs appearing in some version of the graph in the sequence. Formally, the internal nodes $s$ in the component forest are labelled with version numbers $version(s)$, so that $s$ corresponds to an SCC $S$ which first appears in the graph $G_{version(s)}$, and the leaf descendants of $s$ are exactly the vertices in $G$ appearing in $S$. Now, suppose that $u$ and $v$ are internal nodes in the component forest corresponding to SCCs $U, V$ with $U \subset V$, where $U$ and $V$ first appear (respectively) in graphs $G_i, G_j$ with $i < j$. Then, by the Monotonic Components Property, we can always make $v$ an ancestor of $u$ in the component forest. This implies a natural way to check if two vertices $a, b$ appear in the
same strongly connected component, by running a least common ancestor algorithm on
the forest and then just checking the LCA of the two leaves corresponding to \( a \) and \( b \).

To make the SCC updates faster, we maintain in parallel a Union-Find algorithm
storing the latest SCCs of the graph, which allows us to perform SCC lookups and
merges with each new set of edge insertions in \( O(\alpha(m, n)) \) time, where \( \alpha \) is the inverse
Ackermann function. To search for new SCCs after the most recent rule insert, we use
the Tarjan’s algorithm (cf Section 2) which detects SCCs in a graph in \( O(m + n) \) time.

An obvious problem lies with deletions, as it’s possible that the Monotonic Compo-
nents Property will no longer be satisfied if we allow a set of edges to be removed from
any graph \( G_i \) in the sequence. However, this problem can be solved by the following
observation:

Observation 2. If, when deleting a set of edges \( E' \), we delete the edges from each version
of the graph, then the Monotonic Components Property will remain satisfied.

So, after deleting a set of edges from each version of the graph, we can re-build
the component forest and the Union-Find algorithm by running an SCC detection
algorithm on each version of the graph, and building them up step-by-step. This is the
more expensive operation of the two, but Roddity and Zwick observe that the running
time of edge deletions can be reduced to amortized \( O(m \cdot \alpha(m, n)) \) time [5].

Therefore, we have an algorithm DynSCC which supports the following operations
on a graph \( G = (V, E) \) (the formal definitions of the algorithms appear in Appendix
A.1):

- \textbf{insert}(\( E \)): Create a new, identical version of the graph \( G \) and add the edges \( E \) to it.
- \textbf{remove}(\( E \)): Remove the set of edges \( E \) from all versions of the graph \( G \).
- \textbf{query}(\( u, v, i \)): Return 1 if \( u \) and \( v \) are in the same strongly connected component in
  the \( i \)th version of \( G \).

The running times of the algorithm DynSCC are as follows:

- \textbf{insert}(\( E \)): \( O(m \cdot \alpha(m, n)) \),
- \textbf{remove}(\( E \)): \( O(m \cdot \alpha(m, n)) \),
- \textbf{query}(\( u, v, i \)): \( O(1) \).

Before we discuss how to apply this algorithm to the Dynamic Loop Detection Prob-
lem, we will make several useful extensions to the DynSCC algorithm. First, suppose
we were making a query \textbf{query}(\( u, v, i \)) to two vertices \( u \) and \( v \) and a version number \( i \).
With the component forest structure underlying our algorithm above, we can not only
return if \( u \) and \( v \) are in the same strongly connected component in the graph \( G_i \), but
also return the strongly connected component itself. This operation (which we will call
\textbf{getSCC}(\( u, v, i \))) is very simple — run an LCA query on the vertices \( u \) and \( v \) and get
their least common ancestor \( p \), continue travelling up the tree until we find the highest
ancestor \( p' \) of \( p \) with version number not exceeding \( i \), and then perform a BFS and return all of the leaves appearing below the node \( p' \). The formal description of this algorithm appears in Appendix A.1 with the rest of the algorithm definitions, and we remark that the running time of this algorithm is \( O(n_{SCC}) \), where \( n \) is the number of vertices in the corresponding SCC.

Second, we note that the algorithm can easily be extended to inserting and deleting not only edges from the graph, but vertices as well. We therefore extend the definition of DynSCC with two more algorithms \( \text{insertV}(v) \), \( \text{removeV}(v) \). Inserting a vertex \( v \) can be performed by just adding it to the component forest without any other modification, and removing \( v \) can be performed by first calling \( \text{remove} \) on the set of the edges incident to \( v \), and then removing \( v \) itself. Once again, the formal definitions of these operations appears in Appendix A.1.

### 3.3 Combining the Rule Graph and DynSCC

We are now finally in the position to combine all of our ingredients into an algorithm which solves the Dynamic Loop Detection Problem. Given a description of a network \( N \) we use it to construct a rule graph \( G_N \), which can be done in time \( O(ndR^2) \) by Proposition 1. We then initialize a DynSCC structure on \( G_N \). An insertion of a rule \( r = (m_r, f_r) \) into the network will be handled by first enumerating the list of the rules \( L_R \) in the network which intersect with \( r \) by checking if \( f_r(m_r)|_{s_j} \cap m_j \neq \emptyset \) for any rule \( (m_j, f) \) on the switch \( s_j \). Once the list of intersecting rules is enumerated, we construct the corresponding set of edges \( E_r \), and insert \( r \) and \( E_r \) into \( G_N \) by a call to \( \text{insertV}(r) \) followed by a call to \( \text{insert}(E_r) \). We can then check if an SCC was added to the graph in the most recent version \( t \) by a call to \( \text{query}(r, r', t) \) for each neighbor \( r' \) of \( r \).

If an SCC \( S \) is found by our query containing \( r \) and \( r' \), we can call \( \text{getSCC}(r, r', t) \) to retrieve it, log a “warning” message for the network administrator, and check if \( S \) contains a loop which is actually consistent with some header \( h \) by calling Algorithm 1 on \( S \). If we want to remove a rule \( r \) from the network, we can do this by making a call to \( \text{removeV}(r) \), and then DynSCC will handle all of the details for us.

Following the convention we set for DynSCC in the previous section, we will define the above as a dynamic algorithm \( \text{DynLoop} \), and we leave the formal implementation for Appendix A.2.

**initLoop**\((N)\): Initialize a rule graph \( G_N \) from a network description \( N \), and an instance of DynSCC defined on \( G_N \).

**insertRule**(\( r \))\: Insert a rule \( r \) into the rule graph \( G_N \), and update the DynSCC algorithm, while logging any SCCs.

**removeRule**(\( r \))\: Remove a rule \( r \) from the rule graph \( G_N \), and update the DynSCC algorithm.

**loopDetect**\((G_N)\): Return true if there exists a loop which headers can travel on in the current state of the rule graph \( G_N \).
The time it takes for insertRule and removeRule are directly dependent on the algorithms loopDetect, and insert, remove from the underlying DynSCC algorithm. From Theorem 1, we know that the running time of PacketSimulation, which loopDetect runs as a subroutine, is bounded by $O(n + m)$. However, we have to consider running PacketSimulation from each rule in the strongly connected component it receives to be sure that a loop exists. Thus, we have the following proposition.

**Proposition 2.** Let $G_N$ be a strongly connected component of a rule graph with $n$ vertices and $m$ edges. The algorithm loopDetect($G_N$) runs in time $O(n^2)$.

We therefore summarize the running times of each of the operations that our dynamic algorithm supports in the following list:

- **initLoop**($N$): $O(ndR^2)$ (from Proposition 1),
- **insertRule**($r$): $O(m \cdot \alpha(m, n) + n^2)$,
- **removeRule**($r$): $O(m \cdot \alpha(m, n))$,
- **loopDetect**($G_N$): $O(n^2)$.

## 4 Experimental Results

In order to verify that our algorithms are sufficiently efficient to run on functioning SDNs, we have implemented our algorithm for loop detection\(^1\) in Python and run simulated flow rule installations.

We simulate networks with a certain number of switches and clients per switch. The switches are first arranged in a ring with a port attached between every switch and its 2 immediate neighbours. We then add ports between every pair of unconnected switches with a probability given by the parameter *network density*. This sort of topology is implausible for real networks, and is intended to make loops very common thus forcing our algorithms to do as much work as possible to handle our random rule insertions. SCC maintenance and back tracing are the most intensive tasks needed to be done, and we reason that by ensuring that SCC growth is common, we exhibit worst case upper bounds on the number of rules that can be added per second.

Given this network topology, we insert random rules one at a time until the network reaches some predetermined capacity. All of our results in this section are recorded for networks that have this “steady-state” number of rules, while we add new rules and evict the oldest remaining one repetitively for the duration of the test.

The rules installed in the network are divided into two sets, which we call *forwarding rules* and *rewriting rules*. Forwarding rules take all packets that arrive at a small subset of some switch’s ports and relay them on some other small subset of the switch’s ports.

\(^1\)We have made our code publicly available at [https://github.com/kord/dkrr_loop_detection](https://github.com/kord/dkrr_loop_detection)
Rewriting rules also forward packets, but each particular rule rewrites the matched packet’s headers in some fixed (but randomly generated) way.

Our test parameters and results are presented in figure 1. All tests were executed on a laptop running Ubuntu 12.04, with a dual-core 2.13 GHz Intel P6200 with 4 GB of memory. Along with the parameters describing the simulated network and rules installed, we show some statistics describing the rule graph.

Tests 1 and 2 simulate a complete graph network topology and all of the rules installed are forwarding rules, which are more likely than rewriting rules to induce new edges in the rule graph. Tests 3 and 4, have a sparser topology and use a high proportion of rewriting rules and are intended to be somewhat more realistic than the networks in tests 1 and 2, though we still expect real networks to be modelled much more efficiently. Tests 1 and 3 are fairly small networks, with a small number of switches and clients per switch, while Tests 2 and 4 simulate larger networks.

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td># Switches</td>
<td>10</td>
<td>40</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Network Density</td>
<td>100%</td>
<td>100%</td>
<td>15%</td>
<td>15%</td>
</tr>
<tr>
<td># Clients/Switch</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td># Rules Installed in Steady State</td>
<td>1000</td>
<td>4000</td>
<td>1000</td>
<td>4000</td>
</tr>
<tr>
<td>Proportion of Forwarding Rules</td>
<td>100%</td>
<td>100%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>SCC Size For New Rules (Mean)</td>
<td>272</td>
<td>1683</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>SCC Size (Std. Dev.)</td>
<td>263</td>
<td>1247</td>
<td>99</td>
<td>34</td>
</tr>
<tr>
<td>Edges Added per Rule (Mean)</td>
<td>18.1</td>
<td>8.0</td>
<td>7.0</td>
<td>2.4</td>
</tr>
<tr>
<td>SCC Growth Occurrence Frequency</td>
<td>25.7%</td>
<td>32.3%</td>
<td>11.9%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Loop Existence Frequency Given SCC Growth</td>
<td>100%</td>
<td>100%</td>
<td>84%</td>
<td>20.6%</td>
</tr>
<tr>
<td>Rule Installs per Second (Mean)</td>
<td>13.0</td>
<td>2.18</td>
<td>20.0</td>
<td>14.5</td>
</tr>
</tbody>
</table>

Figure 1: Experimental Results for Simulated Networks (No Batching)

The results in Figure 1 appear to be particularly negative. Even in Tests 3 and 4, where the graphs are much more sparse and the size of the SCCs is drastically decreased, the mean number of edges added per rule is quite small. What is the problem?

Some simple code profiling discovered that Tarjan’s algorithm for SCC detection was the culprit. Whenever a rule is inserted into the graph, Tarjan’s algorithm is called by the DynSCC algorithm on the entire rule graph at that point. Since it runs in time linear in the size of the rule graph, regardless of the number of rules currently being inserted, this vastly increases the running time. A natural strategy to combat this is to increase the number of rules we insert at one time so as to amortize out the cost of the repeated calls, i.e., batch updating.

For the second set of tests we batch the rules into groups of 100, and then update the entire rule graph at the end of each batch. For all of the newly inserted rules, we then determine whether they are involved in any loops in the final rule graph. The results from these tests are presented in Figure 2.
<table>
<thead>
<tr>
<th></th>
<th>Test 5</th>
<th>Test 6</th>
<th>Test 7</th>
<th>Test 8</th>
</tr>
</thead>
<tbody>
<tr>
<td># Switches</td>
<td>10</td>
<td>40</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Network Density</td>
<td>100%</td>
<td>100%</td>
<td>15%</td>
<td>15%</td>
</tr>
<tr>
<td># Clients/Switch</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td># Rules Installed in Steady State</td>
<td>1000</td>
<td>4000</td>
<td>1000</td>
<td>4000</td>
</tr>
<tr>
<td>Proportion of Forwarding Rules</td>
<td>100%</td>
<td>100%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Mean SCC Size For New Rules</td>
<td>436</td>
<td>1788</td>
<td>65</td>
<td>8</td>
</tr>
<tr>
<td>Std. Dev. SCC Size</td>
<td>335</td>
<td>1299</td>
<td>110</td>
<td>34</td>
</tr>
<tr>
<td>Mean edges added per batch</td>
<td>2852</td>
<td>817</td>
<td>784</td>
<td>245</td>
</tr>
<tr>
<td>SCC Growth Occurrence Frequency</td>
<td>31.4%</td>
<td>35.2%</td>
<td>11.6%</td>
<td>2.2%</td>
</tr>
<tr>
<td>Loop Existence Frequency Given SCC Growth</td>
<td>100%</td>
<td>100%</td>
<td>89.2%</td>
<td>13.9%</td>
</tr>
<tr>
<td>Mean Rule Installs/Second</td>
<td>20.0</td>
<td>3.03</td>
<td>277</td>
<td>213</td>
</tr>
</tbody>
</table>

Figure 2: Experimental Results for Simulated Networks (100 Rule Batches)

Our results show that with batching every test performs more rule installs per second, with Tests 7 and 8 (corresponding to the more sparse rule graphs) performing dramatically better, while Tests 5 and 6 only improve relatively little. The performance difference between the two groups of tests is caused by the sizes of the SCCs. The cost of running the loop detection procedure is proportional to the size of the inserted SCCs regardless of whether we use batching or not, and the results in Figure 2 show that this has the effect of limiting the gains made by batch updates.

One thing to note is that the network can have undetected looping behaviour in between the batch updates which will not be detected, since we only test for loops in the rule graph as it exists when the batch is completed. We do not consider this a major drawback, as for looping behaviour to significantly degrade the performance of the network, it should permit the loops to remain for a relatively long period of time. At the end of every batch, we can then determine if precisely that graph configuration permits any packets to loop. Indeed, in practice, the batching could possibly be set to different sizes dynamically to respond to how busy the controller is.

We thus have that the performance gain with batching in Tests 7 and 8 is so large because of the fewer calls to Tarjan’s algorithm, combined with the expected lower size of SCCs in the rule graph. It is reasonable to expect that real world networks will have rule graphs with fairly small SCCs, and so we conclude that our algorithm, combined with batched flow rule updates, will permit commodity hardware SDN controllers to maintain an awareness of the network’s capacity to loop.

Although our model follows the OpenFlow specification [3] with respect to wildcard matches and rewriting rules, there are some operations that OpenFlow switches are capable of which our model does not capture, strictly speaking. We will describe these operations next.
4.1 Rule Priority

In OpenFlow, individual flow tables can be set up so that multiple entries all match some particular incoming packet. However, only one of these entries will be used: the one with highest priority. We model flow rules as all having equal priority, so that any action corresponding to an installed flow rule can be performed on incoming packets. This means that our loop detection routines will be somewhat more pessimistic than they need to be. A real OpenFlow switch with the first entry in its first flow table set to drop all packets will never be involved in a loop in the network. However, if that rule is dropped (due perhaps to timing out) the network can suddenly start to send some packets in a loop. By considering all flow rules to have equal priority, a diagnosis of loop-free by our algorithm guarantees that the network is free of loops under any sequence of flow rule evictions. Typically, this will be advantageous for a network administrator, since problems that might only show up in the network under relatively atypical conditions (for instance, unexpected network delays in updating high priority flow rules) will now be easily detectable. On the other hand, there exist sound, if complex, flow tables that will be incorrectly diagnosed as having the potential of looping by our algorithms.

4.2 Multiple Flow Tables

In the OpenFlow specification, packets are matched against a series of tables and can accumulate metadata and group memberships that are used to match the packet in later tables. In this work, we have ignored the possibility of multiple flow tables. The behaviour of OpenFlow switches with multiple flow tables can be much more complicated than those with a single table. Although for every collection of flow tables, there is an equivalent single flow table of the type we model, it may be that a single rule installation on a multiple-table switch can change the switch’s behaviour so much that no rule in the single-table model survives unchanged. In the event that a network uses multiple flow tables in a non-trivial way, our approach may become rapidly inefficient due to a large number of rules that may need to change in the model. Many networks may be efficiently modelled with our techniques, but for some multiple flow table management policies the efficiency of our approach will be significantly reduced.

4.3 Groups and Group Actions

In the OpenFlow specification, it becomes possible for incoming packets to become associated with groups as they pass through a sequence of flow tables which then cause the actions associated with those groups to be applied to the packet before it is output. We have no particular difficulty handling static group tables in our model. For every installed flow rule that associates some wildcard with a group, we can model that rule as instead performing the actions associated with the group. There are, however, difficulties that arise in our model when group actions change. As in the above section,
we have a situation where many rules may change their behaviour at once, resulting in updates being less efficient.

5 Conclusion

Considering the experimental results above, wherein the networks we tested have reasonable amortized performance, and the tested networks are significantly more difficult for our model to update than we expect real networks to be, we believe that the techniques discussed here have the potential to run on real world networks without significant overhead.

As our code is written in Python and does not make use of multiple cores, we expect that optimization and rewriting in a lower level compiled language should permit an implementation with orders of magnitude more throughput in rules per second than is seen here. Moreover, installation on much more powerful industrial strength server would also drastically increase the algorithms performance in practice.

An obvious next step would be the construction an application for some SDN operating system that permits modelling flow tables and detecting loops in running networks. Maintaining the rule graph and its strongly connected components has here been shown to be sufficiently light work that logging loops that occur in networks can be accomplished on a commodity hardware controller. An application that maintains the rule graph for a running network has the potential not just to detect possible looping behaviour, but also to discover many other characteristics of potential packet flow in a dynamically changing network. Indeed, another direction for future work could be to implement another one of the use cases of Header Space Analysis discussed in [1].

A Algorithms

A.1 DynSCC Formal Definition

In this appendix, we reproduce the dynamic strongly connected component algorithm from [5], which we call DynSCC. The definition is extended by the addition of three more operations for practical use, denoted insertV, removeV, getSCC. The following operations are supported.

insert\((E)\): Create a new, identical version of the graph \(G\) and add the edges \(E\) to it

remove\((E)\): Remove the set of edges \(E\) from all versions of the graph \(G\).

query\((u, v, i)\): Return 1 if \(u\) and \(v\) are in the same strongly connected component in the \(i\)th version of \(G\).

insertV\((v)\) Insert a vertex \(v\) into the graph.

removeV\((v)\): Remove a vertex \(v\) from the graph.
getSCC(u, i): Return the SCC containing the vertex u in version i of the graph.

We will use the following data structures and algorithmic primitives in the definition of the algorithm.

SCC(G): a subroutine which returns a list containing all of the strongly connected components of G (cf. Section 2).

parent: a 2n element array which holds, for each node in a forest with n leaves, a pointer to the nodes parent.

version: a 2n element array, where for all v ∈ V, version[v] = i if i is the first version of the graph G that the node v in the SCC forest appears in.

LCA(u, v): an algorithm which, when pre-processed on a forest (cf. parent), returns the least common ancestor of the nodes u and v in the forest (cf. Section 2).

preLCA(parent): an algorithm which pre-processes a forest, allowing LCA to run in O(1) time (cf. Section 2).

Union and Find: the primitive operations in a standard union-find algorithm (cf. Section 2).

---

**Algorithm 2: init**

\[
t = 0;\\nH_1 = \emptyset;\\n\text{foreach } v \in V \textbf{ do}\\n\quad \text{parent}[v] = \text{null};\\n\quad \text{version}[v] = \text{null};\\n\textbf{end}
\]

**Algorithm 3: insert(E')**

**Input**: An arbitrary set of edges, E'.

\[
t = t + 1;\\nH_t = H_t \cup E';\\n\text{findSCC}(H_t, t);\\n\text{shift}(H_t, H_{t+1});\\n\text{preLCA}(parent);
\]
Algorithm 4: remove($E'$)

**Input**: An arbitrary set of edges, $E'$.

**foreach** $v \in V$ do

\[
\text{parent}[v] = \text{null};
\]

end

**for** $i = 1, 2, \ldots, t$ do

\[
H_i = H_i \setminus E_i;
\]

findSCC($H_i, i$);

shift($H_i, H_{i+1}$);

end

$H_{t+1} = H_{t+1} \setminus E'$;

preLCA(parent);

Algorithm 5: query($u, v, i$)

**Input**: Two vertices $u, v \in V$ and a graph version $i$.

**Output**: 1 if and only if $u$ and $v$ are in the same SCC in $G_i$.

**return** $\text{version}[	ext{LCA}(u, v)] \leq i$

Algorithm 6: findSCC($H, i$)

**Input**: A set of edges $H$, and a graph version $i$.

$H' = \{(\text{Find}(u), \text{Find}(v)) \mid (u, v) \in H\}$;

$\mathcal{C} = \text{SCC}(H')$;

**foreach** $C = \{w_1, w_2, \ldots, w_{|C|}\} \in \mathcal{C}$ do

\[
\text{if } |C| > 1 \text{ then}
\]

\[
c = \text{NewNode};
\]

\[
\text{version}(c) = i;
\]

\[
\text{for } j = 1, \ldots, |C| \text{ do}
\]

\[
\text{if } j > 1 \text{ then}
\]

\[
\text{Union}(w_1, w_j);
\]

\[
\text{parent}[w_j] = c;
\]

end

end

end
Algorithm 7: shift($H_1, H_2$)

**Input:** Two sets of edges $H_1, H_2$.

**foreach** $(u, v) \in H_1$ do

- **if** $\text{Find}(u) \neq \text{Find}(v)$ **then**
  - $H_1 = H_1 \setminus \{(u, v)\}$
  - $H_2 = H_2 \cup \{(u, v)\}$

end

Algorithm 8: insertV($v$)

**Input:** A vertex $v$.

$V = V \cup \{v\}$;

Algorithm 9: removeV($v$)

**Input:** A vertex $v$.

Let $E_v$ be the set of edges incident to $v$;

remove($E_v$);

$V = V \setminus \{v\}$;

Algorithm 10: getSCC($u, i$)

**Input:** A vertex $u$ and a version number $i$.

**Output:** The SCC which contains $u$ in the graph $G_i$, if it exists.

$p = u$;

**while** True **do**

- **if** version($\text{parent}(u)$) $> i$ **then**
  - Break loop;
- $p = \text{parent}(p)$;

end

Let $L$ be set of descendants of $p$ which are leaves;

**return** $L$;
A.2 DynLoop Formal Definition

The algorithm DynLoop allows the following operations, where initLoop is assumed to be called first:

\text{initLoop}(N): Initialize a rule graph $G_N$ from a network description $N$, and an instance of DynSCC defined on $G_N$.

\text{insertRule}(r): Insert a rule $r$ into the rule graph $G_N$, and update the DynSCC algorithm, while logging any SCCs.

\text{removeRule}(r): Remove a rule $r$ from the rule graph $G_N$, and update the DynSCC algorithm.

\text{loopDetect}(G_N): Return true if there exists a loop which headers can travel on in the current state of the rule graph $G_N$.

What follows are the definitions of the operations. We note that \text{loopDetect} calls Algorithm 1 (PacketSimulation) as a subroutine.

\begin{algorithm}
\caption{loopDetect($G_N$)}
\begin{algorithmic}
\State \textbf{Input} : A rule graph $G_N$, and implicitly an instance of DynSCC defined on $G_N$.
\State Let $t$ be the current version of $G_N$;
\State \textbf{foreach} $r \in V_N$ \textbf{do}
\State \hspace{1em} \textbf{if} DynSCC.getSCC($r$, $t$) \neq \{r\} \textbf{ then}
\State \hspace{2em} $S$ = DynSCC.getSCC($r$, $t$);
\State \hspace{2em} Run PacketSimulation($S$, $\ast^L$, $r$) on $S$;
\State \hspace{2em} Log the existence of $S$, and also log if there exists a subset of header space which successfully loops in $S$ obtained from Algorithm 1;
\State \hspace{1em} \textbf{end}
\State \textbf{end}
\end{algorithmic}
\end{algorithm}
Algorithm 12: \text{initLoop}(N)

\textbf{Input} : A network \(N\) specified by a set of switches \(S\) and a list of rules \(L_s\) for each \(s \in S\)

Transform \(N\) into a rule graph \(G_N = (V_N, E_N)\) by the procedure given in Proposition 1;

\text{DynSCC.init();}

\textbf{foreach} \(r \in V_N\) \textbf{do}

\text{DynSCC.insertV(r);}

\text{end}

DynSCC.insert(\(E_N\));

loopDetect;

Algorithm 13: \text{insertRule}(r)

\textbf{Input} : A rule \(r = (m_r, f_r)\) not in the current version of \(G_N\).

DynSCC.insertV(r);

\(E_r = \emptyset\);

\textbf{foreach} \(r' = (m_{r'}, f_{r'}) \in V_N\) \textbf{do}

\textbf{if} \(f(m_r)|m_r \cap m_{r'} \neq \emptyset\) \textbf{then}

\text{Add \((r, r')\) to \(E_r\);}

\textbf{end}

\textbf{if} \(f(m_{r'})|m_r \cap m_{r'} \neq \emptyset\) \textbf{then}

\text{Add \((r', r)\) to \(E_r\);}

\textbf{end}

DynSCC.insert(\(E_r\));

loopDetect;

Algorithm 14: \text{removeRule}(r)

\textbf{Input} : A rule \(r = (m_r, f_r)\) not in the current version of \(G_N\).

Let \(E_r\) be the set of edges incident to \(r\);

DynSCC.remove(\(E_r\));

DynSCC.removeV(r);
References


