AMERICAN UNIVERSITY OF BEIRUT

NETWORK DATA PLANE VERIFICATION: A TRADEOFF BETWEEN PROBABILITY OF CORRECTNESS AND COMPUTATIONAL COST

by
IBRAHIM ABDELGHANY

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Maroun Semaan Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
January 2020

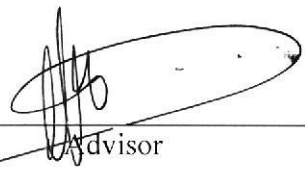
AMERICAN UNIVERSITY OF BEIRUT

NETWORK DATA PLANE VERIFICATION: A TRADEOFF BETWEEN PROBABILITY OF CORRECTNESS AND COMPUTATIONAL COST

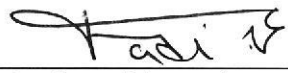
by

IBRAHIM ABDELGHANY

Approved by:



Dr. Imad H. Elhajj, Professor — Advisor
Department of Electrical & Computer Engineering



Dr. Fadi Zaraket, Associate Professor — Member of Committee
Department of Electrical & Computer Engineering

Dr. Wassim Masri, Associate Professor — Member of Committee
Department of Electrical & Computer Engineering

Date of thesis/~~dissertation~~ defense: January 27, 2020

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name:

Abdelghany	Ibrahim	Hassan
Last	First	Middle

☑ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☑ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes
after:

One ---- year from the date of submission of my thesis, dissertation, or project.

Two ---- years from the date of submission of my thesis, dissertation, or project.

Three ---- years from the date of submission of my thesis, dissertation, or project.

Ibrahim Feb 19 2020

Signature Date

ACKNOWLEDGMENTS

First, I would like to thank my parents for their love and support during my academic studies. I would also like express my gratitude to my advisor Prof. Imad Elhajj for his invaluable guidance and comments throughout my studies and my thesis. I also appreciate the comments and the feedback of the rest of the members of my thesis committee Prof. Fadi Zaraket and Prof. Wassim Masri.

I would like to thank my colleagues at SAUGO 360 for their continuous support and motivation. I also would like to thank my friends who were immensely supportive during my master studies.

AN ABSTRACT OF THE THESIS OF

Ibrahim Abdelghany for Master of Engineering
Major: Electrical and Computer Engineering

Title: Network Data Plane Verification: A Tradeoff Between Probability of Correctness and Computational Cost

Traditional and SDN Networks are increasingly more complex and covering more use-cases making reasoning about network behavior all-the-more challenging. Dedicated tools to verify networks for reachability and other invariants exist, but not without scalability limitations as these problems are combinatorially complex. Workarounds that exploit regularities to minimize processing exist, but they depend on data-plane properties that are not guaranteed to exist in SDN networks, especially as more use-cases are being applied with packet headers used in unconventional ways. We propose a tradeoff between the probability of correctness, based on network traffic statistics, and the verification computational cost. Such a tool gives operators the flexibility and freedom to select their own preference in this tradeoff, while making feasible a partial solution of cases that require exponential or factorial time. We represent the network as a Markov Chain, and propose a prioritized traversal algorithm to verify reachability questions. We test our algorithm on randomly generated networks of varying complexities and traffic distributions, proving the effectiveness of our method for high-complexity networks and the efficacy of our traversal algorithm in taking advantage of skewness in traffic weights. We were able to achieve 99% traffic path probability coverage in 2.45% (and 95% traffic path probability coverage in 0.57%) of the time needed for full coverage on randomly-generated test networks.

CONTENTS

TABLES

FIGURES

CHAPTER I
INTRODUCTION

Network monitoring, configuration, and management tools are an indispensable part of operating modern networks. Such tools provide a single pane view where network statistics are collected, device configuration is stored, and network-wide services are managed. Software-Defined Networking (SDN) is built around the paradigm of centralized network management as it separates the control-plane, which is composed of routing configuration and algorithms, from the data-plane, which is responsible for low-level packet forwarding operations. Traditional network operators employ ad-hoc solutions that interface with legacy devices and protocols to provide centralized network management features.

One critical feature of network management is the ability to verify network-wide properties from low-level device configuration. This allows operators to safely push new changes without breaking network policies, and to verify that networks will behave as expected for arbitrary inputs and conditions.

Complex networks with many devices each containing many forwarding rules are not easily verifiable. Traditional networks are composed of many protocols and ad-hoc solutions, making reasoning about their behavior and correctness a daunting task. Moreover, SDN has exposed packet headers to arbitrary matching and rewriting, accelerating the number and scope of possible services a network could run. Asking questions about forwarding behavior needs a framework where forwarding rules are translated into formal statements. Afterwards, these formal statements can be verified for specific criteria of correctness.

One possible framework, among others, for formal verification is SAT-based solvers. The input for such frameworks is a set of Boolean equations representing the behavior of the system. A generalization of SAT-based verification is Satisfiability Modulo Theories (SMT) which introduce support for functions, numbers, bit-vectors, and more complex data-types. Alternatively set-based methods rely on sets to represent packets and on functions on sets to represent forwarding logic.

Representing a network node data-plane in terms of bitvector SAT formulas, or sets and functions, is fairly straightforward. Moreover, encoding the network topology is also straightforward. Using various techniques, verification tools convert reachability questions into tests on network models. The worst-case scenario for such tests is in the order of exponential, even factorial, in terms of the number of nodes and links, the number of forwarding rules, and the header size. This possible combinatorial explosion has attracted much research focused on optimizations for cases where networks satisfy certain properties [2, 3, 4, 5, 6, 7, 8, 9, 10].

Current techniques for avoiding combinatorial scalability issues rely on the fact that packet header representations can be compressed to minimize unnecessary computations. This relies on the fact that the number of header classes (headers that are treated the same way across the network) is relatively small compared to the worst case. Traditional networks consisting of routers and switches heavily depend on range-matching and prefix-matching operations when dealing with packets, limiting the number of header classes needed for verification. SDN networks, however, match packet headers with arbitrary wildcard masks on a large set of fields making the problem complexity closer to the worst-case. Verification complexity is also affected by topology complexity which increases as overlay networks become more widespread. As SDN networks

implement more complex use-cases and treat packet headers with more freedom, the number of header classes is bound to increase.

In this document, we propose a remedy to the problem of combinatorial explosion (for cases where current optimizations fail to reduce running time) by using network traffic statistics to create a tradeoff between computational cost and probability of correctness, based on a Markov Chain representation of the network graph. The proposed solution is also applicable to general cases by introducing flexibility in allowing users to choose the degree of correctness to save on running time.

We use a Markov Chain to represent the graph of nodes in the network, where probabilities are extracted from traffic statistics. Then we take as input a probability threshold that the user selects and traverse the network graph to verify paths covered within the threshold. We minimize running time, under the selected threshold, by traversing the highest probability paths first.

We use the term ‘coverage’ to refer to the cumulative probability of paths verified while traversing the network graph. Full coverage would result in a cumulative probability of paths covered equal to 1 with no more search needed for further paths in the network, while partial coverage under a specific probability threshold will only ensure that paths with probabilities summing up to it are considered in the search.

Our current solution relies heavily on ideas proposed in Anteater [1], however, our reachability algorithm applies more naturally to HSA [3] as path enumeration is part of their method. Moreover, HSA answers reachability questions with a list of packet headers that are able to traverse the network instead of only yes/no answers and single counter-examples as in [1].

We tested our algorithm against randomly generated networks and validated the time-saving power for different network complexities and traffic distributions. We outperform HSA for probability thresholds less than 1, and lag behind it for full traffic coverage due to the extra cost of prioritized traversal. We were able to achieve 99% traffic path probability coverage in 2.45% (and 95% traffic path probability coverage in 0.57%) of the time needed for full coverage by HSA on randomly generated test networks. The remaining uncovered traffic path probabilities are paths in the network that have probabilities less than what is needed to cross the selected probability threshold. Our algorithm performs better for more complex networks and more skewed traffic distributions.

This document consists of the following chapters: Chapter I introduces the problem, highlights the proposed solution, and summarizes the results. Chapter II surveys the literature on network verification and identifies a gap that we intend to fill. Chapter III defines the reachability verification problem formally and motivates the need for our solution in light of trends in modern networks. The proposed solution is explored thoroughly in Chapter IV, we define our modelling approaches and present our prioritized traversal algorithm. Our testing results are discussed in Chapter V. Finally, we conclude and list future directions for research in Chapter VI.

CHAPTER II
LITERATURE REVIEW

Network verification can be classified into two broad categories: data-plane and control-plane verification.

A. Control Plane Verification

Control-plane verification attempts to represent the forwarding decisions of control- plane protocols (such as BGP, OSPF, ..) and their interaction across a network. Verifying the control-plane entails modelling distributed routing protocols (and their interaction) to be tested for desirable properties operators specify (such as reachability, loop detection, black holes, etc.) which could involve complexity in modelling how a distributed protocol runs in a network in addition to the interaction of multiple routing protocols. Moreover, control-plane verification does not have the ability to detect bugs in router software causing undesirable or incorrect forwarding entries to exist in routers' forwarding tables.

In [11], authors present Batfish, a tool for generating a data-plane snapshot from configuration files and environment specifications as 'what-if' scenarios. The generated snapshot is then passed to a data-plane verification tool to check for specific invariants, while keeping a mapping between specific configuration statements and data-plane rules associated with undesired behaviors. Unlike [11], Minesweeper [2] models the stable-state control-plane itself using logical formulas passed to constraint solvers to be tested for network properties such as reachability, loops, blackholes, waypointing, etc. It doesn't generate a data-plane corresponding to input configuration, but rather models the

configurations themselves. Similar to [2], [12] ERA models the control-plane directly which could consist of multiple interacting protocols and checks for reachability given a set of environment specifications. Both Minesweeper and ERA employ a range of optimizations to minimize computation and limit IP Address matching to prefix matching.

1. SDN Control Plane Verification

With SDN's centralized control plane, formally verifiable languages for SDN controllers have been proposed across the literature. The general approach taken is to create a language that is both expressive and amenable to formal verification methods.

FlowLog [13] is declarative finite-state language for SDN controllers attempting to mitigate the complexity inherent in reasoning about general-purpose fully-expressive programming languages generally used. It also allows the reuse of code written in imperative languages as black boxes to make it convenient for integration with existing solutions. Similarly, NetKAT [14] is a language for programming and reasoning about networks based on Kleene Algebra with Tests (KAT) [15]. KAT provides the ability to test equivalences between programs and enables reasoning about local and global switch processing. Network programs in NetKat are functions of packet histories that return sets of packet histories. Kleene operations allow policies to be composed via union, in sequence, or using an arbitrary combination of both. NetKAT was extended in other research papers to include probabilistic predicates [16] to allow for reasoning about networking programs with stochastic elements. Moreover, in [17], event handling was added along with extending the language with mutable states.

Alternative approaches exist, such as in [18], where the authors propose methods for dynamic checking of network properties as the controller reacts to changes by allowing developers to insert assertion statements in general purpose SDN controller languages.

B. Data Plane Verification

Data-plane verification attempts to model the network data-plane as it exists in the router's forwarding tables. This entails modelling the low-level decisions the router makes in handling packets independently from their origin in control-plane algorithms and protocols. Moreover, data-plane verification allows modelling networks of very diverse device-types irrespective of high-level routing protocols that guide their behavior. It therefore applies to traditional networks by pulling their FIBs (forwarding tables) or SDN networks by extracting their forwarding tables (OpenFlow).

In [1] authors propose Anteater, a tool for data-plane verification based on transforming high-level network data-plane configuration and questions about reachability into bit-vector logical expressions passed to a SAT solver. Alternatively, HSA [3] models packets at ports as points (and cubes) in a header space with switches being transfer functions that map one point to a set of points in that space. Reachability questions and checks for certain network properties are converted, consequently, into a series of transformations applied to a header space according to paths in the network topology.

Following on work in [1] and [3], authors in [19] and [20] propose a SAT-based method for verifying network properties. It models the network using propositional logic, to which constraints are added as queries about specific properties. It provides a uniform

network modelling framework capable of representing arbitrary packet matches and rewrites in propositional logic. It doesn't keep packet histories as spatial and temporal information are encoded simultaneously by representing a single packet trace throughout the network. It employs a graph-based representation of the network, which avoids possible expansions in representations based on paths, but denies the ability to deal with multicasting and non-vicious loops.

Authors in [4] propose a general specification language called Network-optimized Datatlog (NoD) for verification of networks. Both network models and queries for invariants are written in Datalog allowing flexibility in modelling a wide variety of networks. It supports packet rewrites and the ability to define new packet types without the need to modify NoD internals.

All above approaches applied naively would result in very expensive computational procedures and impracticalities in applying them. Tools and approaches, therefore, employ a set of optimizations to make it possible for actual network operators to run tests and generate useful results in reasonable time frames.

C. Optimizations for Network Verification

Verifying network properties such as reachability and loop-freedom is an expensive computational problem. [1] proved that the problem of verifying reachability is at least NP-complete by showing that the Boolean Satisfiability problem (3-SAT) can be reduced to a dataplane reachability problem. Thus, most tools resort to SAT solvers (or equivalent model checking tools) which apply heuristics and greedy methods to solve these problems efficiently. Moreover, even SAT solvers struggle with the complexity of network verification due to the large state space of packet headers, in addition to network

topology complexity. Packet headers with **n** bit create a search space of size **2^n** possible options to test against forwarding rules across paths in the network. Minimizing the search space has received a lot of attention in the literature.

Methods to minimize the search space revolve around compressing headers into compact representations that can be efficiently verified for desired network properties. In [2] where matching operations were limited to IP address prefixes, proposes a prefix-hoisting method to minimize the number of variables needed to represent IP-addresses by representing them as integers and using the more efficient integer difference logic (IDL) for prefix matching. HSA [3] employs a set of optimizations to improve its performance such as 'IP table compression' to minimize the number of transfer function rules, in addition to concept called 'lazy subtraction' which avoids unnecessary expansion of header space information as the algorithm progresses. Similarly, [4] overcomes scaling issues by using symbolic header representations (using BDDs) and difference of cubes (inspired by [3]) to minimize an extended representation along with Datalog optimizations specific to operations on header input-output relations.

Authors in [5] and [6] propose the use of atomic predicates as primitives corresponding

to equivalence classes of packet headers to speed up reachability computation by allowing the use of sets of integers to represent packet matches. Similarly, Veriflow [7], in its attempt to run as an online verification system for SDN data-planes, achieves low latency by grouping headers into Equivalency Classes (ECs) which follow the same forwarding behavior throughout the network. A graph per-EC is constructed to which flow updates specific to these ECs are applied after which a graph verification algorithm to check for

custom invariants is run. Moreover, [8] takes a slightly different approach by exploiting forwarding similarities in parts of the network, rather than doing so globally.

Authors in [9] and [10] observe that many tools have faster results than the worse-case combinatorially complex running time and provide a theoretical grounding for these results. They define atoms as classes of headers that follow the same behavior across the network and proceeds to provide a polynomial time (in the number of header classes) algorithm for loop detection. While many practical cases can be solved efficiently, the number of header classes could nonetheless be exponential.

These methods achieve good results in practice as long as the number of header classes is polynomial. However, the problem remains combinatorially complex for cases when the number of header classes is not polynomial.

Our proposed method can use header-compression techniques when relevant to minimize the search space as the input allows, but nonetheless for cases when compression does not yield considerable reduction in complexity (ex. arbitrary wildcard matching in SDN), it creates a tradeoff for further minimizing the search space by allowing for a tradeoff between search space covered and probability of correctness by exploiting statistical properties of actual network traffic. Moreover, in creating a tradeoff, it gives users the ability to define their probability of correctness versus time-saving preference where to the best of our knowledge no existing method provides such flexibility.

D. Comparison with Our Method

Table 1 compares methods from the literature with our approach. Our method's most important contribution is the ability to make probabilistic approximations through prioritized traversal under a probability threshold. Moreover, our method supports loop tolerance and the option to terminate reachability traversal at looping branches, making this feature configurable. We share key optimizations and properties with HSA: (1) We provide sets of headers, along with packet histories, as output to reachability questions, while other methods return yes/no answers and counter-examples instead of full reachability sets; (2) We use header space models and transformations as a base for our implementation which allows us to exploit HSA optimizations such as Lazy Subtraction; (3) We support abstract forwarding functions which process and manipulate headers, regardless of the underlying implementation.

Method	Possible Inputs	Answers	Loop Tolerance	Solver types	Optimizations	Approximations
Anteater	Match -> Rewrite -> Forward	yes/no; ***With counter-examples***	Tolerates Loops; ***Limitation: max path length fixed***	SAT/SMT Solver	SAT solver techniques	ø
HSA	Match -> Rewrite -> Forward; ***Abstract forwarding functions***	Sets of Reachable Headers	Stops at repeated port	Header Space Transformations	Lazy Subtraction	ø
[19-20]	Match -> Rewrite -> Forward	yes/no	Stops at repeated node	SAT/SMT Solver	SAT solver techniques	ø
NoD	Match -> Rewrite -> Forward; ***DataLog Models***	Sets of objects	Tolerates Loops; ***Model-specific interventions needed to enable/disable***	DataLog	Difference of Cubes (eq. to lazy subtraction)	ø
Our Method	Match -> rewrite -> forward; ***Abstract forwarding functions***	Sets of Reachable Headers	Configurable	Header Space Transformations	Lazy Subtraction	Prioritized Traversal

Table 1: Comparison of different methods

Next we define the problem of data-plane verification and discuss its complexity in light of recent trends and use-cases in modern networks.

CHAPTER III
PROBLEM DEFINITION

A. Verify Reachability in a Large Network

Consider a network with many devices each consisting of data-plane forwarding rules of the format:

match *header* ***forward*** *port*

Forwarding rules define how routing should take place in a network device. Each rule consists of a match section, which defines the packets this rule is applied to, and an action section specifying how to treat the matched packet (forward, drop, rewrite, …). Forwarding rules are grouped in routing tables in which they are matched according to priority where the highest-priority rule matching a packet is used to forward the packet. More complex forwarding table organizations exist, but for practical reasons, the treatment of forwarding tables will be restricted to the scheme just described.

Question: *How can we know if a* ***packet <p>*** *can go from* ***device <s>*** *to* ***device <t>****?*

Network reachability questions are tests to identify if certain traffic (defined by packet wildcards) can be forwarded from one network endpoint to another. Reachability tests can also be used as building blocks for more advanced network verification questions such as: network isolation, forwarding loops, and forwarding blackholes as shown in Anteater[1].

The input packet defines one or more header wildcards which represent input traffic to be tested. The input packet can be alternatively defined as a symbolic header

with constraints on its values. Source and target devices/ports are network endpoints that we apply the test to.

We need to use **formal verification** to answer such questions. Formal verification is the application of formal proofs on mathematical models of systems of interest. Formal verification allows for proving or disproving whether a modelled system satisfies a property of interest, usually related to specifications of correctness.

To apply formal verification to the reachability problem, we need to formally model the network's forwarding dataplane as well as the input to our reachability test. Multiple modelling approaches exist, each suited for certain types of tools/methods for answering forwarding questions.

After modelling network and the reachability test input, we feed the test to a theorem prover. Theorem provers (SAT solvers, reasoning engines, ..) are tools and methods which accept formal models and answer questions about the model's properties. Theorem provers come in different flavors but all rely on formal methods (symbolic logic, set theory, graph theory, …) in order to reason through the provided input to identify whether the property in question is satisfied.

B. Problem Complexity

The problem of verifying dataplane reachability in networks is combinatorially complex and at least NP-Complete as proven in [1]. The actual complexity of the problem of s-t graph reachability with boolean filters is P-SPACE Complete as proven in [27]. This means we can't find the correct answer for all possible inputs due to possible combinatorial explosion in running-time. We need heuristics, or we need to be

economical in the way to check for cases. SAT-solvers and theorem provers already do that.

To solve a verification problem, you need to cover all possible cases (inputs) and validate that the result is as expected. SAT-solvers and similar tools try to minimize the checks needed to solve a problem by applying a combination of greedy techniques and heuristics, which exploit regularities and patterns that make most practical problems much easier than the worst-case. Tools and frameworks that are not SAT-based, such as Header Space Analysis, also apply a similar flavor of the above tricks to minimize the scope and number of cases to be checked.

The bag of tricks developed by the formal verification community does a good job as long as the actual complexity of the problem is less than the worst-case. At worst-case they just give the best approximate answer they can in the time window given.

Network verification complexity is a function of the complexity of the input needed for the property being checked for. In a network reachability verification problem, the number and complexity of forwarding rules on devices, the size and complexity of network graphs, and the size of packet headers contribute to the overall complexity of the verification problem.

Forwarding rule complexity is a function of the number of bits being checked for in a match operation, the number of bits being rewritten, and the relative uniqueness of the match/rewrite operation compared to other rules in the network. Network graph complexity is directly related to the size and density of the graph, which increases the number of possible paths taken by a packet. Header complexity is directly related to the number of bits in the header.

Network dataplanes can become too complex when forwarding devices apply free-form header matching operations instead of abiding by legacy header fields and matching operations. This happens when more network devices apply wildcard header matching instead of prefix-matching and range matching in addition to manipulating header fields in non-standard ways (by re-using fields for unintended functions). Free usage of packet headers significantly contributes to dataplane complexity. Case in point are SDN networks that expose packets to full wildcard matching and rewriting.

Network graphs can become too complex as more and more overlay networks are deployed in arbitrary topologies which are not limited to the physical ports existing on a network device. Free overlay networks are enabled by SDN and similar technologies.

Packet headers become too complex as new protocols substantially increase header size. IPv6 has significantly larger header sizes than IPv4, moreover, MPLS can vary in size and grow as their applications evolve.

C. Discussion

For large and complex problems, SAT solvers and similar tools can only give approximate answers or no answer at all. Solvers apply computational tricks to save on running time, but end up giving approximate answers based on general-purpose greedy approaches. It would be of value to make use of traffic statistics to create meaningful metrics to be used in reaching an approximate answer. This will be the topic of the next section.

CHAPTER IV
PROPOSED SOLUTION

For complex networks where it is infeasible to get a full answer to the dataplane reachability problem, we will make use of traffic statistics to prioritize checking and save on running time. It would be even better if we can use meaningful probability metrics to quantify the quality of the approximation, and even better than that would be using a probability threshold as an input to a tradeoff between probability of correctness (based on traffic coverage) and computational cost.

We propose a tradeoff between probability of correctness and computational cost and provide an algorithm that takes a probability threshold as an input and minimizes running time under that threshold.

It is our intuition that as the user varies the selected probability threshold from 1 downwards, the running time of the algorithm will drastically improve as shown in the below conceptual figure (Figure 1). We defend this intuition in the next section.

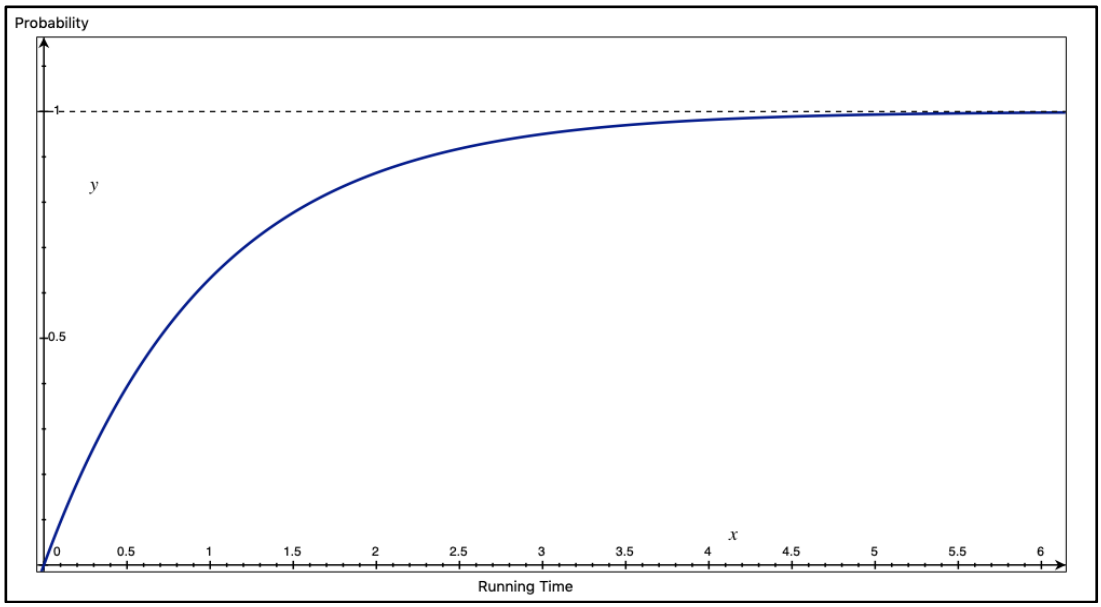


Figure 1: Conceptual plot of convergence of cumulative probability

A. Tradeoff Intuition

Our intuition is that traffic statistics follow a statistical pattern best described using Ziph's Law. Ziph's law is an empirical statistical law describing properties found in social and physical systems, such as population sizes of cities, word frequencies in corpuses, etc... It describes a relationship between frequency and rank where the frequency of an item is inversely proportional to its frequency (raised to a certain power). The law is sometimes popularly referred to as the 80-20 rule where 20% of elements in a population contribute to 80% of its weights. We suspect that path probabilities in the network graph are Ziph distributed and will inherit skewness properties from the Ziph's Law.

Our intuition relies on four important observations. First, internet usage patterns are not evenly spread over the available options. Very few websites/services consume most of the internet traffic (Netflix 15%, YouTube 11%, …) [23]. This means very few traffic streams consume most of the bandwidth. Second, routers group traffic streams using wide packet matches such as IP Prefixes, wildcards, ranges, or other compression mechanisms to minimize forwarding rules. These techniques group diverse traffic flows into limited paths in the network, making few paths correspond to many underlying traffic flows, therefore, aggregating their weights. Third, routers treat similar traffic similarly across the network. This ensures that aggregations of traffic streams does not differ from router to router, preserving groups of traffic streams across the network. Fourth, similar network traffic tends to be processed by few nodes as networks make use of forwarding trunks and hierarchical tree routing schemas, which minimizes the total number of forwarding rules in the network among other objectives.

This allows the covered probability in a greedy traversal algorithm to quickly converge to 1, as a minority of the sample space carries most of the traffic. As networks become more complex, there will still be statistical regularities. We want to exploit them to save on computation and utilize the tradeoff. Next, we discuss how we model traffic statistics.

B. Modelling Network Traffic as a Markov Chain

Representing the network as a Markov Chain allows us to simplify our analysis as we only need to consider probabilities at each node/port, which overlaps with statistics collected by devices.

Markov Chains are memoryless, meaning that

$$P(event \mid history) = P(event \mid present)$$

This property greatly reduces the state space as we only consider transition probabilities between nodes when constructing path probabilities.

A Markov Chain can be represented by a set of traffic states $\{s_1, \ldots, s_n\}$, corresponding to network nodes and ports, and a transition matrix $P_{n \times n}$, corresponding to traffic statistics, where:

$$p_{i,j} = \textit{probability of transition from state } s_i \textit{ to state } s_j$$

We map each node (and port) in the network (graph G) to a state in the Markov Chain, in addition to special states such as s_{drop} being one of them. Traffic statistics from ports will be turned into transition probabilities by dividing traffic volume on an outgoing port by overall node outgoing traffic volume. For the sake of simplicity we stick to mapping the traffic volume ratios as-is to transition probabilities. Other more advanced mechanisms can be used to ensure no existing network link has a transition probability of

0 by introducing a baseline probability of randomly choosing an arbitrary link at random. This baseline probability is referred to as “damping factor” in PageRank [22].

Increasing the order of the Markov Chain increases accuracy of the model at the expense of an expansion in the state space. This can be done by adding one element of history: where $P_{i,j,k}$represents transition probabilities from states i to j to k.

C. Modelling Network Forwarding using Header Space Transfer Functions

We will use HSA [3] as a base for our solution. HSA has many attractive qualities that make it a suitable framework to use since: (1) It is flexible in supporting arbitrary packet headers. (2) It is capable of representing match/rewrite/forward operations. (3) It transparently processes the network graph and packet headers, lending itself to network-specific manipulation. (4) It contains optimizations to limit the search space by exploiting forwarding rule regularities. (5) It supports abstract forwarding functions independent from specific device implementation details.

HSA represents headers as points in an n-dimensional space where all header bits are treated equally with no assumptions and restrictions to their use. Moreover, hypercubes in this header space represent sets of packets with wildcards used for certain bits. Hypercubes allow for symbolic processing of packets in that sets of headers with certain properties can be processed instead of particular concrete headers. Network ports are also modelled as points in space.

In HSA network devices are transfer functions that map a pair of port and header space object into a set of ports and header space object. This represents how an input packet at a source port is forwarded into multiple ports with potentially modified headers. The transfer functions map one input into multiple potential forwarding decisions (ports

& header) which is necessary to deal with hypercubes (symbolic packets, wildcard headers). Such transfer functions also support multicasting in addition to header manipulation and rewrites.

*Transfer Function: (header, port) → *(header, *port)*

The topology is also represented by the same transfer function from ports to ports, irrespective of header values, according to the links present in the network.

To verify reachability between two points in the network, HSA applies the transfer function at the source port and the input header and keeps applying the transfer function to the resulting ports and headers until the destination is reached or a loop is detected.

HSA uses header compression techniques to minimize expansion in header processing by using difference of cubes or lazy subtraction instead of expanding negation operations into flat unions until it is unavoidable. These techniques exploit processing regularities across the network to minimize the number of checks/tests in verification. Such techniques were given theoretical grounding in [9] and [10] by introducing the concept of equivalency classes of similarly treated sets of headers.

D. Probabilistic Traversal Algorithm

The probabilistic traversal algorithm verifies the reachability of an input packet header between a source port and a set of destination ports. It returns a set of reachable packet headers at the destination ports along with their histories (headers and ports during forwarding in the network).

The probabilistic traversal algorithm verifies reachability in a network according to an input probability threshold which picks a point in the tradeoff between correctness and computational cost. It minimizes the computational cost under that threshold by

selecting the largest probability subpaths while traversing the network graph according to forwarding logic.

The probabilistic traversal algorithm is run against a network model consisting of a network transfer function and a network Markov Chain. The network transfer function models the network forwarding logic as a map from a header object at a port to a set of header objects each at a list of ports. The network Markov Chain is constructed from traffic statistics at nodes and ports in the network. It is represented as a transition matrix where each element is the probability of going from node/port at row to node/port at column.

Overall signature of the probabilistic traversal algorithm:

Input:
- Input header
- Source port
- Destination ports
- Probability threshold
- Network Model:
  - Network Transfer Function:
    - ***(header,port) → * (header,*port)***
  - Network Markov Chain:
    - P_{nxn}

Output:
- List of largest probability paths covered by the probability threshold that trace the input packet from input port to output ports

1. Core Transformation Step

We will identify a core transformation step which we will run recursively on the network model until we reach a stopping condition. The core transformation step takes a header at a port as input and transforms them into a set of header at port with probability. This transformation is done (1) by using the network transfer function to generate the

next port-header tuples and (2) by using the network markov chain to get the transition probability from the input port to the output port. (Figure 2)

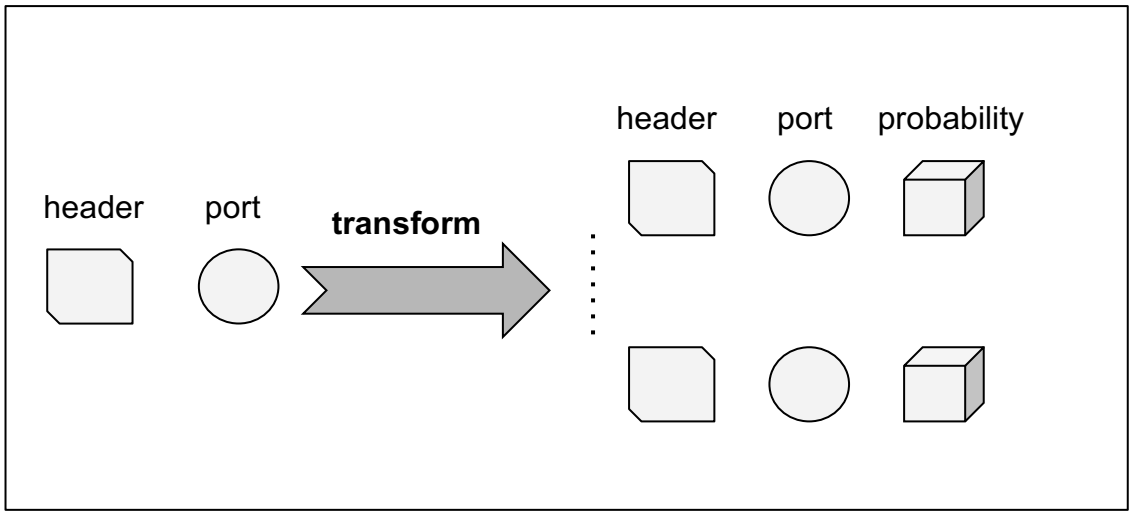


Figure 2: Core Transformation Step

The algorithm needs to use this core transformation step as building block in a way that prioritizes traversal of largest probability subpaths. Moreover, it needs to keep track of a *cumulative probability* to be used as a stopping condition when it is equal or more than the input probability threshold.

In the next sections we will address two concerns: (1) how to calculate the *cumulative probability* at every step in order to identify the stopping condition and (2) how to prioritize traversal by choosing highest probability first.

2. Calculating the Cumulative Probability

The cumulative probability is the sum of the probabilities of terminated sub-paths discovered while traversing the network model. It is used, along with the probability threshold, in the stopping condition in the algorithm.

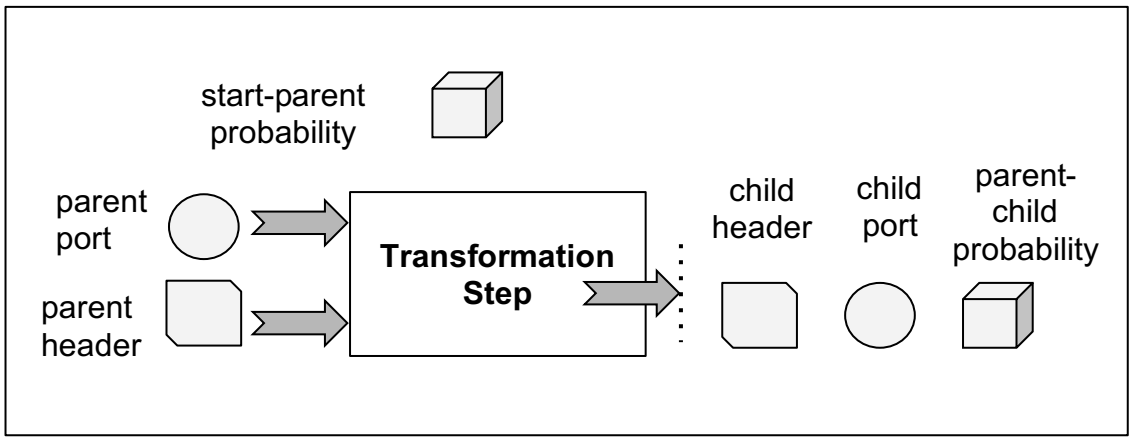


Figure 3: Core Transformation Step & Parent Probability

Each node being traversed has a start-to-node probability which is the path probability from the source node to the current node. In Markov Chains, path probabilities are the product of transition probabilities between the nodes along the path. Initially, the source node has a start-to-node probability of 1, and as the branching traversal of the network model takes place, each step from one node to another multiplies the parent's start-to-node probability with the parent-child transition probability (generated by the core transformation step) to generate the current node's start-to-node probability.

Cumulative probability is incremented when a *terminated node* is reached during traversal. Terminated nodes are either *destination nodes* or *unreachable nodes*. When a destination node is reached we increment the cumulative probability with the node's start-to-node probability which is the start-parent probability multiplied with the parent-child probability returned in the core transformation step. The core transformation step does not explicitly return unreachable nodes, they must be inferred by comparing the output with network model. The core transformation step returns the reachable ports with their corresponding headers and parent-child Markov Chain transition probability (Figure 3). The probability of a step transition (sum of all one-step transitions) in a Markov Chain is **1**. The sum of unreachable parent-child probabilities is therefore $\mathbf{1} - \sum(\boldsymbol{parent\text{-}child}$

probability; for all reachable nodes). The cumulative probability is then incremented with the unreachable parent-child probabilities multiplied by the parent's start-to-node probability.

3. Prioritizing Traversal

For the algorithm to minimize the number of checks under the selected probability threshold, we will prioritize our traversal to cover the largest probabilities first. Every node during traversal has a start-to-node probability. This probability is the maximum probability coverage that can result from traversing recursively through the node until stopping at all destinations or unreachable nodes. This probability is the priority metric which will be used to select which node to process next in the traversal.

This can be achieved by using a priority queue to keep intermediate nodes during traversal and to pop highest priority (probability) nodes. (Figure 4)

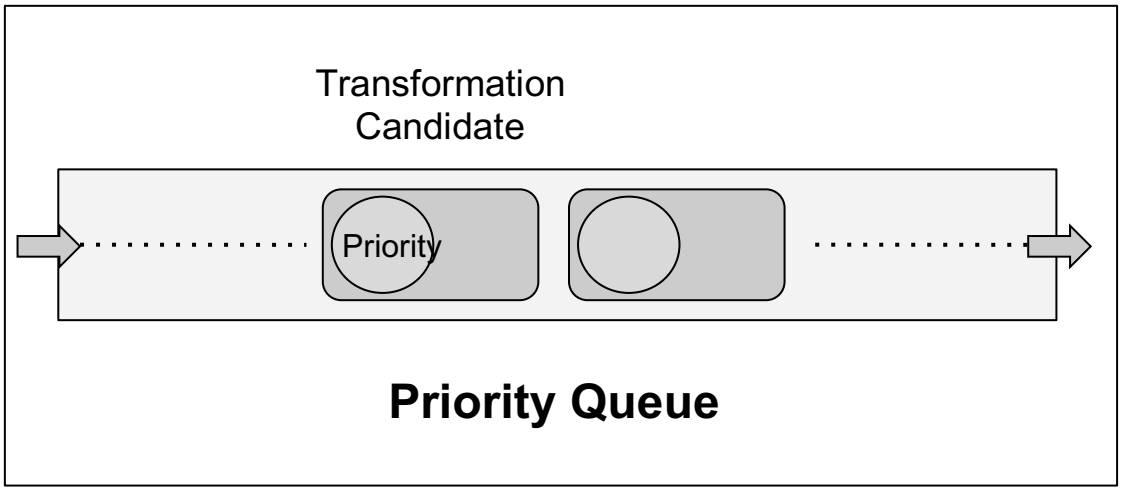


Figure 4: Transformation candidates in the priority queue

4. Big Picture

Starting with the source node as the initial transformation candidate with path probability equal to **1**, we use a priority queue to prioritize traversal according to node path probability. A while loop iterates until either the queue is empty or the cumulative probability reaches the input threshold (Figure 5). At every step in the while loop the algorithm:

1) Pops a transformation candidate from the priority queue
2) Applies the transformation step
3) Updates cumulative probability
4) Inserts child nodes as transformation candidates where the priority for every child is the start-to-child probability

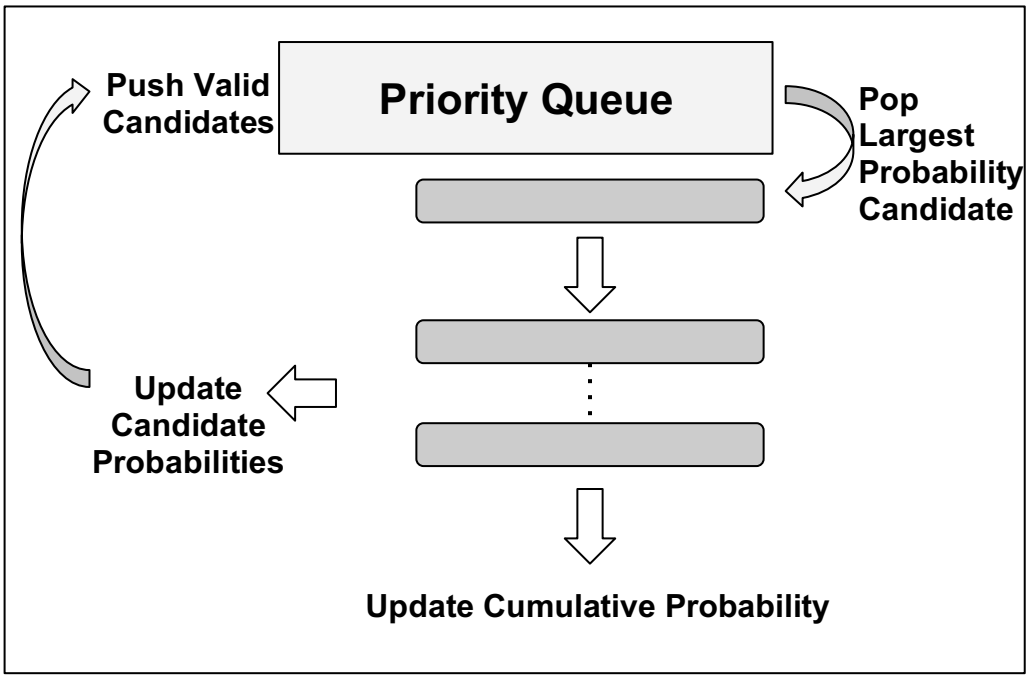


Figure 5: Overall sketch of the algorithm

5. Pseudocode

```
def verify_reachability(
    network,
    input_port,
    input_header,
    destinations,
    prob_thresh):

  pq = PriorityQueue()

  pq.push(1, {
    port: input_port,
    header: input_header,
    history: []
  })

  cum_prob = 0

  while cum_prob <= prob_thresh:
    prob, next = pq.pop()

    hops, leftover_prob = transform(network, next)

    cum_prob += leftover_prob

    for hop in hops:
      path_prob = hop.prob * prob

      if hop.port in destinations:
        cum_prob += hop_prob

        reachable_paths.append(hop, path_prob)

      else:
        pq.push(path_prob, {
          port: hop.port,
          header:hop.header,
          history:hop.history
        })

    return reachable_paths
```

```
def transform(
  network,
  port,
  header):

  ntf = network.transfer_function
  nmc = network.markov_chain

  next_hops = ntf.tx(port, header)

  leftover_prob = 1

  for hop in next_hops:
    hop.prob = nmc[port, hop.port]
    leftover_prob -= hop.prob_thresh

  return next_hops, leftover_prob
```

6. Example

We consider a simple example to illustrate how our prioritized traversal algorithm solves the reachability problem. Consider the network shown in Figure 6 consisting of 3 nodes in a full mesh topology with a header of length 3 bits. The forwarding logic of the network is given in Table 2 detailing the packet matches needed to forward packet between any two nodes as packet wildcards where an unset bit (don't care) is marked by 'x' while set bits are give values '0' or '1'. Table 3 shows the probability matrix of the Markov Chain modelling network statistics.

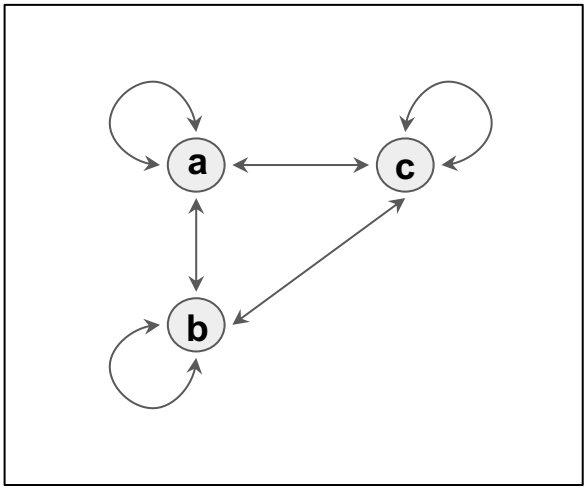


Figure 6: Network topology consisting of 3 nodes ‘a’, ‘b’ and ‘c’

from * to *	a	b	c
a	1xx	0xx	01x
b	1xx	1x1	011
c	x1x	x0x	111

Table 2: Forwarding logic of Network shown in Figure 6. Each entry is the packet wildcard match needed to forward a packet along the edge from node in row to node in column.

$p_{i,j}$	a	b	c
a	0.1	0.4	0.5
b	0.5	0.2	0.3
c	0.3	0.6	0.1

Table 3: Probability matrix of the Markov Chain representing network traffic of Network in Figure 6

We will work through the algorithm for an arbitrary input packet ‘xxx’ with start node ‘a’ and destination node ‘c’. Initially, the cumulative probability is zero and the priority queue only contains node ‘a’ with candidate probability 1 (Figure 7). Node ‘a’ is then popped from the priority queue and transformed using the forwarding logic and the markov chain (Figure 8). At this point, node probabilities are calculated by multiplying the parent node candidate probability with each child’s transition probability. Also, terminated paths are identified and the cumulative probability is updated. Only path ‘a->c’ is terminated and its probability is used to update cumulative probability. Candidate subpaths ‘a->a’ and ‘a->b’ are inserted along with their probabilities (and headers) into the priority queue. Figure 9 shows the result of the queue pop operation of highest probability sub-path ‘a->b’ and the result of its transformation. All its children are terminating paths with ‘a->b->a’ and ‘a->b->b’ being unreachable and ‘a->b->c’ reaching the destination. The resulting cumulative probability is 0.9 equal to the probability threshold, which will terminate the algorithm.

It is worth mentioning that our algorithm does not terminate at loops by default, as not all loops are vicious and infinite due to possible packet rewrites in forwarding logic. Moreover, for probability thresholds less than 1, the algorithm will eventually terminate even if loop detection is disabled.

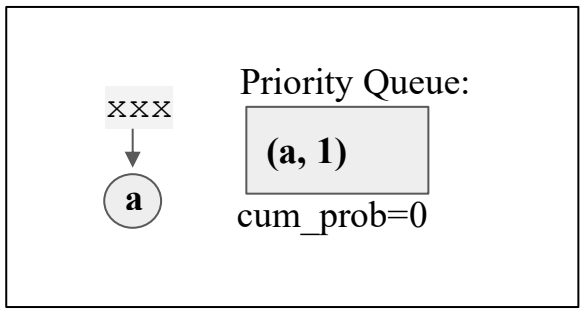


Figure 7: Initial setup of the traversal

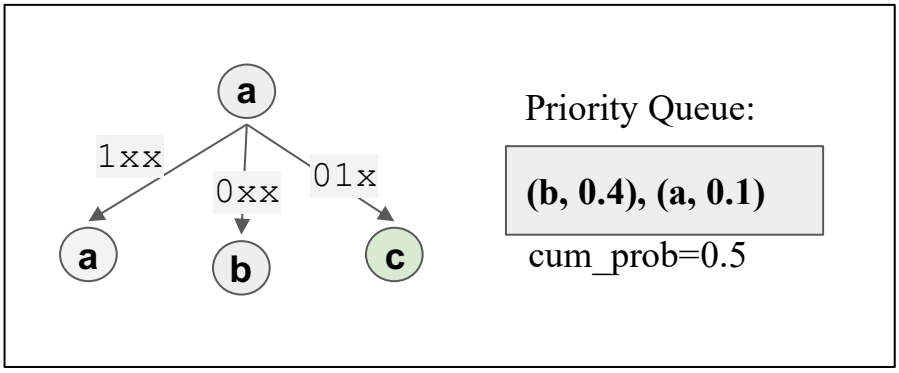


Figure 8: Result of the 1st queue pop and transformation

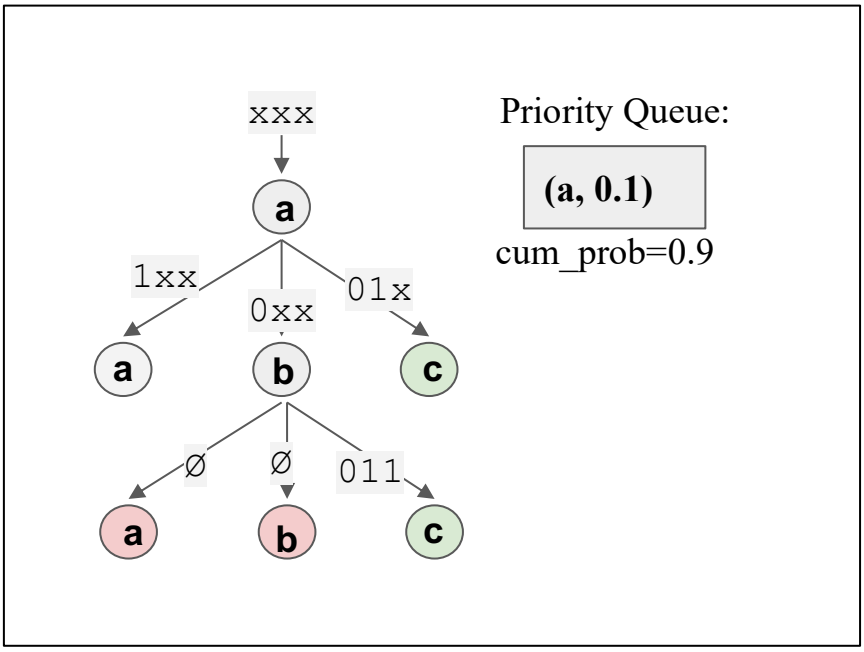


Figure 9: Result of the 2nd queue pop and transformation

7. Correctness Proof

To prove that the proposed algorithm works, we need to satisfy 2 conditions:

1) The algorithm should cover the probability threshold correctly
2) The algorithm should evaluate highest probability sub-paths first in the traversal

a. Proof of Claim 1: The algorithm should cover the probability threshold correctly

The algorithm should calculate and update the cumulative probability correctly according to (sub-path) Markov probabilities. It terminates when the cumulative probability becomes equal or greater than the probability threshold.

The branching traversal of the network graph from the source port until a terminating node is reached follows a tree diagram where the root is the source port and children are next hops from their parent. Children can either terminate or continue branching.

Each node in the tree has a probability. This probability starts at 1 and keeps getting multiplied with hop probabilities when branching. A node in the tree has a probability equal to the product of hop probabilities from the start node until reaching the current node. The resulting probability is the Markov sub-path probability.

If the node is a terminating node (unreachable or destination), its probability is added to the cumulative probability. The cumulative probability tracks the terminated path probabilities. The cumulative probability will add up to 1 if all paths have been traversed, since a parent node's probability will be equal to the sum of child probabilities.

b. Proof of Claim 2: The algorithm should evaluate highest probability sub-paths first in the traversal

The branching traversal keeps unterminated (tree) nodes in a priority queue and pops highest probability paths first. This ensures that if a terminated path is identified, it will be the highest probability path out of all possible options.

8. Running Time

a. Transformation Cost

Let **transform_cost** be the cost of processing one transformation candidate. (Figure 10)

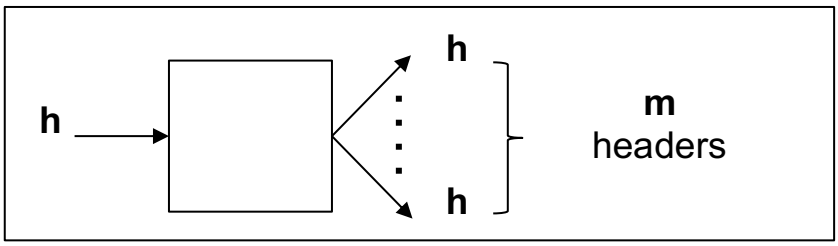


Figure 10: Forwarding transformation at a port

transform_cost = O(F(h, r) * m) <= O(F(h,r) * n)

Where:

- **h** is the number of bits in the header
- **r** is the number of rules in a device
- **m** is the number of edges in a node
- **F(h, r)** is the cost of matching and transforming a header at a port/node

b. Number of Transformations and Candidates

Let ***num_transformations*** be the number of nodes transformed.

Let ***num_candidates*** be the number of node candidates generated in the traversal branching tree (Figure 11).

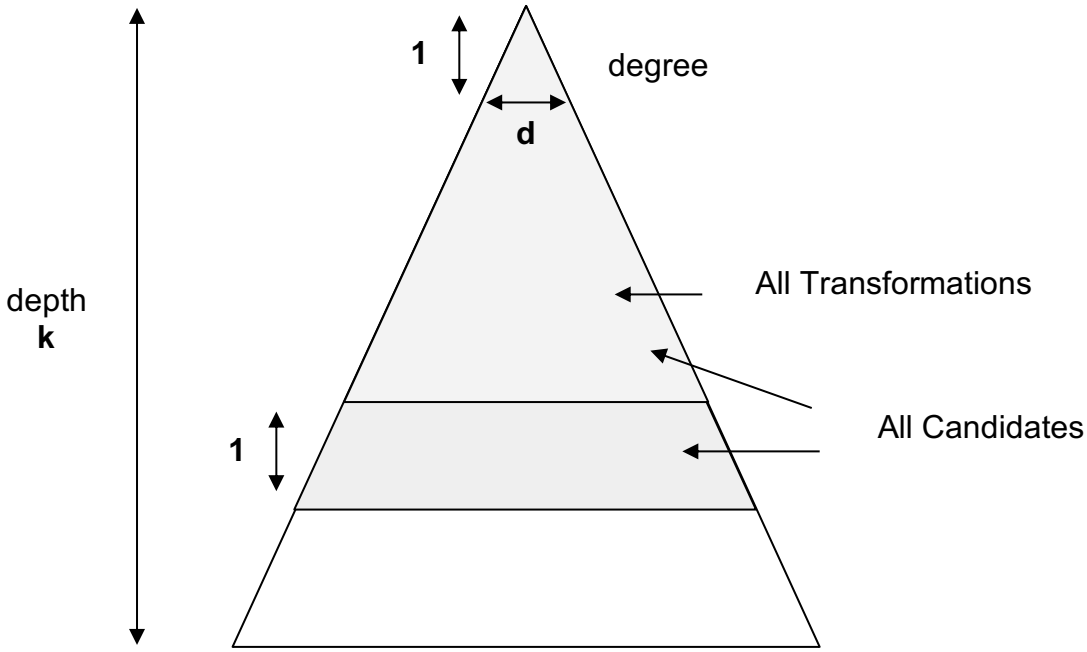


Figure 11: Branching Traversal Tree

The maximum number of elements in a tree of degree **d** and depth **k** is **O(d^(k-1))**.

The number of candidates ***num_candidates*** can at most be equal to the size of the whole branching traversal tree. In many cases, the algorithm does not need to process all possible nodes and terminates before processing the whole tree. Let the **candidate_ratio** be the ratio, out of all possible hypothetical tree nodes, of processed candidates. Note that **candidate_ratio** is not a constant but varies with network traffic, forwarding logic, and network size, where it could reduce num_candidates by orders of magnitude.

num_candidates = O(candidate_ratio * d^(k-1))

The number of transformations **num_transformations** can at most be equal to the number of candidates. A lower bound is not guaranteed as extra assumptions will be required.

$$num\_transformations <= num\_candidates = O(candidate\_ratio * d^{(k-1)})$$

c. Traversal Cost

Let **traversal_cost** be the cost of traversing through the network using a priority queue.

The cost of $queue\_push = O(1)$

The cost of $queue\_pop = O(\log(queue\_size)) = O(\log(num\_candidates))$

$$traversal\_cost = transform\_cost * num\_transform * queue\_pop + queue\_push * all\_candidates$$

$$= O(F(h, r) * n * candidate\_ratio * d^{(k-1)} * \log(candidate\_ratio * d^{(k-1)})) + O(candidate\_ratio * d^{(k-1)})$$

d. Candidate Ratio

We will focus on statistical properties and their relation to running time improvement. We assume no early terminations take place. Early terminations happen due to unreachable packets due to forwarding logic (as opposed to traffic statistics) which halt the branching traversal at depths less than the maximum **k**.

Let **P** be the probability distribution of a single transformation step's output. It assigns a probability to each potential forwarding outcome. (Figure 12)

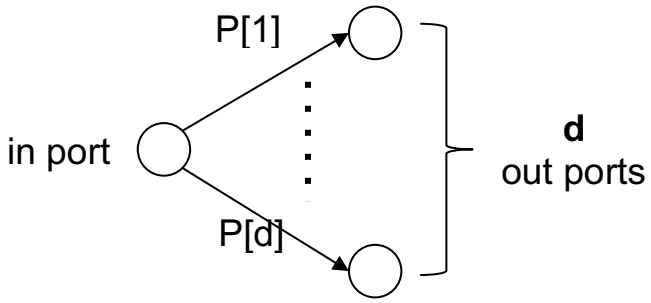


Figure 12: Probability distribution of a transformation step

The probability distribution after **k** recursive transformations is **P^k** with support (number of elements over which the discrete distribution is defined) **d^k.** Which could be interpreted as the probability of a sequence of **k** iid random variables each representing a transformation step.

We are interested in the behavior of **P^k**. Specifically, we are interested in how our tradeoff behaves on a distribution raised to a power. Our tradeoff can be formulated as a function **Fratio** from a distribution **P^k** and a probability threshold **Pth** to the ratio of the minimum number of elements, out of all possible elements, needed from **P^k** to cover **Pth**.

$$\boldsymbol{Fratio(P, k, Pth)} =$$

$$\frac{\boldsymbol{min}\left[\boldsymbol{size}(\boldsymbol{subset}(\boldsymbol{P^k}));\ \boldsymbol{sum}\left(\boldsymbol{Prob}(\boldsymbol{leaf});\ \boldsymbol{leaf}\ \in\ \boldsymbol{subset}(\boldsymbol{P^k})\right)\ \geq\ \boldsymbol{Pth}\right]}{\boldsymbol{sup}(\boldsymbol{P^k})}$$

Fratio gives us the ratio of tree leafs needed to satisfy our threshold, which asymptotically bounds the running time of the algorithm.

Empirical observations in the below graph (Figure 13) show the evolution of **Fratio** as **k** is increased. For different base distributions **P** of varying skewness, **Fratio** decreases exponentially fast.

The observed exponential decline is investigated in theory of large deviations [24], a branch of probability theory concerned with the asymptotic behavior of remote tails of functions applied to sequences of probability distributions. A detailed theoretical analysis is out of scope for this document.

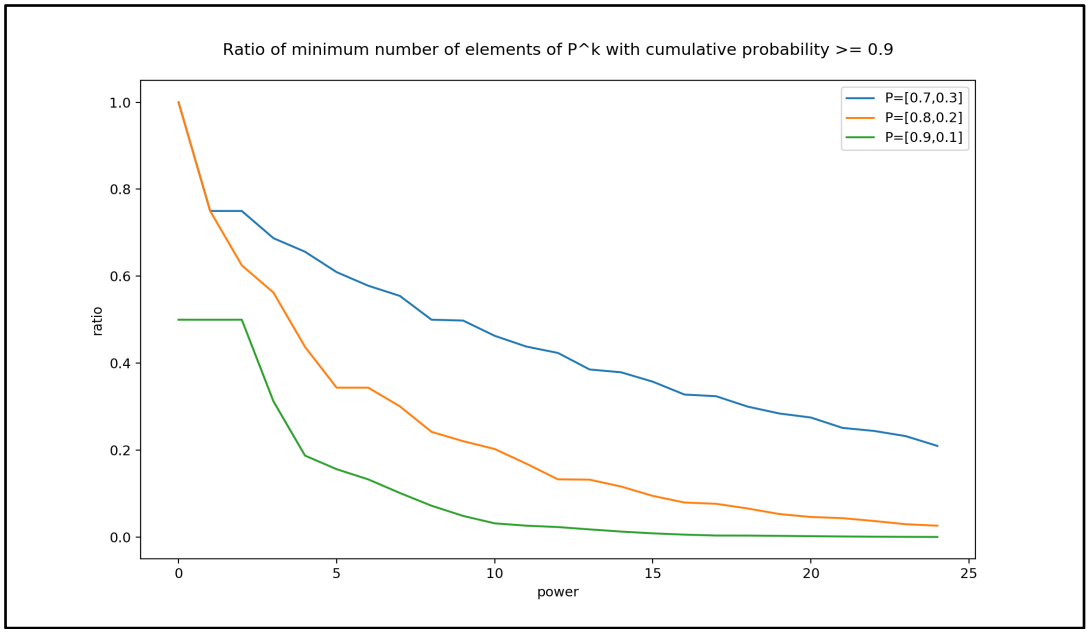


Figure 13: Fratio of P^k over multiple base distributions P

In the next section we present empirical results and investigate the behavior of our method against randomly generated networks with different complexities and statistical properties.

CHAPTER V
EMPIRICAL RESULTS

A. Implementation

Our implementation of the probabilistic traversal algorithm relied on HSA's python codebase [26] to model network headers and forwarding logic. We used their header-space objects to model packet headers, and their transfer functions to model network forwarding. We modelled the network markov chain as an **n**x**n** matrix implemented as a python dictionary (hash-table). We used the standard python implementation of priority queue in the **heapq** module.

B. Results

1. Generating Random Networks

We tested our implementation by generating random networks with random forwarding rules. We also randomly generated markov chain probabilities. We construct random network topologies by generating random G(n,p) graphs using the networkx python library. We select two parameters to randomly generate a graph; **n** the size of the network, and **p** the probability of two nodes having an edge. We use **n** and **p** to generate all other parameters. We set the packet header size to be double the number of bits needed to address all network ports $\mathbf{2 * \log_2(n^2)}$.

We construct forwarding tables by randomly creating **node_rules_n** forwarding rules, where **node_rules_n** is equal to the expected number of connected ports in the network $\mathbf{n^2 * p}$. Each forwarding rule consists of a match and forward section. We generate the match section by randomly selecting an arbitrary subset of bits (15% of

header size) to randomly set to **0** or **1**. We set the forward section of a rule to a random out-port in the node.

We generate the markov chain probability matrix by sampling **n** shuffled vectors, each of size equal to the number of neighbors of a node in the graph**,** from the function:

$$\boldsymbol{skew(x,c) = x^{-c} - 1;\ x \in]0,1[}$$

where **c** is considered a ***skewness*** parameter. We then normalize the vectors to make sure each adds up to **1**. We vary **c** to control the skewness of the resulting distribution. With **c** very close to zero **skew** becomes a constant function much like a normal distribution. With **c** taking larger values, **skew** becomes very uneven. Alternative functions can be used to sample probabilities while controlling skewness.

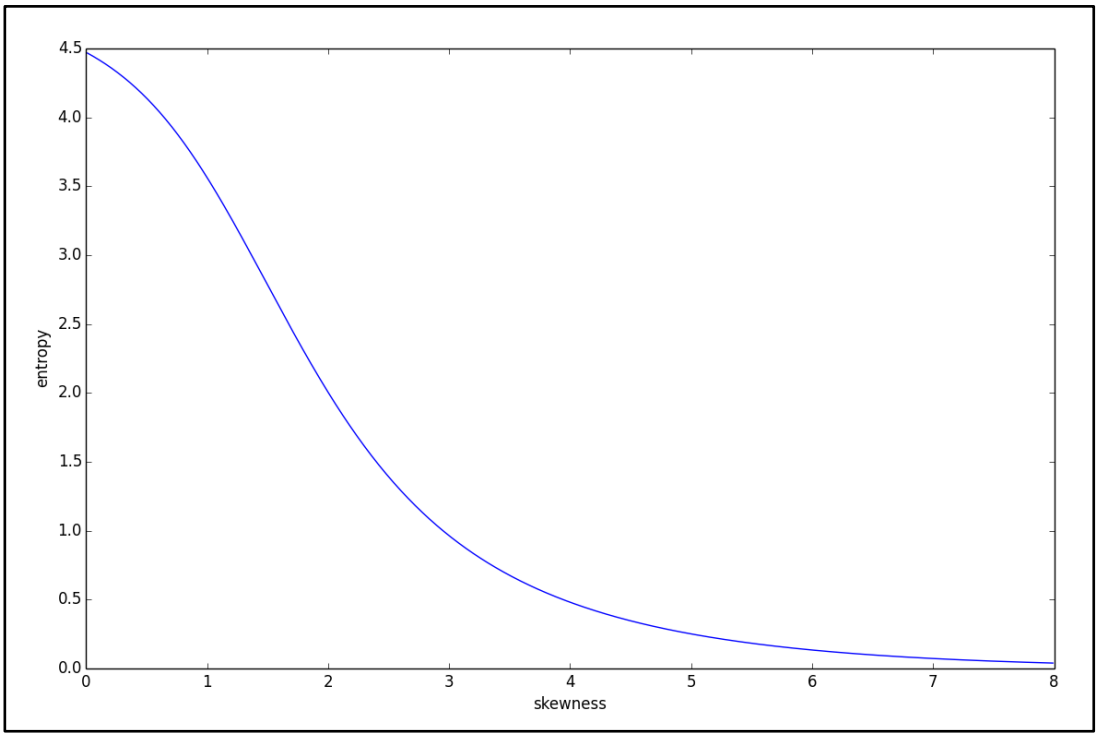


Figure 14: Entropy of 32 samples taken from skew(x,c) plotted against skewness c

Figure 14 shows how the skewness factor **c** affects the entropy of probability vectors sampled from **skew(x,c).** Entropy is almost equal to uniform entropy **5** for **c** very close to **0**, and keeps decreasing as **c** increases. Entropy can be used as a test for evenness/skewness of probability distributions by measuring diversity where it is maximum for equal probabilities across a sample space and minimum for extreme concentration of probabilities in limited parts of it.

2. Convergence of Cumulative Probability

Cumulative probability tracks the covered traffic volume throughout the progress of the algorithm. It is used to terminate the algorithm when it reaches the target threshold (cutoff) probability. Figure 15 shows how the cumulative probability progresses at each step in the algorithm. As expected, cumulative probability grows very quickly in the beginning and becomes slower as time progresses. This is due to the fact that path probability tends to be concentrated at few paths in the network.

The shape of the plot is that of a negative exponential of the form $y = 1 - e^{-\alpha \cdot x^{\beta}}$. Figure 16 shows the result of fitting a negative exponential function to the evolution of cumulative probability.

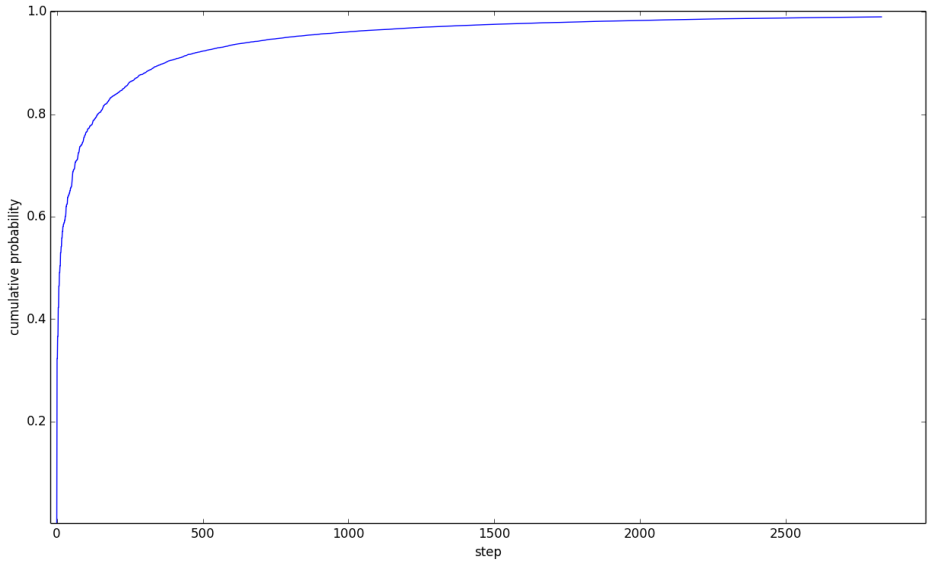


Figure 15: Cumulative path probability (below 0.99 cutoff) as the algorithm progresses (n=30, p=0.99, c=0.6)

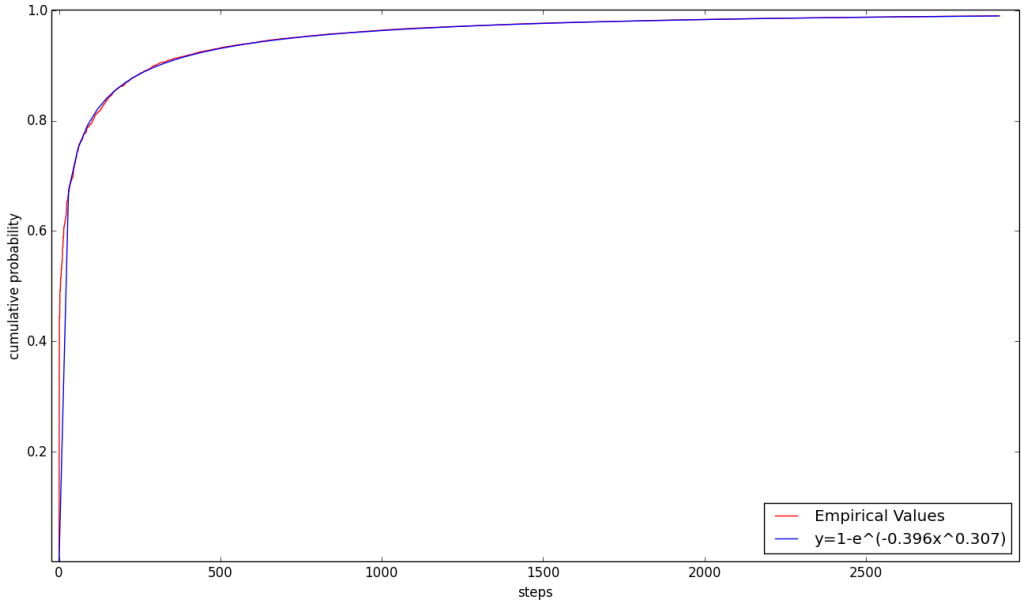


Figure 16: Fitting cumulative path probability evolution plot to negative exponential function (n=30, p=0.99, c=0.6)

The speed of convergence is directly proportional to α and β. Due to the fact that two parameters define the fitting function, it is not possible to use both of them at the same time to compare the convergence of two different instances of the fitting function.

The derivative of y is $y' = \alpha\beta x^{\beta-1}e^{-\alpha x^{\beta}}$. Identifying the speed of convergence through the derivative is a tricky matter as taking the derivative at zero does not yield an answer, as it will actually be infinite ($\beta < 1$). Moreover, taking the derivative at any other point is arbitrary and cannot be used as grounds for comparing different instances of this function.

Another more fundamental problem is our assumption of the validity of y over all possible results of the algorithm. Due to the above reasons, we will use the number of steps needed to reach a cutoff probability threshold of 0.99 as the rate of convergence. This definition avoids making assumptions about a fitting function and gives a practical estimate for the behavior of the algorithm under different circumstances.

3. Varying Complexity

Figure 17 shows the effect of varying two network complexity parameters on the growth of cumulative probability in terms of number of steps. The number of steps needed to reach probability threshold of 0.95 increases with network complexity **n** & **p**, with the number of nodes **n** contributing to most significant increase in running time. It is important to note that in our setup header size ($\mathbf{2 * \log_2(n^2)}$) grows with **n** and the number of forwarding rules ($\mathbf{n^2 * p}$) grows quadratically with **n** and linearly with **p**.

Another observation can be made: the shape of the curve under different conditions takes two forms. It either grows quickly and plateaus as in the case of { n=20, p=0.8 } or grows, more-or-less, uniformly as in the case of { n=30, p=0.8 }. The first shape, of sudden increase and later plateau, reflects a form of skewness where the majority of the probability is concentrated in a group of paths, while the remaining

probability is evenly spread, while the second shape reflects a more global distribution of skewness across the whole network. The β factor in the fitting function seems to be directly related to the relative concentration of path probabilities with higher β correlated with more uniformity in speed of growth and less concentration of skewness.

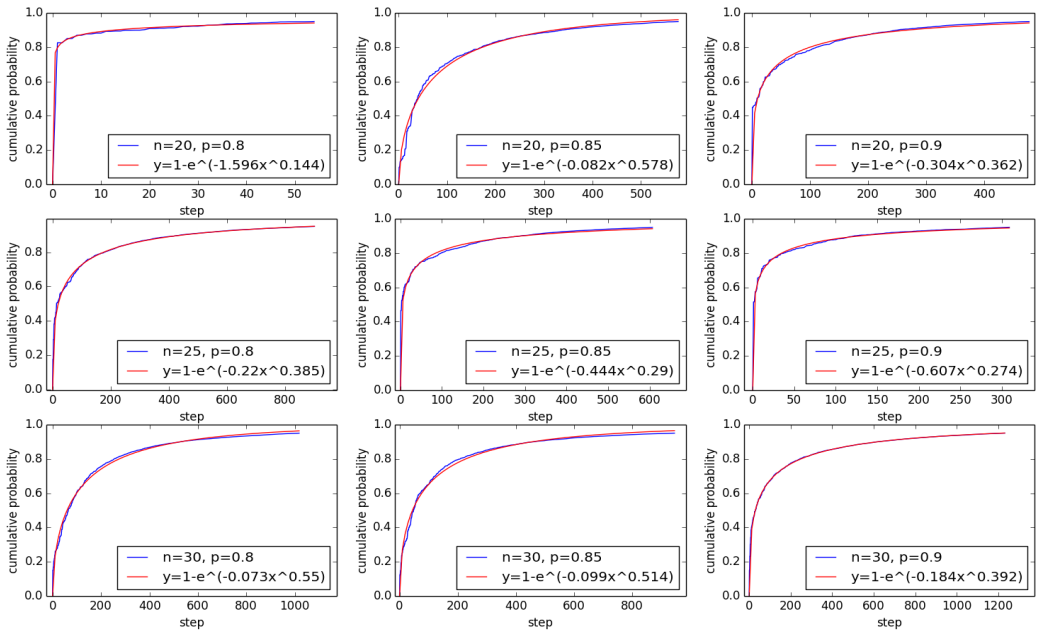


Figure 17: Cumulative Probability Plots for Different Network Complexities (c=0.6, pth=0.95)

4. Varying Skewness

Figure 18 shows the effect of skewness on the running time (in seconds) and the number of steps. As expected, as we increase skewness the algorithm covers the target threshold 0.95 faster and with less steps. This is due to the concentration of probability in fewer and fewer paths in the network, which is taken advantage of by our probabilistic traversal algorithm.

Due to the random assignment of probabilities in the markov chain, there is a variability in the running time and number of steps needed to reach the threshold. Nonetheless, this variability follows a decreasing pattern as we increase skewness. Each

datapoint in Figure 18 corresponds to randomly generated traffic distribution based on the skewness value. We generated 15 samples at each skewness point and averaged out the resulting running time and number of steps per skewness point.

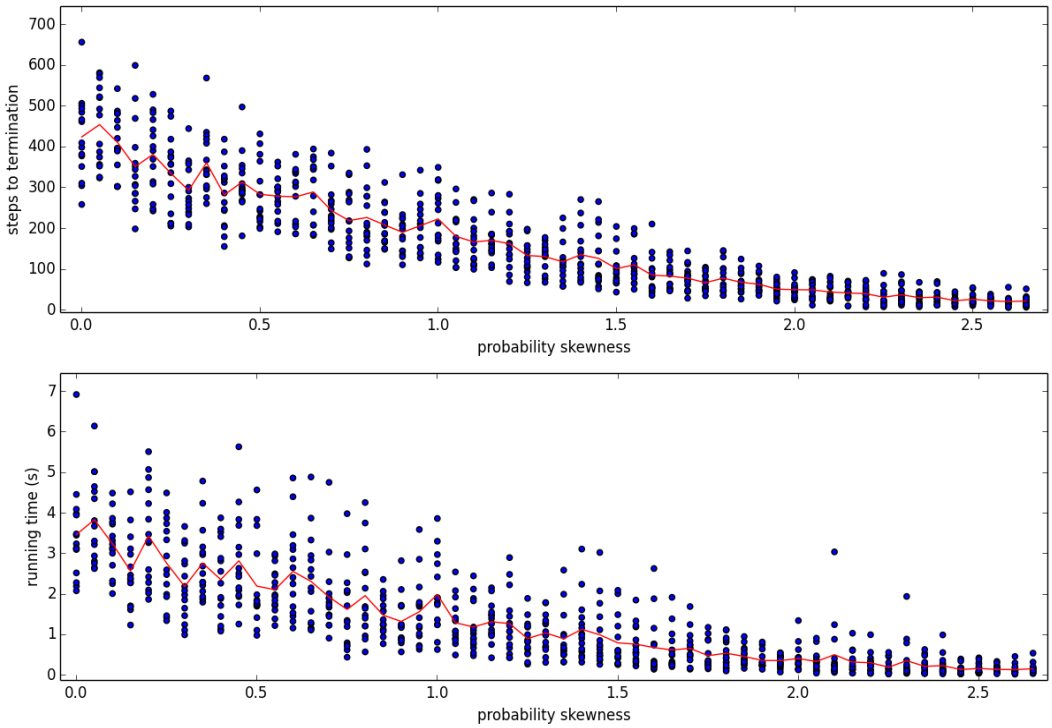


Figure 18: Running time and steps to termination (0.95 threshold) vs multiple values of probability skewness (n=20, p=0.99)

5. Comparison with HSA

Our algorithm differs from the original HSA reachability algorithm in one important aspect. We do not explicitly check for loops and avoid them. We can process the same node/port multiple times. Our implementation relies on intermediate node probabilities taking care of this issue, as the probability of going through a cycle will exponentially decrease (geometric distribution) with time. The markov chains we deal with are connected and do not have islands of isolated groups of nodes. This means that the probability of absorption by the destination nodes and termination nodes is 1. This

does not necessitate that the algorithm will terminate in a finite amount of time if the probability threshold is set to be 1, as the probability of absorption is the probability of *eventually* reaching an absorbing state.

To make our algorithm comparable to the original HSA algorithm, we added a check to avoid processing the same port multiple times in the same path.

Figures 19 and 20 shows how our algorithm compares to the original HSA reachability algorithm under different conditions. Multiple important observations can be made. First, as we increase network size our algorithm at probability threshold equal to 1 needs increasingly more time compared to HSA's reachability algorithm. This can be explained by the extra log(n) factor in our asymptotic running time due to the use of a priority queue. Second observation, is that as we increase network size our algorithm saves more time at probability thresholds 0.95 and 0.99. This can be explained by the increasing concentration of high-probability paths for distributions P^n as n increases.

	HSA	pth=1	pth=0.99	pth=0.95
n=10	100.00%	87.59%	28.19%	13.50%
n=15	100.00%	101.40%	38.83%	18.28%
n=20	100.00%	105.62%	10.99%	2.68%
n=25	100.00%	106.83%	2.45%	0.57%

Figure 19: Table showing running time of the original HSA algorithm and our algorithm (p=0.99, c=1.0) at different probability thresholds pth and different network sizes n normalized by running time of the original HSA algorithm against the same randomly generated network. (Intel Core i5 3.1 GHZ - 8GB RAM)

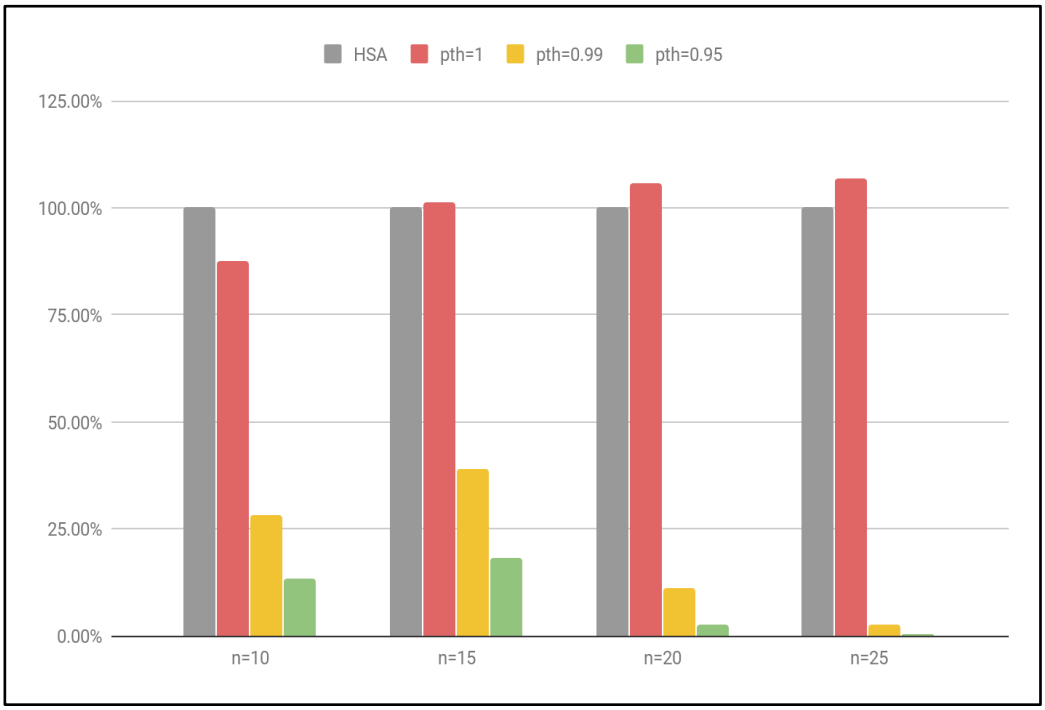


Figure 20: Bar graph visualization of comparison table in Figure 19

We were able to achieve 99% traffic coverage in 2.45% (and 95% traffic coverage in 0.57%) of the time needed for full coverage by HSA for a randomly-generated network with n=25, p=0.99 and skewness c=1.

CHAPTER VI
CONCLUSION

In this document, we motivated and proposed a tradeoff between running time and traffic coverage (probability of correctness) for the problem of identifying reachability between two network endpoints. Our proposed solution converts network traffic statistics into a Markov Chain and applies a prioritized traversal algorithm to minimize running time while ensuring traffic coverage according to a user-defined probability threshold. Our solution makes it feasible to check for reachability for cases when network forwarding complexity becomes too large for complete solutions. It allows approximations to take advantage of statistical properties of networks instead of relying on general-purpose heuristics and tricks present in SAT-solvers and similar tools.

We tested our solution against randomly generated networks with randomly generated traffic statistics. As expected, the prioritized traversal algorithm saves on running time by taking a small fraction of what it would take for a complete solution. The amount of time-saving is correlated to the skewness of traffic distribution, proving our algorithm's efficacy in tracking highest-probability paths and utilizing it to save on computation.

For higher network complexity, our algorithm costs slightly more than the reference implementation in HSA [3] for probability threshold equal to 1 (full traffic coverage), but the amount of time-saving for the probability threshold less than 1 becomes smaller and smaller. This phenomena is supported by another independent observation made in Chapter 4 Section D8 stating that total network skewness increases for increasing network complexity, leading to the conclusion that our algorithm is even better suited for high-complexity settings.

A. Future Directions

This research project can be expanded in multiple directions. Our solution focused on applying the correctness-vs-performance tradeoff to reachability in HSA [3], but this is only one example of an adaptation. The tradeoff can be expanded to cover verification questions other than reachability. Different verification methods can also be explored for possible extension with our proposed tradeoff. One possible adaptation is to make use of optimization features in SMT Solvers such as Microsoft's Z3 [25], and to formulate the tradeoff as an objective function for an optimization, with forwarding logic and reachability questions encoded as SMT specifications.

Another direction is to explore traffic models beyond global (first-order) Markov Chains. Network traffic can have statistics beyond the stateless model of a single-network Markov Chain, which matches traffic counters at individual network ports irrespective of specific header information. Two possible approaches for enhancing the accuracy of traffic model representation exist:

(1) A traffic model can be built per (symbolic) header to capture specific statistics and use them in creating a tradeoff for reachability of specific headers. The challenge in this case would be that of making use of partial statistics from different traffic counters, of varying target packets, where gaps need to be accounted for and covered.

(2) Higher-order Markov Chains can be used to capture statistics collected across multiple devices and ports. This would be useful in enhancing the accuracy of the traffic model to track streams of packets at multiple devices instead of at single nodes. In order to avoid unnecessary state expansion, variable-length sub-

path probabilities can also be explored and used to build an algorithm that makes use of ‘whatever’ statistics are collected from a network.

Additionally, stateful network models and their verification can be explored and adapted to our tradeoff. This would expand our solution to firewalls and stateful middleboxes, in addition to stateful routing algorithms and protocols.

REFERENCES

[1] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey and S. King, “Debugging the data plane with anteater”, ACM SIGCOMM Computer Communication Review, vol. 41, no. 4, p. 290, 2011.

[2] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, “A General Approach to Network Configuration Verification,” Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM 17, 2017.

[3] P. Kazemian, G. Varghese and N. McKeown, “Header space analysis: Static checking for networks.”, Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012.

[4] N.P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, “Checking beliefs in dynamic networks.”, Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015.

[5] H. Yang, “Efficient verification of network reachability properties,” 2013 21st IEEE International Conference on Network Protocols (ICNP), 2013.

[6] H. Yang, "Efficient verification of packet networks", Ph.D, 2015.

[7] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” ACM SIGCOMM Computer Communication Review, vol. 42, no. 4, p. 467, 2012.

[8] A. Horn, A. Kheradmand, and M. Prasad, “Delta-net: Real-time network verification using atoms.”, Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) pp. 735-749, 2017.

[9] Y. Boufkhad, L. Linguaglossa, F. Mathieu, D. Perino, and L. Viennot, “Efficient Loop Detection in Forwarding Networks and Representing Atoms in a Field of Sets.” arXiv preprint arXiv:1809.01896, 2018.

[10] Y. Boufkhad, R. De La Paz, L. Linguaglossa, F. Mathieu, D. Perino, and L. Viennot, “Forwarding Tables Verification through Representative Header Sets.”, arXiv preprint arXiv:1601.07002, 2016.

[11] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T.D. Millstein, “A General Approach to Network Configuration Analysis.”, In NSDI pp. 469-483, 2015.

[12] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese, “Efficient Network Reachability Analysis Using a Succinct Control Plane Representation.”, In OSDI, pp. 217-232, 2016.

[13] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, “A balance of power: Expressive, analyzable controller programming.”, In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pp. 79-84, 2013.

[14] C. J. Anderson, N. Foster, A. Guha, J. B Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic foundations for networks.”, In ACM SIGPLAN Notices, Vol. 49, No. 1, pp. 113-126, 2014.

[15] D. Kozen, “Kleene algebra with tests.” ACM Transactions on Programming Languages and Systems (TOPLAS), 19(3), pp. 427-443, 1997.

[16] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, “Probabilistic netkat.”, In European Symposium on Programming Languages and Systems, pp. 282-309, Springer, Berlin, Heidelberg, 2016.

[17] J. McClurg, H. Hojjat, N. Foster, and P. Černý, “Event-driven network programming.”, In ACM SIGPLAN Notices, Vol. 51, No. 6, pp. 369-385, 2016.

[18] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, “An assertion language for debugging SDN applications.” In Proceedings of the third workshop on Hot topics in software defined networking, pp. 91-96, 2014.

[19] S. Zhang, “Computer Network Verification and Management using Constraint Solvers”, Doctoral dissertation, Princeton University, 2016.

[20] S. Zhang and S. Malik, “SAT based verification of network data planes.”, In International Symposium on Automated Technology for Verification and Analysis, pp. 496-505, Springer, 2014.

[21] G. G. Xie, J.Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, “On static reachability analysis of IP networks.”, In INFOCOM 2005 24th Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings IEEE Vol. 3, pp. 2170-2183, 2005.

[22] S. Brin, and L. Page, “The anatomy of a large-scale hypertextual web search engine.”, Computer networks and ISDN systems, 30(1-7), pp. 107-117, 1998.

[23] M. Armstrong and F. Richter, “Infographic: Netflix is Responsible for 15% of Global Internet Traffic,” Statista Infographics, 09-Oct-2018. [Online]. Available: https://www.statista.com/chart/15692/distribution-of-global-downstream-traffic/. [Accessed: 13-Jan-2020].

[24] J. M. Swart, “Large Deviation Theory.”, 2012.

[25] L. D. Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science, pp. 337–340, 2008.

[26] P. Kazemian, “Header Space Library (Hassel).” [Online]. Available: http://stanford.edu/~kazemian/hassel.tar.gz. [Accessed: 13-Jan-2020].

[27] M. Nagl, “Graph-Theoretic Concepts in Computer Science.” Berlin, Heidelberg: Springer-Verlag, 1990.